LEARNING CONTINUOUS SEMANTIC REPRESENTATIONS OF SYMBOLIC EXPRESSIONS

Miltiadis Allamanis[1], **Pankajan Chanthirasegaran**[1], **Pushmeet Kohli**[2], **Charles Sutton**[1,3]
[1]School of Informatics, University of Edinburgh, Edinburgh, UK
[2]Microsoft Research, Microsoft, Redmond, WA, USA
[3]The Alan Turing Institute, London, UK
{m.allamanis,pankajan.chanthirasegaran,csutton}@ed.ac.uk
pkohli@microsoft.com

ABSTRACT

The question of how procedural knowledge is represented and inferred is a fundamental problem in machine learning and artificial intelligence. Recent work on program induction has proposed neural architectures, based on abstractions like stacks, Turing machines, and interpreters, that operate on abstract computational machines or on execution traces. But the recursive abstraction that is central to procedural knowledge is perhaps most naturally represented by symbolic representations that have syntactic structure, such as logical expressions and source code. Combining abstract, symbolic reasoning with continuous neural reasoning is a grand challenge of representation learning. As a step in this direction, we propose a new architecture, called *neural equivalence networks*, for the problem of learning continuous semantic representations of mathematical and logical expressions. These networks are trained to represent semantic equivalence, even of expressions that are syntactically very different. The challenge is that semantic representations must be computed in a syntax-directed manner, because semantics is compositional, but at the same time, small changes in syntax can lead to very large changes in semantics, which can be difficult for continuous neural architectures. We perform an exhaustive evaluation on the task of checking equivalence on a highly diverse class of symbolic algebraic and boolean expression types, showing that our model significantly outperforms existing architectures.

1 INTRODUCTION

Representing and learning knowledge about the world requires not only learning *declarative knowledge* about facts but also *procedural knowledge*, knowledge about how to do things, which can be complex yet difficult to articulate explicitly. The goal of building systems that learn procedural knowledge has motivated many recent architectures for learning representations of algorithms (Graves et al., 2014; Reed & de Freitas, 2016; Kaiser & Sutskever, 2016). These methods generally learn from execution traces of programs (Reed & de Freitas, 2016) or input-output pairs generated from a program (Graves et al., 2014; Kurach et al., 2015; Riedel et al., 2016; Grefenstette et al., 2015; Neelakantan et al., 2015).

However, the recursive abstraction that is central to procedural knowledge is perhaps most naturally represented not by abstract models of computation, as in that work, but by symbolic representations that have syntactic structure, such as logical expressions and source code. One type of evidence for this claim is the simple fact that people communicate algorithms using mathematical formulae and pseudocode rather than Turing machines. Yet, apart from some notable exceptions (Alemi et al., 2016; Piech et al., 2015; Allamanis et al., 2016; Zaremba & Sutskever, 2014), symbolic representations of procedures have received relatively little attention within the machine learning literature as a source of information for representing procedural knowledge.

In this paper, we address the problem of learning continuous semantic representations (SEMVECS) of symbolic expressions. The goal is to assign continuous vectors to symbolic expressions in such a way that semantically equivalent, but syntactically diverse expressions are assigned to identical (or

highly similar) continuous vectors, when given access to a training set of pairs for which semantic equivalence is known. This is an important but hard problem; learning composable SEMVECs of symbolic expressions requires that we learn about the semantics of symbolic elements and operators and how they map to the continuous representation space, thus encapsulating implicit knowledge about symbolic semantics and its recursive abstractive nature.

Our work in similar in spirit to the work of Zaremba & Sutskever (2014), who focus on learning expression representations to aid the search for computationally efficient identities. They use recursive neural networks (TREENN)[1] (Socher et al., 2012) for modelling *homogenous, single-variable* polynomial expressions. While they present impressive results, we find that the TREENN model fails when applied to more complex symbolic polynomial and boolean expressions. In particular, in our experiments we find that TREENNs tend to assign similar representations to syntactically similar expressions, even when they are semantically very different. The underlying conceptual problem is how to develop a continuous representation that follows syntax but *not too much*, that respects compositionality while also representing the fact that a small syntactic change can be a large semantic one.

