Automated Backend Selection for ProB Using Deep Learning*

Jannik Dunkelau[1]✉, Sebastian Krings[1,2], and Joshua Schmidt[1]

[1] Heinrich-Heine-University, Düsseldorf, Germany
{dunkelau,krings,schmidt}@cs.hhu.de
[2] Niederrhein University of Applied Sciences, Mönchengladbach, Germany

Abstract. Employing formal methods for software development usually involves using a multitude of tools such as model checkers and provers. Most of them again feature different backends and configuration options. Selecting an appropriate configuration for a successful employment becomes increasingly hard. In this article, we use machine learning methods to automate the backend selection for the ProB model checker. In particular, we explore different approaches to deep learning and outline how we apply them to find a suitable backend for given input constraints.

Keywords: Formal Methods · Model Checking · Automated Configuration · Deep Learning.

1 Introduction and Motivation

The typical workflow when using formal methods consists of requirements engineering, writing specifications and analysing them using proof techniques and model checking. For all three tasks, a variety of tools exists, each featuring a multitude of configuration options.

However, selecting the best tool for a task or choosing the optimal configuration is not trivial, even for domain experts. This is in alignment with the *No Free Lunch* theorem [42,43], and also affirmed by empirical evaluation on verification tasks as shown by Krings et al. [23] for the B method.

For instance, when solving constraints involving relations over sets, a SAT solver often provides a better performance than an SMT solver or a solver based on constraint logic programming [25]. However, especially when using integers, an SMT solver is often preferable to a SAT solver: a SAT solver needs to restrict the bitwidth which might result in integer overflows. Furthermore, an SMT solver directly supports integers without translation into propositional logic.

However, one cannot easily generalise on which constraints different solvers are efficient as it is impossible to set up universal selection rules. Consequently, we decided to use statistical models in order to predict the constraint solver which most likely provides the best performance for a given constraint.

* Computational support and infrastructure was provided by the "Centre for Information and Media Technology" (ZIM) at the University of Düsseldorf (Germany).

In this paper, we present our work towards using machine learning for the automated configuration of ProB [29–31], a model-checker and constraint solver for the B method [1]. In particular, we try to automate the selection of backends used for constraint solving in various places, e.g. for computing suitable parameters to operations or for symbolic model checking.

2 Related Work

Using heuristics, statistics or machine learning for configuration or selection of algorithms has been tried both for theorem provers and constraint solvers.

For the SETHEO theorem prover [28], Goller [16] employed folded architecture networks [17] to learn heuristic evaluation functions, i.e. performance measures for individual inference steps within a proof. While the results were promising, experiments were run for simple problems, with Goller stating that 'the next step is to experiment in a more realistic application domain' [16].

In the work of Bridge [7], support vector machines were used to automate the heuristic selection for the theorem solver E [39]. Here, the problem was limited to first order logic with equality. Bridge was able to improve E with his heuristic selection as it outperformed fixed heuristics as well as the already implemented auto-mode of E.

In the works of Healy [20, 21], an SMT solver portfolio was conducted for Why3. Why3 [4, 13] is a platform for deductive program verification, which provides its own language, WhyML, to specify a program and bindings to multiple different SMT solvers for the formal verification. The solver selection was done via decision trees which predicted the runtime needed for each solver constructing a ranking from fastest to slowest. The fastest solver is then proposed for verification of a given proof obligation.

In contrast to the works of Goller or Bridge, this article concerns itself with the higher-level language B. Besides first order logic with equality (c.f. Bridge's work) B also captures multiple different theories as is briefly outlined in the upcoming Section 3, including functions, sets, quantifiers, and non-deterministic assignments.

In contrast to Healy, this article concerns itself with a classification problem rather than a regression task. Further, *unknown* is used as its own class to capture instances neither of the involved backends is able to provide an answer for. Although an unknown solvability of a given proof obligation is implicitly detected in Healy's work (predicted runtime for all backends is greater than the timeout the data was generated for), having an actual probability of how likely a backend would return no answer for a given constraint might be more expressive. For instance, one might intervene early rather than sequentially querying each solver in the ranked list, depending on a probability threshold. Further, the calculated probability distribution also provides an implicit ranking ordering the backends by descending probabilities.

3 Primer on B and ProB

B [1] is a formal verification language for specifying, designing, and coding software systems as well as for performing formal proof of their properties. It follows the correct-by-construction approach and is based on first-order-logic and the Zermelo-Fraenkel set theory with the axiom of choice [14, 15]. Further, it makes use of general substitution for state modifications, and of refinement calculus [2, 3] to describe models at different levels of abstraction [8].

B machines consist of variable and type definitions as well as possible initial states. By defining machine operations, one is able to specify transitions between states. These transitions consist of substitutions, which may be non-deterministic depending on the level of abstraction. An operation can have a precondition enabling or disabling execution based on the current state. To ensure certain behaviour, the user can define machine invariants, i.e. safety properties that have to hold in every reachable state. The correctness of a formal model hence refers to the preservation of the specified properties in each reachable state.

For instance, consider the invariant $\forall x \in S \cdot (\exists n \cdot x = 2^n)$ for a manually assembled set S, and an operation with parameter n which adds 2^n to the set if not yet present: `op(n) = PRE n:NAT & 2**n/:S THEN S:=S\/{2**n} END`. The invariant now poses a constraint onto S which has to be satisfied in each reachable state including the states transitioned into by executing `op`.

