ENIGMA-NG: Efficient Neural and Gradient-Boosted Inference Guidance for E

Karel Chvalovský(✉), Jan Jakubův, Martin Suda, and Josef Urban

Czech Technical University in Prague, Prague, Czech Republic
karel@chvalovsky.cz

Abstract. We describe an efficient implementation of given clause selection in saturation-based automated theorem provers, extending the previous ENIGMA approach. Unlike in the first ENIGMA implementation where a fast linear classifier is trained and used together with manually engineered features, we have started to experiment with more sophisticated state-of-the-art machine learning methods such as gradient boosted trees and recursive neural networks. In particular, the latter approach poses challenges in terms of efficiency of clause evaluation, however, we show that deep integration of the neural evaluation with the ATP data-structures can largely amortize this cost and lead to competitive real-time results. Both methods are evaluated on a large dataset of theorem proving problems and compared with the previous approaches. The resulting methods improve on the manually designed clause guidance, providing the first practically convincing application of gradient-boosted and neural clause guidance in saturation-style automated theorem provers.

1 Introduction

Automated theorem provers (ATPs) have been developed for decades by manually designing proof calculi and search heuristics. Their power has been growing and they are already very useful, e.g., as parts of large interactive theorem proving (ITP) verification toolchains (hammers) [5]. On the other hand, with small exceptions, ATPs are still significantly weaker than trained mathematicians in finding proofs in most research domains.

Recently, machine learning over large formal corpora created from ITP libraries [24,32,42] has started to be used to develop guidance of ATP systems [2,30,44]. This has already produced strong systems for selecting relevant facts for proving new conjectures over large formal libraries [1,4,13]. More recently, machine learning has also started to be used to guide the internal search of the ATP systems. In sophisticated saturation-style provers this has been done

Supported by the ERC Consolidator grant no. 649043 AI4REASON, and by the Czech project AI&Reasoning CZ.02.1.01/0.0/0.0/15_003/0000466 and the European Regional Development Fund.

by feedback loops for strategy invention [21,38,43] and by using supervised learning [19,31] to select the next given clause [35]. In the simpler connection tableau systems such as LeanCoP [34], supervised learning has been used to choose the next tableau extension step [25,45] and first experiments with Monte-Carlo guided proof search [12] and reinforcement learning [26] have been done.[1]

In this work, we add two state-of-the-art machine learning methods to the ENIGMA [19,20] algorithm that efficiently guides saturation-style proof search. The first one trains gradient boosted trees on efficiently extracted manually designed (handcrafted) clause features. The second method removes the need for manually designed features, and instead uses end-to-end training of recursive neural networks. Such architectures, when implemented naively, are typically expensive and may be impractical for saturation-style ATP. We show that deep integration of the neural evaluation with the ATP data-structures can largely amortize this cost, allowing competitive performance.

The rest of the paper is structured as follows. Section 2 introduces saturation-based automated theorem proving with the emphasis on machine learning. Section 3 briefly summarizes our previous work with handcrafted features in ENIGMA and then extends the previously published ENIGMA with additional classifiers based on decision trees (Sect. 3.3) and simple feature hashing (Sect. 3.4). Section 4 presents our new approach of applying neural networks for ATP guidance. Section 5 provides experimental evaluation of our work. We conclude in Sect. 6.

2 Automated Theorem Proving with Machine Learning

State-of-the-art saturation-based automated theorem provers (ATPs) for first-order logic (FOL), such as E [40] and Vampire [28] are today's most advanced tools for general reasoning across a variety of mathematical and scientific domains. Many ATPs employ the *given clause algorithm*, translating the input FOL problem $T \cup \{\neg C\}$ into a refutationally equivalent set of clauses. The search for a contradiction is performed maintaining sets of *processed* (P) and *unprocessed* (U) clauses. The algorithm repeatedly selects a *given clause* g from U, moves g to P, and extends U with all clauses inferred with g and P. This process continues until a contradiction is found, U becomes empty, or a resource limit is reached. The size of the unprocessed set U grows quickly and it is a well-known fact that the selection of the right given clause is crucial for success. Machine learning from a large number of proofs and proof searches may help guide the selection of the given clauses.

E allows the user to select a *proof search strategy* $\mathcal{S}$ to guide the proof search. An E strategy $\mathcal{S}$ specifies parameters such as term ordering, literal selection function, clause splitting, paramodulation setting, premise selection, and, most importantly for us, the *given clause selection* mechanism. The given clause selection in E is implemented using a list of priority queues. Each priority queue stores

[1] Other, less immediately relevant, previous work on combining machine learning with automated theorem proving includes, e.g., [6,8,9,11,39].

all the generated clauses in a specific order determined by a clause *weight function*. The clause weight function assigns a numeric (real) value to each clause, and the clauses with smaller weights ("lighter clauses") are prioritized. To select a given clause, one of the queues is chosen in a round robin manner, and the clause at the front of the chosen queue gets processed. Each queue is additionally assigned a *frequency* which amounts to the relative number of clause selections from that particular queue. Frequencies can be used to prefer one queue over another. We use the following notation to denote the list of priority queues with frequencies f_i and weight functions $\mathcal{W}_i$:

$$(f_1 * \mathcal{W}_1, \ldots, f_k * \mathcal{W}_k).$$

To facilitate machine learning research, E implements an option under which each successful proof search gets analyzed and outputs a list of clauses annotated as either *positive* or *negative* training examples. Each processed clause that is present in the final proof is classified as positive. On the other hand, processing of clauses not present in the final proof was redundant, hence they are classified as negative. Our goal is to learn such classification (possibly conditioned on the problem and its features) in a way that generalizes and leads to solving previously unsolved related problems.

Given a set of problems $\mathcal{P}$, we can run E with a strategy $\mathcal{S}$ and obtain positive and negative training data $\mathcal{T}$ from each of the successful proof searches. In this work, we use three different machine learning methods to learn the clause classification given by $\mathcal{T}$, each method yielding a *classifier* or *model* $\mathcal{M}$. The concrete structure of $\mathcal{M}$ depends on the machine learning method used, as explained in detailed below. With any method, $\mathcal{M}$ provides a function to compute the weight of an arbitrary clause. This weight function is then used in E to guide further proof runs.