To tackle this problem, we propose a new architecture, called *neural equivalence networks* (EQNETs). EQNETs learn how syntactic composition recursively composes SEMVECs, like a TREENN, but are also designed to model large changes in semantics as the network progresses up the syntax tree. As equivalence is transitive, we formulate an objective function for training based on equivalence classes rather than pairwise decisions. The network architecture is based on composing residual-like multi-layer networks, which allows more flexibility in modeling the semantic mapping up the syntax tree. To encourage representations within an equivalence class to be tightly clustered, we also introduce a training method that we call *subexpression forcing*, which uses an autoencoder to force the representation of each subexpression to be predictable from its syntactic neighbors. Experimental evaluation on a highly diverse class of symbolic algebraic and boolean expression types shows that EQNETs dramatically outperform existing architectures like TREENNs and RNNs.

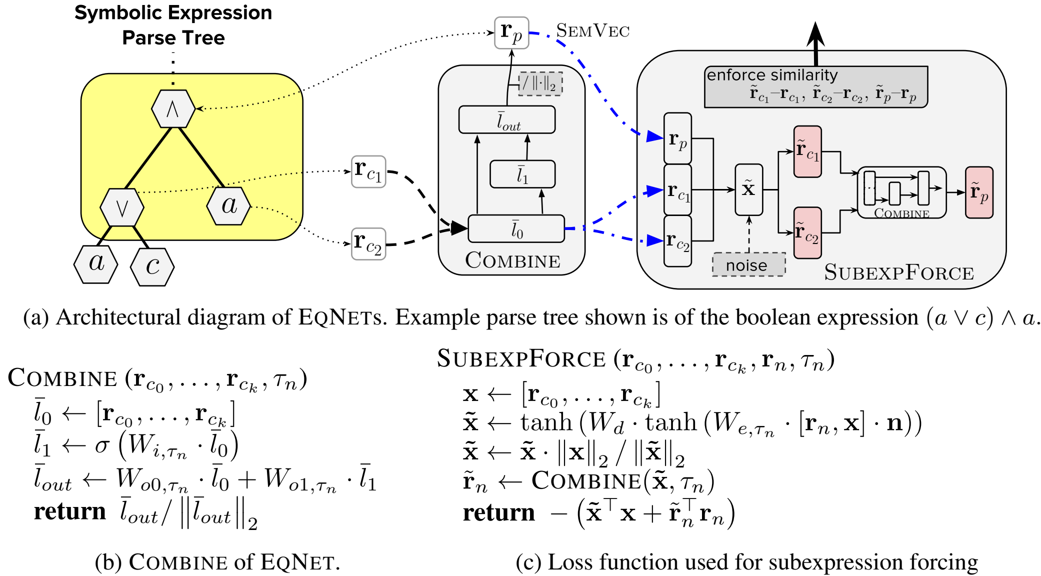
To summarize, the main contributions of our work are: (a) We formulate the problem of learning continuous semantic representations (SEMVECs) of symbolic expressions and develop benchmarks for this task. (b) We present neural equivalence networks (EQNETs), a neural network architecture that learns to represent expression semantics onto a continuous semantic representation space and how to perform symbolic operations in this space. (c) We provide an extensive evaluation on boolean and polynomial expressions, showing that EQNETs perform dramatically better than state-of-the-art alternatives. Code and data are available at `groups.inf.ed.ac.uk/cup/semvec`.

2 MODEL

In this work, we are interested in learning semantic, composable representations of mathematical expressions (SEMVEC) and learn to generate identical representations for expressions that are *semantically* equivalent, *i.e.* they belong to the same equivalence class. Equivalence is a stronger property than similarity that is habitually learned by neural networks, since equivalence is additionally a transitive relationship.