Using Atelier B [11] or ProB [29–31] one can verify a B model and analyse its state space. In particular, ProB allows the user to animate formal models, providing a model checker and constraint solver. ProB's kernel [29] is implemented in SICStus Prolog [10], using the CLP(FD) finite domain library [9]. Alternatively, the backend can be substituted with a binding to a different solver. For one, a constraint solving backend based on Kodkod [41] is available [34]. Furthermore, an integration with the SMT solver Z3 [12] (connected to ProB as outlined in [24]) can be used to solve constraints.

The different backends have their own strengths and weaknesses. As the name suggests, CLP(FD) is particularly strong when dealing with variables having finite domains. On unrestricted problems, CLP(FD) can fail even on trivial problems such as $X < Y \wedge Y < X$, whereas it easily detects unsatisfiability if we restrict the domains of X and Y.

The Kodkod backend performs well on problems involving relations between different sets. However, it does not support the full range of constructs available in B. Consequently, ProB includes a fallback to the CLP(FD) backend for untranslatable parts.

In contrast to CLP(FD), the SMT-based backend performs well on unrestricted problems. Its particular strength is detecting unsatisfiability, while it does not perform as well for model finding, i.e. for finding variable valuations for satisfiable constraints. Again, the backend can be used on its own, i.e. with Z3 as the only solver involved. Comparable to Kodkod, Z3 can also be used together with the CLP(FD) backend in an integrated solving procedure as described by Krings et al. [24]. As we wanted to understand what influences the performance of the different backends, we used the standalone backend in the following.

4 Machine Learning on B Constraints

In this article, we consider three different classification problems:

Singular ProB Classification Given a single constraint p, is it possible to classify whether ProB's default backend will be able to determine whether p has a solution?

ProB+Kodkod Classification Given a single constraint p, is it possible to classify whether the default backend or the one based on Kodkod can determine satisfiability of p faster than the other, or if both will answer with unknown?

ProB+Z3 Classification Given a single constraint p, is it possible to classify whether the default backend or the one based on Z3 can determine satisfiability of p faster than the other, or if both will answer with unknown?

Initially, we aimed at creating an expert system able to propose a suitable backend for a given task or constraint. However, soon we realised that we lack deeper understanding of why a solver performs better on certain tasks than another. Further, the assembly of an expert system of this magnitude of complexity requires an unreasonable amount of pure programming work presumably consisting of myriads of edge-cases.

Hence, we opted for machine learning techniques. We supposed a machine learning algorithm might be capable of capturing any characteristics necessary for selecting the most suited backend for a given constraint in a fast and automated way.

4.1 Brief Introduction to Deep Learning

A deep neural network (DNN) [36,37] aims to approximate a function $y = f^\star(x)$ by learning a function $\hat{y} = f(x; W)$. Hereby, W is a matrix of parameters to be learned, whereas $\hat{y}$ is the prediction. During a training phase, the difference between the prediction $\hat{y}$ and the corresponding ground truth y is calculated and minimised by adjustments to the parameters in W. This process is called backpropagation (c.f. [38]). Internally, a neural network conducts a matrix multiplication $\hat{y} = f(x; W) = g(W^\mathsf{T} x)$ with a chosen activation function $g : \mathbb{R} \longrightarrow \mathbb{R}$. This matrix application can be layered by alternating parameters and activation functions, resulting in multiple parameter matrices $W_1, \ldots, W_n$. Such a neural network is said to have n layers, with $n - 1$ *hidden layers*. For $n > 1$, a neural network is said to be a deep neural network. Besides parameters to be learned, a neural network further depends on a selection of hyperparameters, which are manually selected configurations referring to a network's architecture that are not adjusted during training, e.g. the amount of layers n is a hyperparameter.

DNNs work over numeric vectors of fixed length d as input. Constraints however are neither vectors nor of fixed length. Hence, a translation from a given B constraint into a vector $x \in \mathbb{R}^d$ is necessary. For this, d characterizing features $x_1, \ldots, x_d$ are collected per constraint, resulting in a vector $(x_1, \ldots, x_d)^\mathsf{T} \in \mathbb{R}^d$.

Such features should be descriptive enough to characterise the sample they were collected from sufficiently for the problem at hand. As an example for classification purposes: it is easy to distinguish between cats and elephants by size, and it is impossible to do so by number of legs.

We present two different translations from constraints as they would be presented to our backends into vectors, one based on 17, the other based on 185 features. All features are manually selected characteristics incorporating knowledge of the problem domain. Note that these features are not invariant under rewriting that preserves logical equivalence, i.e. two constraints that are logically equivalent possibly result in two different feature vectors. For instance, the expressions `x+x+x` and `3*x` have different features but equivalent semantics.

Additionally, we followed an alternative approach relying on convolutional neural networks (CNNs). A CNN is a specialised kind of neural network, which processes data with a grid-like topology, most notably images [18, 27]. For this, we translate B constraints into images of a predetermined size of $n \times n$ pixels. As this translation requires no prior domain knowledge, it serves as a comparison metric for the aforementioned hand-crafted features. Internally, a set of image processing filters are learned during training, suitable to the very problem at hand. Thus, one might say that a CNN learns necessary features itself.