A model $\mathcal{M}$ can be used in E in different ways. We use two methods to combine $\mathcal{M}$ with a strategy $\mathcal{S}$. Either (1) we use $\mathcal{M}$ to select *all* the given clauses, or (2) we combine $\mathcal{M}$ with the given clause guidance from $\mathcal{S}$ so that roughly half of the clauses are selected by $\mathcal{M}$. We denote the resulting E strategies as (1) $\mathcal{S} \odot \mathcal{M}$, and (2) $\mathcal{S} \oplus \mathcal{M}$. The two strategies are equal up to the priority queues for given clause selection which are changed ($\rightsquigarrow$) as follows.

$$\begin{aligned}
&\text{in } \mathcal{S} \odot \mathcal{M}: \ (f_1 * \mathcal{W}_1, \ldots, f_k * \mathcal{W}_k) \rightsquigarrow (1 * \mathcal{M}),\\
&\text{in } \mathcal{S} \oplus \mathcal{M}: \ (f_1 * \mathcal{W}_1, \ldots, f_k * \mathcal{W}_k) \rightsquigarrow ((\textstyle\sum f_i) * \mathcal{M}, f_1 * \mathcal{W}_1, \ldots, f_k * \mathcal{W}_k).
\end{aligned}$$

The strategy $\mathcal{S} \oplus \mathcal{M}$ usually performs better in practice as it helps to counter overfitting by combining powers with the original strategy $\mathcal{S}$. The strategy $\mathcal{S} \odot \mathcal{M}$ usually provides additional proved problems, gaining additional training data, and it is useful for the evaluation of the training phase. When $\mathcal{S} \odot \mathcal{M}$ performs better than $\mathcal{S}$, it indicates that $\mathcal{M}$ has learned the training data well. When it performs much worse, it indicates that $\mathcal{M}$ is not very well-trained. The strategy $\mathcal{S} \oplus \mathcal{M}$ should always perform better than $\mathcal{S}$, otherwise the guidance of $\mathcal{M}$ is not useful. Additional indication of successful training can be obtained from the number of clauses processed during a successful proof search. The strategy $\mathcal{S} \odot \mathcal{M}$

should run with much fewer processed clauses, in some cases even better than $\mathcal{S} \oplus \mathcal{M}$, as the original $\mathcal{S}$ might divert the proof search. In the best case, when $\mathcal{M}$ would learn to guide for certain problem perfectly, the number of processed clauses would not need to exceed the length of the proof.

It is important to combine a model $\mathcal{M}$ only with a "compatible" strategy $\mathcal{S}$. For example, let us consider a model $\mathcal{M}$ trained on samples obtained with another strategy S_0 which has a different term ordering than $\mathcal{S}$. As the term ordering can change term normal forms, the clauses encountered in the proof search with $\mathcal{S}$ might look quite different from the training clauses. This may be an issue unless the trained models are independent of symbol names, which is not (yet) our case. Additional complications might arise as term orderings and literal selection might change the proof space and the original proofs might not be reachable. Hence we only combine $\mathcal{M}$ with the strategy $\mathcal{S}$ which provided the examples on which $\mathcal{M}$ was trained.

3 ATP Guidance with Handcrafted Clause Features

In order to employ a machine learning method for ATP guidance, first-order clauses need to be represented in a format recognized by the selected learning method. A common approach is to manually extract a finite set of various properties of clauses called *features*, and to encode these clause features by a fixed-length numeric vector. Various machine learning methods can handle numeric vectors and their success heavily depends on the selection of correct clause features. In this section, we work with handcrafted clause features which, we believe, capture information important for ATP guidance.

ENIGMA [19,20] is our *efficient* learning-based method for guiding given clause selection in saturation-based ATPs. Sections 3.1 and 3.2 briefly summarizes our previous work. Sections 3.3 and 3.4 describe extensions, first presented in this work.

3.1 ENIGMA Clause Features

So far the development of ENIGMA has focused on fast and practically usable methods, allowing E users to directly benefit from our work. Various possible choices of efficient clause features for theorem prover guidance have been experimented with [19,20,26,27]. The original ENIGMA [19] uses term-tree walks of length 3 as features, while the second version [20] reaches better results by employing various additional features. In particular, the following types of features are used (see [19, Sec. 3.2] and [20, Sec. 2] for details):

Vertical Features are (top-down-)oriented term-tree walks of length 3. For example, the unit clause $P(f(a,b))$ contains only features (P,f,a) and (P,f,b).

Horizontal Features are horizontal cuts of a term tree. For every term $f(t_1,\ldots,t_n)$ in the clause, we introduce the feature $f(s_1,\ldots,s_n)$ where s_i is the top-level symbol of t_i.

Symbol Features are various statistics about clause symbols, namely, the number of occurrences and the maximal depth for each symbol.

Length Features count the clause length and the numbers of positive and negative literals.

Conjecture Features embed information about the conjecture being proved into the feature vector. In this way, ENIGMA can provide conjecture-dependent predictions.

Since there are only finitely many features in any training data, the features can be serially numbered. This numbering is fixed for each experiment. Let n be the number of different features appearing in the training data. A clause C is translated to a feature vector φ_C whose i-th member counts the number of occurrences of the i-th feature in C. Hence every clause is represented by a sparse numeric vector of length n.

With conjecture features, instead of using the vector φ_C of length n, we use a vector (φ_C, φ_G) of length $2n$ where φ_G contains the features of the conjecture G. For a training clause C, G corresponds to the conjecture of the proof search where C was selected as a given clause. When classifying a clause C during proof search, G corresponds to the conjecture currently being proved. When the conjecture consists of several clauses, their vectors are computed separately and then summed (except for features corresponding to maxima, such as the maximal symbol depth, where maximum is taken instead).

3.2 ATP Guidance with Fast Linear Classifiers

ENIGMA has so far used simple but fast linear classifiers such as *linear SVM* and *logistic regression* efficiently implemented by the LIBLINEAR open source library [10]. In order to employ them, clause representation by numeric feature vectors described above in Sect. 3.1 is used. Clausal training data $\mathcal{T}$ are translated to a set of fixed-size labeled vectors. Each (typically sparse) vector of length n is labeled either as positive or negative.

The labeled numeric vectors serve as an input to LIBLINEAR which, after the training, outputs a model $\mathcal{M}$ consisting mainly of a weight vector w of length n. The main cost in classifying a clause C consists in computing its feature vector φ_C and its dot product with the weight vector $p = \varphi_C \cdot w$. ENIGMA the assigns to the positively classified clauses (i.e., $p \geq 0$) a chosen small weight (1.0) and a higher weight (10.0) to the negatively classified ones (i.e., $p < 0$). This weight is then used inside E to guide given clause selection as described in Sect. 2.

The training data obtained from the proof runs are typically not balanced with respect to the number of positive and negative examples. Usually, there are many more negative examples and the method of *Accuracy-Balancing Boosting* [20] was found useful in practice to improve precision on the positive training data. This is done as follows. Given training data $\mathcal{T}$ we create a LIBLINEAR classifier $\mathcal{M}$, test $\mathcal{M}$ on the training data, and collect the positives mis-classified by $\mathcal{M}$. We then repeat (*boost*) the mis-classified positives in the training data, yielding updated $\mathcal{T}_1$ and an updated classifier $\mathcal{M}_1$. We iterate this process, and with every iteration, the accuracy on the positive samples increases, while the accuracy on the negatives typically decreases. We finish the boosting when the positive accuracy exceeds the negative one. See [20, Sec. 2] for details.