Problem Hardness. Finding the equivalence of arbitrary symbolic expressions is a NP-hard problem or worse. For example, if we focus on boolean expressions, reducing an expression to the representation of the `false` equivalence class amounts to proving its non-satisfiability — an NP-complete problem. Of course, we do *not* expect to circumvent an NP-complete problem with neural networks. A network for solving boolean equivalence would require an exponential number of nodes in the size of the formula if $P \neq NP$. Instead, our goal is to develop architectures whose inductive biases allow them to efficiently learn to solve the equivalence problems for expressions that are similar to a smaller number of expressions in a given training set. This requires that the network learn identical representations for expressions that may be syntactically different but semantically equivalent and also discriminate between expressions that may be syntactically very similar but are non-equivalent. Appendix A shows a sample of such expressions that illustrate the hardness of this problem.

[1] To avoid confusion, we use TREENN for *recursive* neural networks and retain RNN for *recurrent* neural networks.



(a) Architectural diagram of EQNETs. Example parse tree shown is of the boolean expression $(a \vee c) \wedge a$.

(b) COMBINE of EQNET.

(c) Loss function used for subexpression forcing

Figure 1: EQNET architecture.

Notation and Framework. We employ the general framework of recursive neural networks (TREENN) (Socher et al., 2012; 2013) to learn to compose subtree representations into a single representation. The TREENNs we consider operate on tree structures of the syntactic parse of a formula. Given a tree T, TREENNs learn distributed representations by recursively computing the representations of its subtrees. We denote the children of a node n as $\text{ch}(n)$ which is a (possibly empty) ordered tuple of nodes. We also use $\text{par}(n)$ to refer to the parent node of n. Each node in our tree has a type, *e.g.* a terminal node could be of type "a" referring to the variable a or of type "and" referring to a node of the logical and ($\wedge$) operation. We refer to the type of a node n as τ_n. At a high level, TREENNs retrieve the representation of a tree T rooted at node ρ, by invoking TREENET(ρ) that returns a vector representation $\mathbf{r}_\rho \in \mathbb{R}^D$, i.e., a SEMVEC, using the function

TREENET (current node n)
  if n **is not** a leaf **then**
    $\mathbf{r}_n \leftarrow$ COMBINE(TREENET(c_0), ..., TREENET(c_k), τ_n), where $(c_0, \ldots, c_k) = \text{ch}(n)$
  else
    $\mathbf{r}_n \leftarrow$ LOOKUPLEAFEMBEDDING(τ_n)
  return $\mathbf{r}_n$

The general framework of TREENET allows two points of variation, the implementation of LOOKUPLEAFEMBEDDING and COMBINE. The traditional TREENNs (Socher et al., 2013) define LOOKUPLEAFEMBEDDING as a simple lookup operation within a matrix of embeddings and COMBINE as a single-layer neural network. As discussed next, these will both prove to be serious limitations in our setting.

2.1 NEURAL EQUIVALENCE NETWORKS

We now define the *neural equivalence networks* (EQNET) that learn to compose representations of equivalence classes into new equivalence classes (Figure 1a). Our network follows the TREENN architecture, that is, our EQNETs are implemented using the TREENET, so as to model the compositional nature of symbolic expressions. However, the traditional TREENNs (Socher et al., 2013) use a single-layer neural network at each tree node. During our preliminary investigations and in Section 3, we found that single layer networks are *not* expressive enough to capture many operations, even a simple XOR boolean operator, because representing these operations required high-curvature operations in the continuous semantic representation space. Instead, we turn to multi-layer neural networks. In particular, we define the COMBINE in Figure 1b. This uses a two-layer MLP with a residual-like connection to compute the SEMVEC of each parent node in that syntax tree given that of its children. Each node type τ_n, e.g., each logical operator, has a different set of weights. We experimented with deeper networks but this did not yield any improvements. However, as TREENN

become deeper, they suffer from optimization issues, such as diminishing and exploding gradients. This is essentially because of the highly compositional nature of tree structures, where the same network (*i.e.* the COMBINE non-linear function) is used recursively, causing it to "echo" its own errors and producing unstable feedback loops. We observe this problem even with only two-layer MLPs, as the overall network can become quite deep when using two layers for each node in the syntax tree.