4.2 The Initial Set of 17 Features

The initial set of features consists of 17 values mainly consisting of the absolute numbers of certain operators used in a constraint. These features capture the usages over the different theories supported in B mainly on an operator level. For instance, the features include the number of arithmetic operators used in the constraint, the number of set operations such as set memberships, as well as the number of universal and existential quantifiers. Further, they aim to capture some properties of the contained identifiers, e.g. the amount of unique identifiers used, or the amount of identifiers with finite or infinite domains.

4.3 The Set of 185 Features

The second set of features grew bigger as we aimed to cover most of the operators and theories used in the B method more precisely than the 17 features did. This led to 185 distinct features. One of the main differences to the first set of features is that the features formulate a ratio per operator over the corresponding theory or the number of top-level conjuncts in the given constraint.

For instance, for a given theory T (such as integer arithmetic) for which B implements n operators $\mathrm{op}_1, \ldots \mathrm{op}_n$ (e.g. $+, \times, \div, mod, succ, pred$), some features are: The amount of occurrences of an operator op_i divided by the number of top-level conjuncts in the respective constraint; The amount of operator occurrences divided by the sum of all operators belonging to the theory T; The sum of all of T's operators divided by the sum of top-level conjuncts.

4.4 A Convolution Approach

Constraints are of dynamic size, i.e. their string representation can be arbitrarily long. Furthermore, they do not inherit an obvious grid-like topology. Thus, CNNs cannot be used for classification without appropriate preprocessing. However, Loreggia et al. [33] proposed a promising approach of translating SAT, CSP, or MIP problems into images, visualised by an example in Figure 1.

The translation process makes use of the fact that ASCII character codes range from 0 to 255. This makes the constraints conveniently mappable into grey scale pixels by an identity mapping. A given constraint of length N is fit into an $M \times M$ matrix with $M = \lceil \sqrt{N} \rceil$. The missing entries are filled with the value 32 (ASCII value for the space character). This matrix is a lossless mapping from the original constraint with each entry being interpretable as a grey scale pixel. The image now is scaled to an arbitrary target size of $n \times n$ pixels.

Choosing the space character as a filler is arbitrary, as it could have also been the line feed character (10) or even a null byte (0). However, the space character already has a natural occurrence in the ASCII version of a B constraint, as it may separate variables from operators and such alike. Meanwhile, neither the line feed nor the null byte occur in any of those constraints.

Although potential downscaling results in the translation no longer being lossless, Loreggia et al. express the strong believe that structure and self-similarity exposed by the instances remain throughout the scaling step: "While scaling the images incurs a high loss in information it seems to be the case that the retained structure is sufficient to address decision problems [...]" [33].

For this article, the resulting images are scaled to the sizes of 32×32 and 64×64 pixels, which hold up to 1024 and 4096 ASCII characters respectively. As the constraints have no theoretical upper limit on their size, it is not possible to find an image size that can contain any constraint losslessly (i.e. the resulting image does not need to be down-scaled). However, of the training data gathered for this article, around 64.6% of constraints fit into 1024 ASCII characters, and even 92.0% of them fit into 4096 characters. Thus, the chosen sizes compromise between fitting most of the data losslessly and having a part of the data being downscaled to decrease computational complexity.

5 Methodology

For the training of the DNNs, the training data was randomly split into three subsets. These are the training set, the validation, and the test set, consisting of 64%, 14%, and 20% of the training constraints respectively, and being pairwise distinct.

During training, the training set is fed through the model multiple times, referred to as epochs, in order to enable the model to learn to generalise over it. The performance is then measured on the validation set. Usually, the performance drops from training set to validation set, as the model has already adapted to the training data. If the performance on the validation set can keep up, this

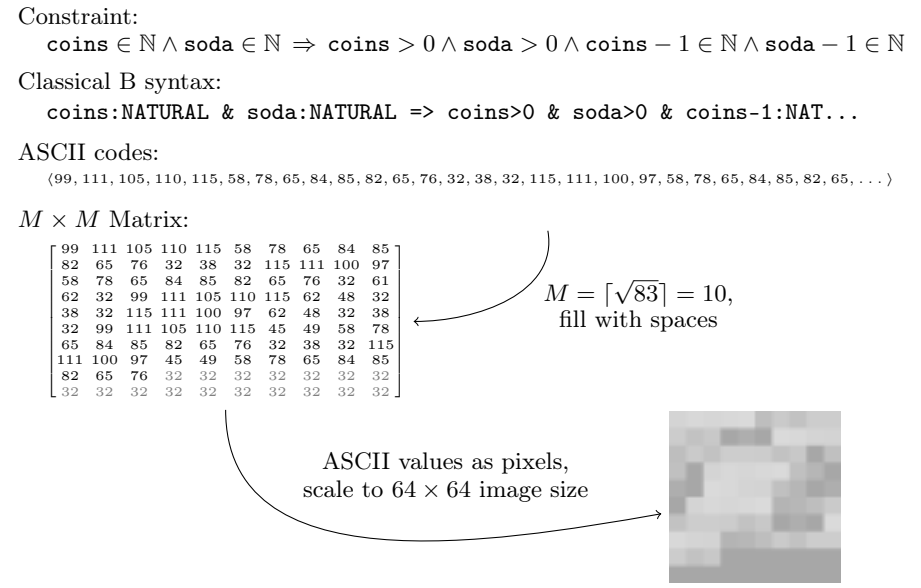


Fig. 1: The translation of constraints into images. A given constraint is interpreted as a sequence of characters. Those characters are then fit into a grid with each character's ASCII value being interpreted as a pixel value generating an image of size $M \times M$. The image is scaled to a fixed size, here 64×64 pixels.