3.3 ATP Guidance with Gradient Boosted Trees

Fast linear classifiers together with well-designed features have been used with good results for a number of tasks in areas such as NLP [23]. However, more advanced learning models have been recently developed, showing improved performance on a number of tasks, while maintaining efficiency. One such method is *gradient boosted trees* and, in particular, their implementation in the XGBoost library [7]. Gradient boosted trees are ensembles of decision trees trained by tree boosting.

The format of the training and evaluation data used by XGBoost is the same as the input used by LIBLINEAR (sparse feature vectors). Hence, we use practically the same approach for obtaining the positive and negative training examples, extracting their features, and clause evaluation during proof runs as described in Sects. 3.1 and 3.2. XGBoost, however, does not require the accuracy-balancing boosting. This is because XGBoost can deal with unbalanced training data by setting the ratio of positive and negative examples.[2]

The model $\mathcal{M}$ produced by XGBoost consists of a set (*ensemble* [37]) of decision trees. The inner nodes of the decision trees consist of conditions on feature values, while the leafs contain numeric scores. Given a vector φ_C representing a clause C, each tree in $\mathcal{M}$ is navigated to a unique leaf using the values from φ_C, and the corresponding leaf scores are aggregated across all trees. The final score is translated to yield the probability that φ_C represents a positive clause. When using $\mathcal{M}$ as a weight function in E, the probabilities are turned into binary classification, assigning weight 1.0 for probabilities ≥ 0.5 and weight 10.0 otherwise. Our experiments with scaling of the weight by the probability did not yet yield improved functionality.

3.4 Feature Hashing

In the previous version of ENIGMA, the vectors representing clauses had always length n where n is the total number of features in the training data $\mathcal{T}$ (or $2n$ with conjecture features). Experiments revealed that both LIBLINEAR and XGBoost are capable of dealing with vectors up to the length of 10^5 with a reasonable performance. This might be enough for smaller benchmarks, but with the need to train on bigger training data, we might need to handle much larger feature sets. In experiments with the whole translated Mizar Mathematical Library [42], the feature vector length can easily grow over 10^6. This significantly increases both the training and the clause evaluation times. To handle such larger data sets, we have implemented a simple *hashing* method to decrease the dimension of the vectors.

[2] We use the XGBoost parameter `scale_pos_weight`.

Instead of serially numbering all features, we represent each feature f by a unique string and apply a general-purpose string hashing function to obtain a number n_f within a required range (between 0 and an adjustable *hash base*). The value of f is then stored in the feature vector at the position n_f. If different features get mapped to the same vector index, the corresponding values are summed up.

We use the following hashing function *sdbm* coming from the open source SDBM project. Given a string s, the value h_i is computed for every character as follows:

$$h_i = s_i + (h_{i-1} \ll 6) + (h_{i-1} \ll 16) - h_{i-1}$$

where $h_0 = 0$, s_i is the ASCII code of the character at the i-th position, and the operation $\ll$ stands for a bit shift. The value for the last character is computed with a fixed-size data type (we use 64-bit unsigned integers) and this value modulo the selected hash base is returned. We evaluate the effect of the selected hashing function later in Sect. 5.

4 Neural Architecture for ATP Guidance

Although the handcrafted clause features described in Sect. 3.1 lead to very good results, they naturally have several limitations. It is never clear whether the selected set of features is the best available given the training data. Moreover, a rich set of features can easily lead to long sparse vectors and thus using them for large corpora requires the use of dimensionality reduction techniques (c.f. Sect. 3.4). Hence selecting the features automatically is a natural further step.

Among various techniques used to extract features fully automatically, neural networks (NN) have recently become the most popular thanks to many successful applications in, e.g., computer vision and natural language processing. There have been several attempts to use NNs for guiding ATPs. However, such attempts have so far typically suffered from a large overhead needed to evaluate the used NN [31], making them impractical for actual proving.

A popular approach for representing tree-structured data, like logical formulae, is based on recursive NNs [16]. The basic idea is that all objects (tree nodes, subterms, subformulas) are represented in a high dimensional vector space and these representations are subject to learning. Moreover, the representation of more complex objects is a function of representations of their arguments. Hence constants, variables, and atomic predicates are represented as directly learned vectors, called *vector embeddings*. Assume that all such objects are represented by n-dimensional vectors. For example, constants a and b are represented by learned vectors v_a and v_b, respectively. The representation of $f(a,b)$ is then produced by a learned function (NN), say v_f, that has as an input two vectors and returns a vector; hence $v_f(v_a, v_b) \in \mathbf{R}^n$. Moreover, the representation of $P(f(a,b), a)$ is obtained similarly, because from our point of view a representation is just a function of arguments. Therefore we have

$$v_f \colon \underbrace{\mathbf{R}^n \times \cdots \times \mathbf{R}^n}_{k\text{-times}} \to \mathbf{R}^n \qquad \text{for every } k\text{-ary, } k \geq 0\text{, function symbol } f,$$
$$v_P \colon \underbrace{\mathbf{R}^n \times \cdots \times \mathbf{R}^n}_{k\text{-times}} \to \mathbf{R}^n \qquad \text{for every } k\text{-ary, } k \geq 0\text{, predicate symbol } P,$$

in our language. We treat all variables as a single symbol, i.e., we always represent a variable by a fixed learned vector, and similarly for Skolem names. This is in line with how the ENIGMA features are constructed and thus allows for a more straightforward comparison. We also replace symbols that appear rarely (fewer than 10 times) in our training set by a representative, e.g., all rare binary functions become the same binary function. Loosely speaking, we learn a general binary function this way. Because we treat equality and negation as learned functions, we have described how a representation of a literal is produced.

We could now produce the representation of clauses by assuming that disjunction is a binary connective, however, we instead use a more direct approach and we treat clauses directly as sequences of literals. Recurrent neural networks (RNN) are commonly used to process arbitrary sequences of vectors. Hence we train an RNN, called *Cl*, that consumes the representations of literals in a clause and produces the representation of the clause, $\mathit{Cl} \colon \mathbf{R}^n \times \cdots \times \mathbf{R}^n \to \mathbf{R}^n$.

Given a representation of a clause we could learn a function that says whether the clause is a good given clause. However, without any context this may be hard to decide. As in ENIGMA and [31], we introduce more context into our setting by using the problem's conjecture. The negated conjecture is translated by E into a set of clauses. We combine the vector representations of these clauses by another RNN, called *Conj* and defined by $\mathit{Conj} \colon \mathbf{R}^n \times \cdots \times \mathbf{R}^n \to \mathbf{R}^n$.