We resolve this issues in a few different ways. First, we constrain each SEMVEC to have unit norm. That is, we set $\text{LOOKUPLEAFEMBEDDING}(\tau_n) = C_{\tau_n} / \|C_{\tau_n}\|_2$, and we normalize the output of the final layer of COMBINE in Figure 1b. The normalization step of $\bar{l}_{out}$ and C_{τ_n} is somewhat similar to layer normalization (Ba et al., 2016), although applying layer normalization directly did not work for our problem. Normalizing the SEMVECs partially resolves issues with diminishing and exploding gradients, and removes a spurious degree of freedom in the semantic representation. As simple as this modification may seem, we found that it was vital to obtaining effective performance, and all of our multi-layer TREENNs converged to low-performing parameters without it.

However this modification is not sufficient, since the network may learn to map expressions from the same equivalence class to multiple SEMVECs in the continuous space. We alleviate this problem using a method that we call *subexpression forcing* that guides EQNET to cluster its output to one location per equivalence class. We encode each parent-children tuple $[\mathbf{r}_{c_0}, \dots, \mathbf{r}_{c_k}, \mathbf{r}_n]$ containing the (computed) representations of the children and parent node into a low-dimensional space using a denoising autoencoder. We then seek to minimize the reconstruction error of the child representations $(\tilde{\mathbf{r}}_{c_0}, \dots, \tilde{\mathbf{r}}_{c_k})$ as well as the reconstructed parent representation $\tilde{\mathbf{r}}_n$ that can be computed from the reconstructed children. Thus more formally, we minimize the return value of SUBEXPFORCE in Figure 1c where $\mathbf{n}$ is a binary noise vector with κ percent of its elements set to zero. Note that the encoder is specific to the type of τ_n. Although our SUBEXPFORCE may seem similar to the recursive autoencoders of Socher et al. (2011) it differs significantly in form and purpose, since it acts as an autoencoder on the whole parent-children representation tuple and the encoding is *not* used within the computation of the parent representation. In addition, this constraint has two effects. It forces each parent-children tuple to "live" in a low-dimensional space, providing a clustering-like behavior. Second, it implicitly joins distinct locations that belong to the same equivalence class. To illustrate the latter point, imagine two semantically equivalent c_0' and c_0'' child nodes of different nodes that have two geometrically distinct representations $\left\|\mathbf{r}_{c_0'} - \mathbf{r}_{c_0''}\right\|_2 \gg \epsilon$ and $\text{COMBINE}(\mathbf{r}_{c_0'}, \dots) \approx \text{COMBINE}(\mathbf{r}_{c_0''}, \dots)$. In some cases due to the autoencoder noise, the differences between the input tuple $\mathbf{x}', \mathbf{x}''$ that contain $\mathbf{r}_{c_0'}$ and $\mathbf{r}_{c_0''}$ will be non-existent and the decoder will be forced to predict a single location $\tilde{\mathbf{r}}_{c_0}$ (possibly different from $\mathbf{r}_{c_0'}$ and $\mathbf{r}_{c_0''}$). Then, when minimizing the reconstruction error, both $\mathbf{r}_{c_0'}$ and $\mathbf{r}_{c_0''}$ will be attracted to $\tilde{\mathbf{r}}_{c_0}$ and eventually should merge.

2.2 TRAINING

We train EQNETs from a dataset of expressions whose semantic equivalence is known. Given a training set $\mathcal{T} = \{T_1 \dots T_N\}$ of parse trees of expressions, we assume that the training set is partitioned into equivalence classes $\mathcal{E} = \{e_1 \dots e_J\}$. We use a supervised objective similar to classification; the difference between classification and our setting is that whereas standard classification problems consider a fixed set of class labels, in our setting the number of equivalence classes in the training set will vary with N. Given an expression tree T that belongs to the equivalence class $e_i \in \mathcal{E}$, we compute the probability

$$P(e_i|T) = \frac{\exp\left(\text{TREENN}(T)^\top \mathbf{q}_{e_i} + b_i\right)}{\sum_j \exp\left(\text{TREENN}(T)^\top \mathbf{q}_{e_j} + b_j\right)} \quad (1)$$