suggests that the model actually learned to generalise. Otherwise, the model is adjusted to increase performance on the validation set after a new training step. At no point in time, the model is trained on any sample of the validation set. The test set serves as a final sanity-check for performance. Training multiple models, their performances on the validation set are implicitly dependent on the choice of hyperparameters. Testing the performance of the most promising models one can see whether the models generalise over the data, or only fit the validation set. Samples from the test set are never used for training or validation of the model. To find suitable architectures for the neural networks employed, we used a random search approach. That is, we set up ranges of possible values for any hyperparameters, and created new models by randomly choosing values from those given ranges. This was done to get a good intuition about what hyperparameters work best for the problem at hand. Found architectures were not reused for other experiments. We assumed it sensible to keep the architectures between the different experimental settings independent from each other. As they all shared the same search space, similar models should be found were suitable. To reduce the time needed for the random search, the training process for a model was terminated if it could not increase its performance for a set number of training epochs.

6 Training Data

To obtain the necessary training set, the first step was to acquire a sufficient amount of constraints. For this, we extracted invariants, preconditions of each operation, properties, and more data from 3638 B machines[3]. These were gathered from the chair of *Software Engineering and Programming Languages* at the University of Düsseldorf stemming from different application areas and thus varying in size and complexity.

These gathered constraints were then used to construct more complex ones. This served two purposes. Firstly, the amount of examples at hand was increased. Secondly, it ensured the presence of constraints which are harder to solve. In total, the generation yielded 321,742 constraints. Measured on number of characters in their ASCII representation, the average constraint length is 1,377.66 characters, with a minimum of 5 and a maximum of 15,383. The length distribution throughout the constraints is shown in Figure 2.

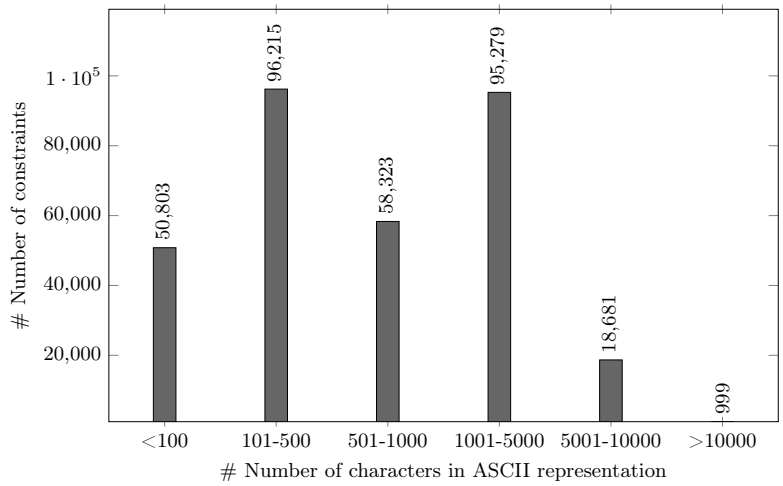


Fig. 2: Length distribution of the constraints in the training data.

Each constraint is annotated with the information of which backend is able to find a solution or show unsatisfiability. For this, we measured the average run time needed in three runs per backend. As timeout, ProB's default setting of 2.5 seconds per constraint was used. The resulting dataset can be found on GitHub[4]. From a classification point of view, constraints for which a backend can determine whether they are satisfiable or unsatisfiable belong to the class of this backend's *positive samples*, while those for which a backend is unable to

[3] www3.hhu.de/stups/downloads/prob/source/ProB_public_examples.tgz
[4] https://github.com/hhu-stups/prob-examples-metadata

determine satisfiability, for example, due to a timeout or unsupported constructs, belong to the class of its *negative samples*.

Analysing the generated training data reveals that the data poses the class imbalance problem as shown in Figure 3. The class imbalance problem [22, 26] occurs in a training set for classification with a significant disproportion of class representation. Ideally, the classes are equally distributed. Otherwise, one runs at risk to train a dummy-classifier predicting only the stronger presented classes but classifying poorly elsewise [26, 32].

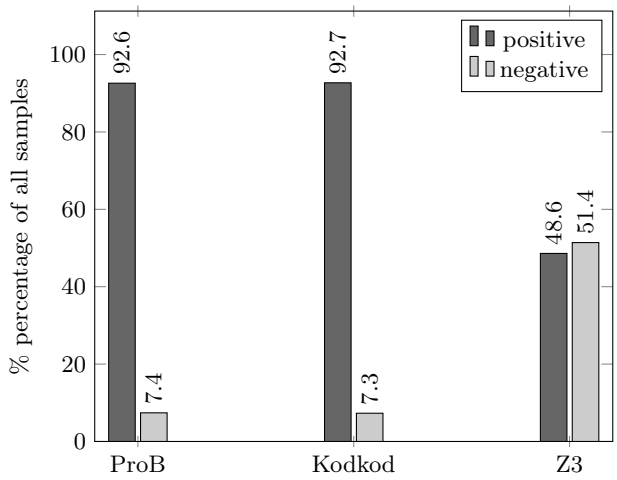


Fig. 3: Distribution of positive and negative samples throughout the training data.