Now that we know how to represent a conjecture and a given clause by a vector, we can define a function that combines them into a decision, called *Fin* and defined by $\mathit{Fin} \colon \mathbf{R}^n \times \mathbf{R}^n \to \mathbf{R}^2$. The two real values can later be turned into probabilities of whether the clause will (will not) be useful, see Sect. 4.2.

Although all the representations have been vectors in $\mathbf{R}^n$, this is an unnecessary restriction. It suffices if the objects of the same type are represented by vectors of the same length. For example, we have experimented with *Conj* where outputs are shorter (and inputs to *Fin* are changed accordingly) with the aim to decrease overfitting to a particular problem.

4.1 Neural Model Parameters

The above mentioned neural model can be implemented in many ways. Although we have not performed an extensive grid search over various variants, we can discuss some of them shortly. The basic parameter is the dimension n of the vectors. We have tried various models with $n \in \{8, 16, 32, 64, 128\}$. The functions used for v_f and v_P can be simple linear transformations (tensors), or more complex combinations of linear and nonlinear layers. An example of a frequently used nonlinearity is the rectified linear unit (ReLU), defined by $\max(0, x)$.[3]

[3] Due to various numerical problems with deep recursive networks we have obtained better results with ReLU6, defined by $\min(\max(0, x), 6)$, or tanh.

For *Cl* and *Conj* we use (multi-layer) long short-term memory (LSTM) RNNs [18]. We have tried to restrict the output vector of *Conj* to $m = \frac{n}{2}$ or $m = \frac{n}{4}$ to prevent overfitting with inconclusive results. The *Fin* component is a sequence of alternating linear and nonlinear layers (ReLU), where the last two linear layers are $\mathbf{R}^{n+m} \rightarrow \mathbf{R}^{\frac{n}{2}}$ and $\mathbf{R}^{\frac{n}{2}} \rightarrow \mathbf{R}^2$.

4.2 ATP Guidance with Pytorch

We have created our neural model using the Pytorch library and integrated it with E using the library's C++ API.[4] This API allows to load a previously trained model saved to a file in a special TorchScript format. We use a separate file for each of the neural parts described above. This includes computing of the vector embeddings of terms, literals, and clauses, as well as the conjecture embedding *Conj* summarizing the conjecture clauses into one vector, and finally the part *Fin*, which classifies clauses into those deemed useful for proving the given conjecture and the rest.

We have created a new clause weight function in E called TorchEval which interfaces these parts and can be used for evaluating clauses based on the neural model. One of the key features of the interface, which is important for ensuring reasonable evaluation speed, is *caching* of the embeddings of terms and literals. Whenever the evaluation encounters a term or a literal which was evaluated before, its embedding is simply retrieved from the memory in constant time instead of being computed from the embeddings of its subterms recursively. We use the fact that terms in E are perfectly shared and thus a pointer to a particular term can be used as a key for retrieving the corresponding embedding. Note that this pervasive caching is possible thanks to our choice of recursive neural networks (that match our symbolic data) and it would not work with naive use of other neural models such as convolutional or recurrent networks without further modifications.

The clause evaluation part of the model returns two real outputs x_0 and x_1, which can be turned into a probability that the given clause will be useful using the sigmoid (logistic) function:

$$p = \frac{1}{1 + e^{(x_0 - x_1)}}. \tag{1}$$

However, for classification, i.e. for a yes-no answer, we can just compare the two numbers and "say yes" whenever

$$x_0 < x_1. \tag{2}$$

After experimenting with other schemes that did not perform so well,[5] we made TorchEval return 1.0 whenever condition (2) is satisfied and 10.0 otherwise.

[4] https://pytorch.org/cppdocs/.

[5] For instance, using the probability (1) for a more fine-grained order on clauses dictated by the neural model.

This is in accord with the standard convention employed by E that clauses with smaller weight should be preferred and also corresponds to the ENIGMA approach. Moreover, E implicitly uses an ever-increasing clause id as a tie breaker, so among the clauses within the same class, both TorchEval and ENIGMA behave as FIFO.

Another performance improvement was obtained by forcing Pytorch to use just a single core when evaluating the model in E. The default Pytorch setting was causing degradation of performance on machines with many cores, probably by assuming by default that multi-threading will speed up frequent numeric operations such as matrix multiplication. It seems that in our case, the overhead for multi-threading at this point might be higher than the gain.

5 Experimental Evaluation

We experimentally evaluated the three learning-based ATP guidance methods on the MPTP2078 benchmark [1].[6] MPTP2078 contains 2078 problems coming from the MPTP translation [42] of the Mizar Mathematical Library (MML) [3] to FOL. The consistent use of symbol names across the MPTP corpus is crucial for our symbol-based learning methods. We evaluated ATP performance with a good-performing baseline E strategy, denoted $\mathcal{S}$, which was previously optimized [22] on Mizar problems (see Appendix A for details).

Section 5.1 provides details on model training and the hyperparameters used, and analyzes the most important features used by the tree model. The model based on linear regression (Sect. 3.2) is denoted $\mathcal{M}_{\text{lin}}$, the model based on decision trees (Sect. 3.3) is denoted $\mathcal{M}_{\text{tree}}$, and the neural model (Sect. 4) is denoted $\mathcal{M}_{\text{nn}}$. Sections 5.2 and 5.3 evaluate the performance of the models both by standard machine learning metrics and by plugging them into the ATPs. Section 5.4 evaluates the effect of the feature hashing described in Sect. 3.4.

All experiments were run on a server with 36 hyperthreading Intel(R) Xeon(R) Gold 6140 CPU @ 2.30 GHz cores, with 755 GB of memory available in total. Each problem is always assigned one core. For training of the neural models we used NVIDIA GeForce GTX 1080 Ti GPUs. As described above, neither GPU nor multi-threading was, however, employed when using the trained models for clause evaluation inside the ATP.

5.1 Model Training, Hyperparameters and Feature Analysis

We evaluated the baseline strategy $\mathcal{S}$ on all the 2078 benchmark problems with a fixed CPU time limit of 10 s per problem.[7] This yielded 1086 solved problems and provided the training data for the learning methods as described in Sect. 2. For $\mathcal{M}_{\text{lin}}$ and $\mathcal{M}_{\text{tree}}$, the training data was translated to feature vectors (see

[6] The benchmark can be found at https://github.com/JUrban/MPTP2078. For all the remaining materials for reproducing the experiments please check out the repository https://github.com/ai4reason/eprover-data/tree/master/CADE-19.