where $\mathbf{q}_{e_i}$ are model parameters that we can interpret as representations of each equivalence classes that appears in the training class, and b_i are bias terms. Note that in this work, we only use information about the equivalence class of the whole expression T, ignoring available information about subexpressions. This is without loss of generality, because if we do know the equivalence class of a subexpression of T, we can simply add that subexpression to the training set. Directly maximizing $P(e_i|T)$ would be bad for EQNET since its unit-normalized outputs cannot achieve high probabilities within the softmax. Instead, we train a max-margin objective that maximizes

classification accuracy, *i.e.*

$$\mathcal{L}_{\text{ACC}}(T, e_i) = \max\left(0, \underset{e_j \neq e_i, e_j \in \mathcal{E}}{\arg\max} \log P(e_j|T) - \log P(e_i|T) + m\right) \quad (2)$$

where $m > 0$ is a scalar margin. And therefore the optimized loss function for a single expression tree T that belongs to equivalence class $e_i \in \mathcal{E}$ is

$$\mathcal{L}(T, e_i) = \mathcal{L}_{\text{ACC}}(T, e_i) + \frac{\mu}{|Q|} \sum_{n \in Q} \text{SUBEXPFORCE}(\text{ch}(n), n) \quad (3)$$

where $Q = \{n \in T : |\operatorname{ch}(n)| > 0\}$, *i.e.* contains the non-leaf nodes of T and $\mu \in (0, 1]$ a scalar weight. We found that subexpression forcing is counterproductive early in training, before the SEMVECs begin to represent aspects of semantics. So, for each epoch t, we set $\mu = 1 - 10^{-\nu t}$ with $\nu \geq 0$.

Instead of the supervised objective that we propose, an alternative option for training EQNET would be a Siamese objective (Chopra et al., 2005) that learns about similarities (rather than equivalence) between expressions. In practice, we found the optimization to be very unstable, yielding suboptimal performance. We believe that this has to do with the compositional and recursive nature of the task that creates unstable dynamics and the fact that equivalence is a stronger property than similarity.

3 EVALUATION

Datasets. We generate datasets of expressions grouped into equivalence classes from two domains. The datasets from the BOOL domain contain boolean expressions and the POLY datasets contain polynomial expressions. In both domains, an expression is either a variable, a binary operator that combines two expressions, or a unary operator applied to a single expression. When defining equivalence, we interpret distinct variables as referring to different entities in the domain, so that, e.g., the polynomials $c \cdot (a \cdot a + b)$ and $f \cdot (d \cdot d + e)$ are not equivalent. For each domain, we generate "simple" datasets which use a smaller set of possible operators and "standard" datasets which use a larger set of more complex operators. We generate each dataset by exhaustively generating *all* parse trees up to a maximum tree size. All expressions are then simplified into a canonical from in order to determine their equivalence class and are grouped accordingly. Table 1 shows the datasets we generated. We also present in Appendix A some sample expressions. For the polynomial domain, we also generated ONEV-POLY datasets, which are polynomials over a single variable, since they are similar to the setting considered by Zaremba & Sutskever (2014) — although ONEV-POLY is still a little more general because it is not restricted to homogeneous polynomials. Learning SEMVECs for boolean expressions is already a hard problem; with n boolean variables, there are 2^{2^n} equivalence classes (*i.e.* one for each possible truth table). We split the datasets into training, validation and test sets. We create two test sets, one to measure generalization performance on equivalence classes that were seen in the training data (SEENEQCLASS), and one to measure generalization to unseen equivalence classes (UNSEENEQCLASS). It is easiest to describe UNSEENEQCLASS first. To create the UNSEENEQCLASS, we randomly select 20% of all the equivalence classes, and place all of their expressions in the test set. We select equivalence classes only if they contain at least two expressions but less than three times the average number of expressions per equivalence class. We thus avoid selecting very common (and hence trivial to learn) equivalence classes in the testset. Then, to create SEENEQCLASS, we take the remaining 80% of the equivalence classes, and randomly split the expressions in each class into training, validation, SEENEQCLASS test in the proportions 60%–15%–25%. We provide the datasets online.