Note, that the positive and negative samples may differ between two backends. For instance, the 92.6% of positive samples for ProB might not necessarily all be positive for Kodkod as well.

As a look at the intersatisfiability shows, there are indeed positive samples for each backend, that are negative samples for another one:

	ProB	Kodkod	Z3
ProB	-	99.80%	49.78%
Kodkod	99.60%	-	49.74%
Z3	94.76%	94.90%	-

Each row states the percentages of positive samples of the solver on the left, which also belong to the positive samples of the solver on the top. Omitted were the trivial 100% entries where a solver is compared to itself.

The class distribution is unequal, except of Z3 where it is roughly equal. To overcome this problem, Japkowicz et al. [22] proposed the method of *random under-sampling*, where samples from the training data are randomly deleted from the overrepresented classes, until all classes pose an equal distribution in

the data set. Following this approach, an individual training set was generated by random under-sampling for each of the three aforementioned classification problems. For the singular ProB classification the resulting training set consisted of 47,850 remaining constraints, 49.9% of which ProB could determine satisfiability for and with the satisfiability of the other 50.1% being unknown to ProB. The training data for the ProB+Kodkod classification still contained 68,201 constraints, where both backends returned the result unknown for 33.4% of the data, ProB determining satisfiability faster than KodKod for 33.4%, and for the remaining 33.2% Kodkod being faster. The data of the ProB+Z3 classification consisted of 47,248 samples split into the classes unknown, ProB, and Z3 to 33.4%, 33.2%, and 33.4% respectively.

7 Results

Before discussing the results of the training phase, a quick note about how the performance of the resulting neural networks was measured. As in classification each sample belongs to one class, the aim is to measure correctly and incorrectly predicted classes. For each class, a sample constraint x in the training data can be labelled with $l \in \{+, -\}$, where $+$ indicates the belonging of x to the positive class, and $-$ indicates the belonging to the negative class respectively. Another such labelling $z \in \{+, -\}$ can be given to the prediction $\hat{y}(x)$, indicating whether x was predicted to belong to the class in question or not. From said labels one can now build a confusion matrix [19] consisting of the number of *true positives* (TP), *true negatives* (TN), *false positives* (FP), and *false negatives* (FN) as entries:

	$z = +$	$z = -$
$l = +$	TP	FN
$l = -$	FP	TN

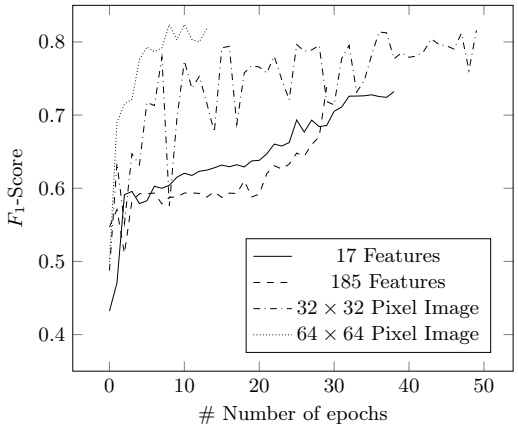
Using the confusion matrix, we can apply the common definitions of the performance measurements precision, recall, and F_1-score [19, 40]:

$$p = \frac{TP}{TP + FP} \qquad \text{(precision)}$$

$$r = \frac{TP}{TP + FN} \qquad \text{(recall)}$$

$$F_1 = \frac{2pr}{p + r}\,. \qquad (F_1\text{-score})$$

Precision represents the predictive value of a label [40] and high precision for a class indicates that predicting said class usually is correct. Recall represents effectiveness for a single class [40] and high recall indicates that most of the samples belonging to said class are predicted as such as well. Now, the F_1-score is defined as the harmonic mean of precision and recall. As we ultimately aim for a predictor that achieves both, a high precision and recall, we will use the F_1-score as measure of performance.



Features	F_1**-score**
17 Features	0.732
185 Features	0.738
32×32 Pixels	0.815
64×64 Pixels	0.823

Fig. 4: Best performing models for singular ProB classification. Learning curves over the validation set are shown on the left, the table on the right summarises the best performances achieved.

7.1 Results for Singular ProB Classification

For the singular ProB classification, the results appear to be promising. The best performing models' performances on the validation set are plotted against the respective training epochs in Figure 4.

In conclusion, the 185 B-features do not appear to be much more expressive as the initial 17 already were, as can be seen by both models achieving an almost identical performance on the validation set: **0.732** and **0.738** respectively. As in the experiments the number and size of the hidden layers was chosen randomly, the model for 17 happened to have more hidden layers of a larger size (namely 7 layers with 48 units each) than the one for 185 features (2 layers with 4 units each). As hidden layers can be interpreted as more abstract features themselves, the smaller architecture might suggest that a small and precise set of features is sufficient for training a well-performing predictor.