[7] This appears to be a reasonable waiting time for, e.g., the users of ITP hammers [5].

Sect. 3) which were then fed to the learner. For $\mathcal{M}_{\text{nn}}$ we used the training data directly without any feature extraction.

Training Data and Training of Linear and Tree Models: The training data consisted of around 222 000 training samples (21 000 positives and 201 000 negatives) with almost 32 000 different ENIGMA features. This means that the training vectors for $\mathcal{M}_{\text{lin}}$ and $\mathcal{M}_{\text{tree}}$ had dimension close to 64 000,[8] and so had the output weight vector of $\mathcal{M}_{\text{lin}}$. For $\mathcal{M}_{\text{tree}}$, we reused the parameters that performed well in the ATPBoost [36] and rlCoP [26] systems and produced models with 200 decision trees, each with maximal depth 9. The resulting models—both linear and boosted trees—were about 1MB large in their native representation. The training time for $\mathcal{M}_{\text{lin}}$ was around 8 min (five iterations of accuracy-balancing boosting), and approximately 5 min for $\mathcal{M}_{\text{tree}}$. Both of them were measured on a single CPU core. During the boosting of $\mathcal{M}_{\text{lin}}$, the positive samples were extended from 21k to 110k by repeating the mis-classified vectors.

Learned Tree Features: The boosted tree model $\mathcal{M}_{\text{tree}}$ allows computing statistics of the most frequently used features. This is an interesting aspect that goes in the direction of *explainable* AI. The most important features can be analyzed by ATP developers and compared with the ideas used in standard clause evaluation heuristics. There were 200 trees in $\mathcal{M}_{\text{tree}}$ with 20215 decision nodes in total. These decision nodes refer to only 3198 features out of the total 32000. The most frequently used feature was the clause length, used 3051 times, followed by the conjecture length, used 893 times, and by the numbers of the positive and negative literals in the clauses and conjectures. In a crude way, the machine learning here seems to confirm the importance assigned to these basic metrics by ATP researchers. The set of top ten features additionally contains three symbol counts (including "$\in$" and "$\subseteq$") and a vertical feature corresponding to a variable occurring under negated set membership $\in$ (like in "$x \notin \cdot$" or "$\cdot \notin x$"). This seems plausible, since the Mizar library and thus MPTP2078 are based on set theory where membership and inclusion are key concepts.

Neural Training and Final Neural Parameters: We tried to improve the training of $\mathcal{M}_{\text{nn}}$ by randomly changing the order of clauses in conjectures, literals in clauses, and terms in equalities. If after these transformations a negative example pair (C, G) was equivalent to a positive one, we removed the negative one from the training set. This way we reduced the number of negative examples to 198k. We trained our model in batches[9] of size 128 and used the negative log-likelihood as a loss function (the learning rate is 10^{-3}), where we applied log-softmax on the output of *Fin*. We weighted positive examples more to simulate a balanced training set. All symbols of the same type and arity that have less than 10 occurrences in the training set were represented by one symbol. We set the vector dimension to be $n = 64$ for the neural model $\mathcal{M}_{\text{nn}}$ and we set the output of *Conj* to be $m = 16$. All the functions representing function symbols

[8] Combining the dimensions for the clause and the conjecture.

[9] Moreover, we always try to put examples with the same conjecture G into the same batch to share the time for recomputing the representation of G.

Table 1. True Positive Rate (TPR) and True Negative Rate (TNR) on training data.

	$\mathcal{M}_{\text{lin}}$	$\mathcal{M}_{\text{tree}}$	$\mathcal{M}_{\text{nn}}$
TPR	90.54%	99.36%	97.82%
TNR	83.52%	93.32%	94.69%

and predicates were composed of a linear layer and ReLU6. *Fin* was set to be a sequence of linear, ReLU, linear, ReLU, and linear layers. The training time for $\mathcal{M}_{\text{nn}}$ was around 8 min per epoch and the model was trained for 50 epochs. Note that we save a model after each epoch and randomly test few of these generated models and select the best performing one (in our case the model generated after 35 epochs).

5.2 Evaluation of the Model Performance

Training Performance of the Models: We first evaluate how well the individual models managed to learn the training data. Due to possible overfitting, this is obviously used only as a heuristic and the main metric is provided by the ultimate ATP evaluation. Table 1 shows for each model the *true positive* and *true negative* rates (TPR, TNR) on the training data, that is, the percentage of the positive and negative examples, classified correctly by each model.[10] The highest TPR, also called sensitivity, is achieved by $\mathcal{M}_{\text{tree}}$ while the highest TNR, also called specificity, by $\mathcal{M}_{\text{nn}}$. As expected, the accuracy of the linear model is lower. Its main strength seems to come from the relatively high speed of evaluation (see below).

ATP Performance of the Models: Table 2 shows the total number of problems solved by the four methods. For each learning-based model $\mathcal{M}$, we always consider the model alone ($\mathcal{S} \odot \mathcal{M}$) and the model combined equally with $\mathcal{S}$ ($\mathcal{S} \oplus \mathcal{M}$). All methods are using the same time limit, i.e., 10 s. This is our ultimate "real-life" evaluation, confirming that the boosted trees indeed outperform the guidance by the linear classifier and that the recursive neural network and its caching implementation is already competitive with these methods in real time. The best method $\mathcal{S} \oplus \mathcal{M}_{\text{tree}}$ solves 15.7% more problems than the original strategy $\mathcal{S}$, and 3.8% problems more than the previously best linear strategy $\mathcal{S} \oplus \mathcal{M}_{\text{lin}}$.[11] Table 2 provides also further explanation of these aggregated numbers. We show the number of unique solutions provided by each of the methods and the difference to the original strategy. Table 3 shows how useful are the particular methods when used together. Both the linear and the neural

[10] For $\mathcal{M}_{\text{lin}}$, we show the numbers after five iterations of the boosting loop (see Sect. 3.2). The values in the first round were 40.81% for the positive and 98.62% for the negative rate.

[11] We have also measured how much $\mathcal{S}$ benefits from increased time limits. It solves 1099 problems in 20 s and 1137 problems in 300 s.

Table 2. Number of problems solved (and uniquely solved) by the individual models. $\mathcal{S}+$ and $\mathcal{S}-$ indicate the number of problems gained and lost w.r.t. the baseline $\mathcal{S}$.

	$\mathcal{S}$	$\mathcal{S} \odot \mathcal{M}_{\mathrm{lin}}$	$\mathcal{S} \oplus \mathcal{M}_{\mathrm{lin}}$	$\mathcal{S} \odot \mathcal{M}_{\mathrm{tree}}$	$\mathcal{S} \oplus \mathcal{M}_{\mathrm{tree}}$	$\mathcal{S} \odot \mathcal{M}_{\mathrm{nn}}$	$\mathcal{S} \oplus \mathcal{M}_{\mathrm{nn}}$
Solved	1086	1115	1210	1231	**1256**	1167	1197
Unique	0	3	7	10	**15**	3	2
$\mathcal{S}+$	0	+119	+138	+155	**+173**	+114	+119
$\mathcal{S}-$	0	−90	−14	−10	**−3**	−33	−8

Table 3. The greedy sequence—methods sorted by their greedily computed contribution to all the problems solved.

	$\mathcal{S} \oplus \mathcal{M}_{\mathrm{tree}}$	$\mathcal{S} \oplus \mathcal{M}_{\mathrm{lin}}$	$\mathcal{S} \odot \mathcal{M}_{\mathrm{nn}}$	$\mathcal{S} \odot \mathcal{M}_{\mathrm{tree}}$	$\mathcal{S} \odot \mathcal{M}_{\mathrm{lin}}$	$\mathcal{S} \oplus \mathcal{M}_{\mathrm{nn}}$	$\mathcal{S}$
Addition	1256	33	13	11	3	2	0
Total	1256	1289	1302	1313	1316	1318	1318

models complement the boosted trees well, while the original strategy is made completely redundant.

Testing Performance of the Models on Newly Solved Problems: There are 232 problems solved by some of the six learning-based methods but not by the baseline strategy $\mathcal{S}$. To see how the trained models behave on new data, we again extract positive and negative examples from all successful proof runs on these problems. This results in around 31 000 positive testing examples and around 300 000 negative testing examples.

Table 4 shows again for each of the previously trained models the *true positive* and *true negative* rates (TPR, TNR) on these testing data. The highest TPR is again achieved by $\mathcal{M}_{\mathrm{tree}}$ and the highest TNR by $\mathcal{M}_{\mathrm{nn}}$. The accuracy of the linear model is again lower. Both the TPR and TNR testing scores are significantly lower for all methods compared to their training counterparts. TPR decreases by about 15% and TNR by about 20%. This likely shows the limits of our current learning and proof-state characterization methods. It also points to the very interesting issue of obtaining many *alternative proofs* [29] and learning from them. It seems that just using learning or reasoning is not sufficient in our AI domain, and that feedback loops combining the two multiple times [36,44] are really necessary for building strong ATP systems.

5.3 Speed of Clause Evaluation by the Learned Models

The number of generated clauses reported by E can be used as a rough estimate of the amount of work done by the prover. If we look at this statistic for those runs that timed out—i.e., did not find a proof within the given time limit—we can use it to estimate the slowdown of the clause processing rate incurred by employing a machine learner inside E. (Note that each generated clause needs to be evaluated before it is inserted on the respective queue.)

Table 4. True Positive Rate (TPR) and True Negative Rate (TNR) on testing data from the newly solved 232 problems.

	$\mathcal{M}_{\text{lin}}$	$\mathcal{M}_{\text{tree}}$	$\mathcal{M}_{\text{nn}}$
TPR	80.54%	83.35%	82.00%
TNR	62.28%	72.60%	76.88%

Table 5. The ASRPA and NSRGA ratios. ASRPA are the average ratios (and standard deviations) of the relative number of *processed* clauses with respect to $\mathcal{S}$ on problems on which all runs succeeded. NSRGA are the average ratios (and standard deviations) of the relative number of *generated* clauses with respect to $\mathcal{S}$ on problems on which all runs timed out. The numbers of problems were 898 and 681, respectively.

	$\mathcal{S}$	$\mathcal{S} \odot \mathcal{M}_{\text{lin}}$	$\mathcal{S} \oplus \mathcal{M}_{\text{lin}}$	$\mathcal{S} \odot \mathcal{M}_{\text{tree}}$	$\mathcal{S} \oplus \mathcal{M}_{\text{tree}}$	$\mathcal{S} \odot \mathcal{M}_{\text{nn}}$	$\mathcal{S} \oplus \mathcal{M}_{\text{nn}}$
ASRPA	1 ± 0	2.18 ± 20.35	0.91 ± 0.58	0.60 ± 0.98	0.59 ± 0.36	0.59 ± 0.75	0.69 ± 0.94
NSRGA	1 ± 0	0.61 ± 0.52	0.56 ± 0.35	0.42 ± 0.38	0.43 ± 0.35	0.06 ± 0.08	0.07 ± 0.09

Complementarily, the number of processed clauses when compared across those problems on which all runs succeeded may be seen as an indicator of how well the respective clause selection guides the search towards a proof (with a perfect guidance, we only ever process those clauses which constitute a proof).[12]

Table 5 compares the individual configurations of E based on the seven evaluated models with respect to these two metrics. To obtain the shown values, we first normalized the numbers on per problem basis with respect to the result of the baseline strategy $\mathcal{S}$ and computed an average across all relevant problems. The comparison of thus obtained All Solved Relative Processed Average (ASRPA) values shows that, except for $\mathcal{S} \odot \mathcal{M}_{\text{lin}}$ (which has a very high standard deviation), all other configurations on average manage to improve over $\mathcal{S}$ and find the corresponding proofs with fewer iterations of the given clause loop. This indicates better guidance towards the proof on the selected benchmarks.

The None Solved Relative Generated Average (NSRGA) values represent the speed of the clause evaluation. It can be seen that while the linear model is relatively fast (approximately 60% of the speed of $\mathcal{S}$), followed closely by the tree-based model (around 40%), the neural model is more expensive to evaluate (achieving between 6% and 7% of $\mathcal{S}$).

We note that without caching, NSRGA of $\mathcal{S} \oplus \mathcal{M}_{\text{nn}}$ drops from 7.1% to 3.6% of the speed of $\mathcal{S}$. Thus caching currently helps to approximately double the speed of the evaluation of clauses with $\mathcal{M}_{\text{nn}}$.[13] It is interesting and encouraging that despite the neural method being currently about ten times slower than the linear method—and thus generating about ten times fewer inferences within the

[12] This metric is similar in spirit to *given clause utilization* introduced by Schulz and Möhrmann [41].

[13] Note that more global caching (of, e.g., whole clauses and frequent combinations of literals) across multiple problems may further amortize the cost of the neural evaluation. This is left as future work here.

10 s time limit used for the ATP evaluation—the neural model already manages to outperform the linear model in the unassisted setting. I.e., $\mathcal{S} \odot \mathcal{M}_{\mathrm{nn}}$ is already better than $\mathcal{S} \odot \mathcal{M}_{\mathrm{lin}}$ (recall Table 2), despite the latter being much faster.