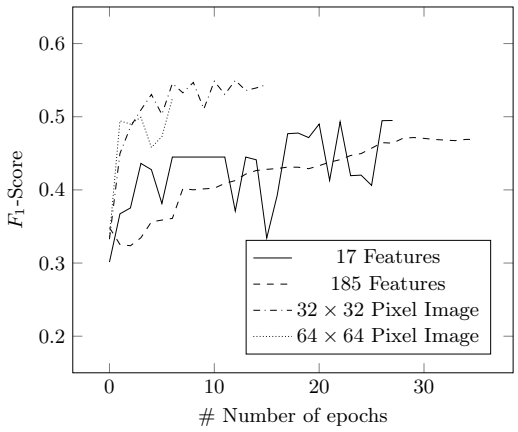
For us, the most notable surprise was the performance of the image based approach. With an F_1-score of **0.823**, the best performing CNN model topped those models for hand-crafted features by a notable margin. This result can be interpreted in two manners. Firstly, it appears that the structure of a constraint observable in the ASCII representation is sufficient for classification. In consequence, it may be that a hand-crafted feature set does not need to include features which are counting nodes in the constraint's syntax tree. Secondly, as the CNNs could not rely on any domain knowledge and still outperformed the models trained on specifically designed features, it may be that the crafted features themselves still lack the crucial characteristics needed to properly classify the constraints. On the other hand, it is possible that the CNN's advantage lies simply in the amount of parameters learned. The top CNN model learned 162,348

parameters, whereas the top FNN models only employed 14,736 parameters for 17 B-features, and 766 parameters for 185 B-features.

Overall, the reached F_1-score of 0.823 appears to be a promising result, indicating that this approach is indeed feasible. Verifying the performance on the test set still yielded an F_1-score of 0.819.

7.2 Results for ProB+Kodkod classification

For the ProB+Kodkod classification, there exist three distinct classes as mentioned in Section 4. Thus, the base performance of a respective classifier has to be greater than 0.333 to outperform uninformed guessing. The training performances on the corresponding validation set are shown in Figure 5.



Features	F_1**-score**
17 Features	0.495
185 Features	0.472
32×32 Pixels	0.550
64×64 Pixels	0.524

Fig. 5: Best performing models for the ProB+Kodkod classification. Learning curves over the validation set are shown on the left, the table on the right summarises the best performances achieved.

Top performances for the 17 and 185 feature sets were **0.495** and **0.472** respectively. Again, the CNN approach takes the first place with an F_1-score of **0.550** on the validation set. It even performed a bit better on the corresponding test set achieving an F_1-score of **0.562**. Table 1 shows the corresponding test set precision and recall performances for each class.

As can be seen, the average precision and recall are fairly close. On a per-class basis, the model seems to predict unknown instances right most of the time (precision of 0.768), whereas ProB appears to be a class that the model struggles to recognise in the first place (recall of 0.369), tending to generally predict in favour of Kodkod (high recall but moderate precision). While the Kodkod selection models performed better than guessing, having a model which is only correct in every other case appears not to be practical in the context of backend selection. Under the assumption that one backend will be chosen

eventually, the model does not perform notably better than picking a backend at random. The high precision on unknown constraints however may still help to detect problematic constraints beforehand.

Class	Precision	Recall
unknown	0.768	0.591
ProB	0.516	0.369
Kodkod	0.447	0.683
Average	0.577	0.548

Table 1: Precision and recall for the ProB+Kodkod classification on the test set.

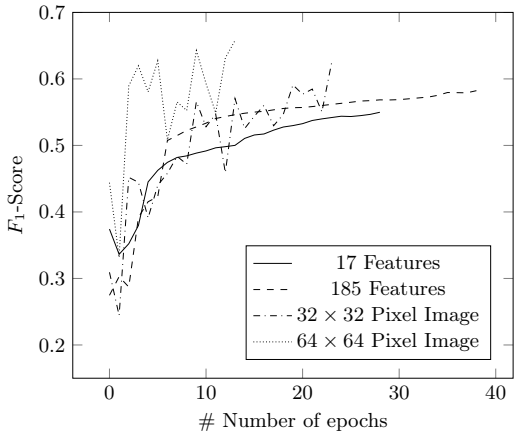
7.3 Results for ProB+Z3 classification

For the ProB+Z3 classification, the results outperformed the respective ones from the Kodkod experiments notably. Although tackling a comparable problem, the better performances might reside in the fact that ProB and Z3 excel on more divergent problem classes than ProB and Kodkod do. Again, the image based approach is the best, reaching an F_1-score of **0.658** on the validation set, and still one of **0.652** on the test set. Figure 6 summarises the learning curves and validation set performances of the top models for each feature set, whereas Table 2 displays the best model's precision and recall on the test set.

Like in the ProB+Kodkod classification, the precision and recall values for unknown instances are quite high compared to those of ProB or Z3. However, this time the model appears to predict in favour of ProB, contrary to the results in the ProB+Kodkod classification. While an F_1-score of 0.652 is not a satisfying rate of success, it notably outperforms uninformed guessing and, contrary to the respective Kodkod variant, could already improve the selection step. Further, a corresponding uninformed workflow could consist of defaulting to one backend, then switching to the second one for instances where the first one failed to provide a definite answer, rather than guessing. Considering the high precision of the Z3 class, a workflow that defaults to ProB could already be improved in overall-performance by using Z3 over ProB for instances where the model predicts to do so.