5.4 Evaluation of Feature Hashing

Finally, we evaluate the feature hashing described in Sect. 3.4. We try different hash bases in order to reduce dimensionality of the vectors and to estimate the influence on the ATP performance. We evaluate on 6 hash bases from 32k (2^{15}), 16k (2^{14}), down to 1k (2^{10}). For each hash base, we construct models $\mathcal{M}_{\mathrm{lin}}$ and $\mathcal{M}_{\mathrm{tree}}$, we compute their prediction rates, and evaluate their ATP performance.

With the hash base n, each feature must fall into one of n *buckets*. When the number of features is greater than the base—which is our case as we intend to use hashing for dimensionality reduction—collisions are inevitable. When using hash base of 32000 (ca 2^{15}) there are almost as many hashing buckets as there are features in the training data (31675). Out of these features, ca 12000 features are hashed without a collision and 12000 buckets are unoccupied. This yields a 40% probability of a collision. With lower bases, the collisions are evenly distributed.

Lower hash bases lead to larger loss of information, hence decreased performance can be expected. On the other hand, dimensionality reduction sometimes leads to better generalization (less overfitting of the learners). Also, the evaluation in the ATP can be done more efficiently in a lower dimension, thus giving the ATP the chance to process more clauses. The prediction rates and ATP performance for models with and without hashing are presented in Table 6. We compute the true positive (TPR) and negative (TNR) rates as in Sect. 5.1, and we again evaluate E's performance based on the strategy $\mathcal{S}$ in the two ways ($\odot$ and $\oplus$) as in Sect. 5.2. The best value in each row is highlighted. Both models perform comparably to the version without hashing even when the vector dimension is reduced to just 25%, i.e. 8k. With reduction to 1000 (32x), the models still provide a decent improvement over the baseline strategy $\mathcal{S}$, which solved 1086 problems. The $\mathcal{M}_{\mathrm{tree}}$ model deals with the reduction slightly better.

Table 6. Effect of feature hashing on prediction rates and ATP performance.

Model\hash size		*Without*	32k	16k	8k	4k	2k	1k
$\mathcal{M}_{\mathrm{lin}}$	TPR [%]	90.54	89.32	88.27	89.92	82.08	**91.08**	83.68
	TNR [%]	83.52	82.40	**86.01**	83.02	81.50	76.04	77.53
	$\mathcal{S} \odot \mathcal{M}$	**1115**	1106	1072	1078	1076	1028	938
	$\mathcal{S} \oplus \mathcal{M}$	1210	**1218**	1189	1202	1189	1183	1119
$\mathcal{M}_{\mathrm{tree}}$	TPR [%]	99.36	99.38	99.38	99.51	99.62	99.65	**99.69**
	TNR [%]	93.32	93.54	93.29	93.69	93.90	94.53	**94.88**
	$\mathcal{S} \odot \mathcal{M}$	1231	1231	**1233**	1232	1223	1227	1215
	$\mathcal{S} \oplus \mathcal{M}$	**1256**	1244	1244	**1256**	1245	1236	1232

Interestingly, the classification accuracy of the models (again, measured only on the training data) seems to increase with the decrease of hash base (especially for $\mathcal{M}_{\text{tree}}$). However, with this increased accuracy, the ATP performance mildly decreases. This could be caused by the more frequent collisions and thus learning on data that has been made less precise.

6 Conclusions and Future Work

We have described an efficient implementation of gradient-boosted and recursive neural guidance in E, extending the ENIGMA framework. The tree-based models improve on the previously used linear classifier, while the neural methods have, for the first time, been shown practically competitive and useful, by using extensive caching corresponding to the term sharing implemented in E. While this is clearly not the last word in this area, we believe that this is the first practically convincing application of gradient-boosted and neural clause guidance in saturation-style automated theorem provers.

There are a number of future directions. For example, research in better proof state characterization of saturation-style systems has been started recently [14, 15] and it is likely that evolving vectorial representations of the proof state will further contribute to the quality of the learning-based guidance. Our recursive neural model is just one of many, and a number of related and combined models can be experimented with.

A Strategy $\mathcal{S}$ from Experiments in Sect. 5

The following E strategy has been used to undertake the experimental evaluation in Sect. 5. The given clause selection strategy (heuristic) is defined using parameter "-H".

```
--definitional-cnf=24 --split-aggressive --simul-paramod -tKBO6 -c1 -F1
-Ginvfreq -winvfreqrank --forward-context-sr --destructive-er-aggressive
--destructive-er --prefer-initial-clauses -WSelectMaxLComplexAvoidPosPred
-H'(1*ConjectureTermPrefixWeight(DeferSOS,1,3,0.1,5,0,0.1,1,4),
    1*ConjectureTermPrefixWeight(DeferSOS,1,3,0.5,100,0,0.2,0.2,4),
    1*Refinedweight(ConstPrio,4,300,4,4,0.7),
    1*RelevanceLevelWeight2(PreferProcessed,0,1,2,1,1,1,200,200,2.5,
                                                       9999.9,9999.9),
    1*StaggeredWeight(DeferSOS,1),
    1*SymbolTypeweight(DeferSOS,18,7,-2,5,9999.9,2,1.5),
    2*Clauseweight(ConstPrio,20,9999,4),
    2*ConjectureSymbolWeight(DeferSOS,9999,20,50,-1,50,3,3,0.5),
    2*StaggeredWeight(DeferSOS,2))'
```

References

1. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., Urban, J.: Premise selection for mathematics by corpus analysis and kernel methods. J. Autom. Reason. **52**(2), 191–213 (2014)
2. Alemi, A.A., Chollet, F., Eén, N., Irving, G., Szegedy, C., Urban, J.: DeepMath - deep sequence models for premise selection. In: Lee, D.D., Sugiyama, M., Luxburg, U.V., Guyon, I., Garnett, R. (eds.) Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, Barcelona, Spain, 5–10 December 2016, pp. 2235–2243 (2016)
3. Bancerek, G., et al.: Mizar: state-of-the-art and beyond. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) CICM 2015. LNCS (LNAI), vol. 9150, pp. 261–279. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20615-8_17
4. Blanchette, J.C., Greenaway, D., Kaliszyk, C., Kühlwein, D., Urban, J.: A learning-based fact selector for Isabelle/HOL. J. Autom. Reason. **57**(3), 219–244 (2016)
5. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formalized Reason. **9**(1), 101–148 (2016)
6. Bridge, J.P., Holden, S.B., Paulson, L.C.: Machine learning for first-order theorem proving - learning to select a good heuristic. J. Autom. Reason. **53**(2), 141–172 (2014)
7. Chen, T., Guestrin, C.: XGBoost: a scalable tree boosting system. In: KDD, pp. 785–794. ACM (2016)
8. Denzinger, J., Fuchs, M., Goller, C., Schulz, S.: Learning from previous proof experience. Technical report AR99-4, Institut für Informatik, Technische Universität München (1999)
9. Ertel, W., Schumann, J.M.P., Suttner, C.B.: Learning heuristics for a theorem prover using back propagation. In: Retti, J., Leidlmair, K. (eds.) 5. Österreichische Artificial Intelligence-Tagung. INFORMATIK, vol. 208, pp. 87–95. Springer, Heidelberg (1989). https://doi.org/10.1007/978-3-642-74688-8_10
10. Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., Lin, C.-J.: LIBLINEAR: a library for large linear classification. J. Mach. Learn. Res. **9**, 1871–1874 (2008)
11. Färber, M., Brown, C.: Internal guidance for satallax. In: Olivetti and Tiwari [33], pp. 349–361
12. Färber, M., Kaliszyk, C., Urban, J.: Monte Carlo tableau proof search. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 563–579. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_34
13. Gauthier, T., Kaliszyk, C.: Premise selection and external provers for HOL4. In: Certified Programs and Proofs (CPP 2015) (2015). https://doi.org/10.1145/2676724.2693173
14. Goertzel, Z., Jakubův, J., Schulz, S., Urban, J.: ProofWatch: watchlist guidance for large theories in E. In: Avigad, J., Mahboubi, A. (eds.) ITP 2018. LNCS, vol. 10895, pp. 270–288. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94821-8_16
15. Goertzel, Z., Jakubuv, J., Urban, J.: ProofWatch meets ENIGMA: first experiments. In: Barthe, G., Korovin, K., Schulz, S., Suda, M., Sutcliffe, G., Veanes, M. (eds.) LPAR-22 Workshop and Short Paper Proceedings. Kalpa Publications in Computing, vol. 9, pp. 15–22. EasyChair (2018)
16. Goller, C., Küchler, A.: Learning task-dependent distributed representations by backpropagation through structure. In: Proceedings of International Conference on Neural Networks (ICNN 1996), vol. 1, pp. 347–352, June 1996

17. Gottlob, G., Sutcliffe, G., Voronkov, A. (eds.): Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, 16–19 October 2015. EPiC Series in Computing, vol. 36. EasyChair (2015)
18. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**, 1735–1780 (1997)
19. Jakubův, J., Urban, J.: ENIGMA: efficient learning-based inference guiding machine. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) CICM 2017. LNCS (LNAI), vol. 10383, pp. 292–302. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62075-6_20
20. Jakubův, J., Urban, J.: Enhancing ENIGMA given clause guidance. In: Rabe, F., Farmer, W.M., Passmore, G.O., Youssef, A. (eds.) CICM 2018. LNCS (LNAI), vol. 11006, pp. 118–124. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96812-4_11
21. Jakubův, J., Urban, J.: Hierarchical invention of theorem proving strategies. AI Commun. **31**(3), 237–250 (2018)
22. Jakubuv, J., Urban, J.: BliStrTune: hierarchical invention of theorem proving strategies. In: Bertot, Y., Vafeiadis, V. (eds.) Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, 16–17 January 2017, pp. 43–52. ACM (2017)
23. Joulin, A., Grave, E., Bojanowski, P., Mikolov, T.: Bag of tricks for efficient text classification. In: Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics, Short Papers, vol. 2, pp. 427–431. Association for Computational Linguistics, April 2017
24. Kaliszyk, C., Urban, J.: Learning-assisted automated reasoning with Flyspeck. J. Autom. Reason. **53**(2), 173–213 (2014)
25. Kaliszyk, C., Urban, J.: FEMaLeCoP: fairly efficient machine learning connection prover. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 88–96. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_7
26. Kaliszyk, C., Urban, J., Michalewski, H., Olšák, M.: Reinforcement learning of theorem proving. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, Canada, Montréal, 3–8 December 2018, pp. 8836–8847 (2018)
27. Kaliszyk, C., Urban, J., Vyskocil, J.: Efficient semantic features for automated reasoning over large theories. In: IJCAI, pp. 3084–3090. AAAI Press (2015)
28. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
29. Kuehlwein, D., Urban, J.: Learning from multiple proofs: first experiments. In: Fontaine, P., Schmidt, R.A., Schulz, S. (eds.) PAAR-2012. EPiC Series, vol. 21, pp. 82–94. EasyChair (2013)
30. Kühlwein, D., van Laarhoven, T., Tsivtsivadze, E., Urban, J., Heskes, T.: Overview and evaluation of premise selection techniques for large theory mathematics. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 378–392. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_30
31. Loos, S.M., Irving, G., Szegedy, C., Kaliszyk, C.: Deep network guided proof search. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, 7–12 May 2017. EPiC Series in Computing, vol. 46, pp. 85–105. EasyChair (2017)

32. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Autom. Reason. **40**(1), 35–60 (2008)
33. Olivetti, N., Tiwari, A. (eds.): IJCAR 2016. LNCS (LNAI), vol. 9706. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1
34. Otten, J., Bibel, W.: leanCoP: lean connection-based theorem proving. J. Symb. Comput. **36**(1–2), 139–161 (2003)
35. Overbeek, R.A.: A new class of automated theorem-proving algorithms. J. ACM **21**(2), 191–200 (1974)
36. Piotrowski, B., Urban, J.: ATPboost: learning premise selection in binary setting with ATP feedback. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 566–574. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_37
37. Polikar, R.: Ensemble based systems in decision making. IEEE Circuits Syst. Mag. **6**(3), 21–45 (2006)
38. Schäfer, S., Schulz, S.: Breeding theorem proving heuristics with genetic algorithms. In: Gottlob et al. [17], pp. 263–274
39. Schulz, S.: Learning search control knowledge for equational deduction. DISKI, vol. 230. Infix Akademische Verlagsgesellschaft (2000)
40. Schulz, S.: E - a brainiac theorem prover. AI Commun. **15**(2–3), 111–126 (2002)
41. Schulz, S., Möhrmann, M.: Performance of clause selection heuristics for saturation-based theorem proving. In: Olivetti and Tiwari [33], pp. 330–345
42. Urban, J.: MPTP 0.2: design, implementation, and initial experiments. J. Autom. Reason. **37**(1–2), 21–43 (2006)
43. Urban, J.: BliStr: the blind strategymaker. In: Gottlob et al. [17], pp. 312–319
44. Urban, J., Sutcliffe, G., Pudlák, P., Vyskočil, J.: MaLARea SG1 - machine learner for automated reasoning with semantic guidance. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 441–456. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_37
45. Urban, J., Vyskočil, J., Štěpánek, P.: MaLeCoP machine learning connection prover. In: Brünnler, K., Metcalfe, G. (eds.) TABLEAUX 2011. LNCS (LNAI), vol. 6793, pp. 263–277. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22119-4_21