Class	Precision	Recall
unknown	0.845	0.668
ProB	0.490	0.789
Z3	0.714	0.416
Average	0.683	0.624

Table 2: Precision and recall for the ProB+Z3 classification on the test set.



Features	F_1**-score**
17 Features	0.550
185 Features	0.582
32×32 Pixels	0.624
64×64 Pixels	0.658

Fig. 6: Best performing models for the ProB+Z3 classification. Learning curves over the validation set are shown on the left, the table on the right summarises the best performances achieved.

8 Conclusion and Future Work

In summary, we conducted a broad random search over different deep learning models for the classification problems stated in Section 4. For this, we gathered a training set consisting of various constraints, and crafted two feature sets, one of 17 and one of 185 features respectively, which incorporated domain knowledge of the B language and method.

As an alternative approach to compare with, the constraints were translated into images to train convolutional neural networks, as outlined in Section 4.4. To our surprise, the image based approach outperformed the domain-specific features notably. These results might suggest that our hand-crafted features are still too domain-unspecific. This underlines the initial motivation to use machine learning instead of an expert system, as we apparently do not understand the problem domain well enough to precisely formulate meaningful characteristics as of why a certain backend might outperform another.

We assume that a concise and domain-specific set of features should be able to yield a better performance than the image based approach. Ideally, the learned correlations between constraints and suitable backends can be extracted from trained models. This could lead to a more sophisticated understanding of the problem domain which allows to formulate more precise feature sets. Thus, it might be sensible to change the machine learning algorithm to a more transparent one, Refining the feature sets with the help of decision trees [6, 35] and random forests [5] is subject to future work.

Be that as it may, one of the main take-aways is that approaches which require no domain knowledge can be preferred for initial performance probing and kickstarting results. Regarding deep learning, such techniques include an image based approach, as presented in this article, or a sequence based approach

with recurrent neural networks (RNNs). Comparing performances achieved by an RNN to those of our CNNs would be quite interesting.

The performances of the best models lead us to the belief, that our approach is feasible after all. Putting more work into fine-grained training of the top-performing models should lead to even better results allowing to assemble a portfolio of different backends from which the most suitable is selected automatically on a per-instance base.

The classification problems which were concerned with selecting between *unknown* and two backends tend to favour one of the backends over the other. This is fine, as in such a case the corresponding uninformed workflow would presumably consist of defaulting to one backend, then switching to the second one for instances where the first one failed to provide a definite answer. As already discussed for the ProB+Z3 classifier, the overall computation time could be decreased for this workflow by only partially following the predictions (e.g. for predictions with a high precision).

The performances for detecting instances for which both backends could not return any answer were consistently the highest in the respective classification problem, which falls also in line with the notably better performances achieved for the singular ProB classification. In fact, if we revisit the best performing model for ProB+Z3 classification and interpret it as a binary classifier between the classes *unknown* and *either* ProB *or Z3*, it achieves an F_1-score of 0.827 which is quite comparable to the singular ProB classifier.

Comparing the presented approach again with the regression approach of Healy [20, 21] it stands out that the latter is more extensible. Adding another backend would consist of adding a new regression model for said backend's runtime under Healy's approach, whereas in our approach we are not able to add Kodkod easily to the ProB+Z3 model, since we would have to train a new neural network instead which classifies between the three backends and the class of *unknown* samples. This is a huge drawback, rooted in the fact that the backends' runtimes are pairwise independent, but determining the fastest in the mix directly depends on the performances of all.

As the prediction of *unknown* samples per backend appeared to work well as stated above, implementing two models per backend might combine the best of both worlds. On the one hand, a runtime regression model per backend would easily allow for ranking the individual backends as in Healy's work. On the other hand, a singular classifier like the one presented for ProB in Section 7.1 can give an *independently computed* estimation over the backend's capability of finding an answer. For instance, given a ranking of ProB $\succ$ Z3 $\succ$ Kodkod with success probabilities of 62%, 51%, and 97% respectively, it might actually be more feasible to directly run Kodkod instead of risking two timeouts by running ProB and Z3 first. In fact the computed probability can be used for a weighted and more informed ranking.

References

1. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press (1996)
2. Back, R.J., Wright, J.: Refinement calculus: a systematic introduction. Springer Science & Business Media (2012)
3. Back, R.: On correct refinement of programs. Journal of Computer and System Sciences **23**(1), 49–68 (1981)
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wrocław, Poland (August 2011), https://hal.inria.fr/hal-00790310
5. Breiman, L.: Random forests. Machine learning **45**(1), 5–32 (2001)
6. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and regression trees (1984)
7. Bridge, J.P.: Machine learning and automated theorem proving. Tech. rep., University of Cambridge, Computer Laboratory (2010)
8. Cansell, D., Méry, D.: Foundations of the b method. Computing and informatics **22**(3-4), 221–256 (2012)
9. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Programming Languages: Implementations, Logics, and Programs. vol. 1292, pp. 191–206. Springer (1997)
10. Carlsson, M., Widen, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H., Sjöland, T.: SICStus Prolog user's manual, vol. 3. Swedish Institute of Computer Science Kista, Sweden (1988)
11. ClearSy: Atelier B, User and Reference Manuals. Aix-en-Provence, France (2016), available at http://www.atelierb.eu/
12. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. Tools and Algorithms for the Construction and Analysis of Systems pp. 337–340 (2008)
13. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)
14. Fraenkel, A.A., Bar-Hillel, Y., Levy, A.: Foundations of set theory, vol. 67. Elsevier (1973)
15. Fraenkel, A.: Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. Mathematische Annalen **86**(3), 230–237 (1922)
16. Goller, C.: Learning search-control heuristics for automated deduction systems with folding architecture networks. In: ESANN. pp. 45–50 (1999)
17. Goller, C., Kuchler, A.: Learning task-dependent distributed representations by backpropagation through structure. In: Proceedings of International Conference on Neural Networks (ICNN'96). vol. 1, pp. 347–352. IEEE (1996)
18. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), http://www.deeplearningbook.org
19. Goutte, C., Gaussier, E.: A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In: ECIR. vol. 5, pp. 345–359. Springer (2005)
20. Healy, A.: Predicting smt solver performance for software verification (2016), http://eprints.maynoothuniversity.ie/8770/

21. Healy, A., Monahan, R., Power, J.F.: Evaluating the use of a general-purpose benchmark suite for domain-specific smt-solving. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. pp. 1558–1561. SAC '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2851613.2851975, http://doi.acm.org/10.1145/2851613.2851975
22. Japkowicz, N.: The class imbalance problem: Significance and strategies. In: Proc. of the Int'l Conf. on Artificial Intelligence (2000)
23. Krings, S., Bendisposto, J., Leuschel, M.: From failure to proof: the prob disprover for b and event-b. In: Software Engineering and Formal Methods, pp. 199–214. Springer (2015)
24. Krings, S., Leuschel, M.: Smt solvers for validation of b and event-b models. In: International Conference on Integrated Formal Methods. pp. 361–375. Springer (2016)
25. Krings, S., Schmidt, J., Brings, C., Frappier, M., Leuschel, M.: A translation from alloy to b. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 71–86. Springer International Publishing, Cham (2018)
26. Kubat, M., Matwin, S., et al.: Addressing the curse of imbalanced training sets: one-sided selection. In: ICML. vol. 97, pp. 179–186. Nashville, USA (1997)
27. LeCun, Y., Kavukcuoglu, K., Farabet, C.: Convolutional networks and applications in vision. In: Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on. pp. 253–256. IEEE (2010)
28. Letz, R., Schumann, J., Bayerl, S., Bibel, W.: Setheo: A high-performance theorem prover. Journal of Automated Reasoning **8**(2), 183–212 (1992)
29. Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D.: From Animation to Data Validation: The ProB Constraint Solver 10 Years On. In: Boulanger, J.L. (ed.) Formal Methods Applied to Complex Systems: Implementation of the B Method, chap. 14, pp. 427–446. Wiley ISTE, Hoboken, NJ (2014)
30. Leuschel, M., Butler, M.: Prob: an automated analysis toolset for the b method. International Journal on Software Tools for Technology Transfer **10**(2), 185–203 (2008)
31. Leuschel, M., Butler, M., et al.: Prob: A model checker for b. In: FME. vol. 2805, pp. 855–874. Springer (2003)
32. Lewis, D.D., Catlett, J.: Heterogeneous uncertainty sampling for supervised learning. In: Proceedings of the eleventh international conference on machine learning. pp. 148–156 (1994)
33. Loreggia, A., Malitsky, Y., Samulowitz, H., Saraswat, V.A.: Deep learning for algorithm portfolios. In: AAAI. pp. 1280–1286 (2016)
34. Plagge, D., Leuschel, M.: Validating b, z and tla+ using prob and kodkod. In: International Symposium on Formal Methods. pp. 372–386. Springer (2012)
35. Quinlan, J.R.: Induction of decision trees. Machine learning **1**(1), 81–106 (1986)
36. Rosenblatt, F.: The perceptron: A probabilistic model for information storage and organization in the brain. Psychological review **65**(6), 386 (1958)
37. Rosenblatt, F.: Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Tech. rep., CORNELL AERONAUTICAL LAB INC BUFFALO NY (1961)
38. Rumelhart, D., Hinton, G., Williams, R.: Learning representations by back-propagating errors. Nature **323**(6088), 533–538 (1986)
39. Schulz, S.: E–a brainiac theorem prover. Ai Communications **15**(2, 3), 111–126 (2002)

40. Sokolova, M., Japkowicz, N., Szpakowicz, S.: Beyond accuracy, f-score and roc: a family of discriminant measures for performance evaluation. In: Australian conference on artificial intelligence. vol. 4304, pp. 1015–1021 (2006)
41. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 632–647. Springer (2007)
42. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE transactions on evolutionary computation **1**(1), 67–82 (1997)
43. Wolpert, D.H., Macready, W.G., et al.: No free lunch theorems for search. Tech. rep., Technical Report SFI-TR-95-02-010, Santa Fe Institute (1995)