Baselines. To compare the performance of our model, we train the following baselines. **TF-IDF**: learns a representation given the tokens of each expression (variables, operators and parentheses). This can capture topical/declarative knowledge but is unable to capture procedural knowledge. **GRU** refers to the token-level gated recurrent unit encoder of Bahdanau et al. (2015) that encodes the token-sequence of an expression into a distributed representation. **Stack-augmented RNN** refers to the work of Joulin & Mikolov (2015) which was used to learn algorithmic patterns and uses a stack as a memory and operates on the expression tokens. We also include two recursive neural network (TREENN) architectures. The **1-layer TREENN** which is the original TREENN also used

Table 1: Dataset statistics and results. SIMP datasets contain simple operators ("$\wedge$, $\vee$, $\neg$" for BOOL and "$+$, $-$" for POLY) while the rest contain all operators (*i.e.* "$\wedge$, $\vee$, $\neg$, $\oplus$, $\Rightarrow$" for BOOL and "$+$, $-$, $\cdot$" for POLY). $\oplus$ is the XOR operator. The number in the dataset name is the maximum tree size of the parsed expressions within that dataset. L refers to a "larger" number of 10 variables. H refers to the entropy of equivalence classes.

Dataset	# Vars	# Equiv Classes	# Exprs	H	$score_5$ (%) in UNSEENEQCLASS tf-idf	GRU	Stack RNN	TREENN 1-L	TREENN 2-L	EQNET
SIMPBOOL8	3	120	39,048	5.6	17.4	30.9	26.7	27.4	25.5	**97.4**
SIMPBOOL10^S	3	191	26,304	7.2	6.2	11.0	7.6	25.0	93.4	**99.1**
BOOL5	3	95	1,239	5.6	34.9	35.8	12.4	16.4	26.0	**65.8**
BOOL8	3	232	257,784	6.2	10.7	17.2	16.0	15.7	15.4	**58.1**
BOOL10^S	10	256	51,299	8.0	5.0	5.1	3.9	10.8	20.2	**71.4**
SIMPBOOLL5	10	1,342	10,050	9.9	53.1	40.2	50.5	3.48	19.9	**85.0**
BOOLL5	10	7,312	36,050	11.8	31.1	20.7	11.5	0.1	0.5	**75.2**
SIMPPOLY5	3	47	237	5.0	21.9	6.3	1.0	40.6	27.1	**65.6**
SIMPPOLY8	3	104	3,477	5.8	36.1	14.6	5.8	12.5	13.1	**98.9**
SIMPPOLY10	3	195	57,909	6.3	25.9	11.0	6.6	19.9	7.1	**99.3**
ONEV-POLY10	1	83	1,291	5.4	43.5	10.9	5.3	10.9	8.5	**81.3**
ONEV-POLY13	1	677	107,725	7.1	3.2	4.7	2.2	10.0	56.2	**90.4**
POLY5	3	150	516	6.7	37.8	34.1	2.2	46.8	**59.1**	55.3
POLY8	3	1,102	11,451	9.0	13.9	5.7	2.4	10.4	14.8	**86.2**

^{S}Datasets are sampled at uniform from all possible expressions, and include all equivalence classes but sampling 200 expressions per equivalence class if more expressions can be formed.

by Zaremba & Sutskever (2014). We also include a **2-layer TREENN**, where COMBINE is a classic two-layer MLP without residual connections. This shows the effect of SEMVEC normalization and subexpression forcing.

Hyperparameters. We tune the hyperparameters of the baselines and EQNET using Bayesian optimization (Snoek et al., 2012), optimizing on a boolean dataset with 5 variables and maximum tree size of 7 (not shown in Table 1). We use the average k-NN ($k = 1, \dots, 15$) statistics (described next) as an optimization metric. The selected hyperparameters are detailed in Appendix C.

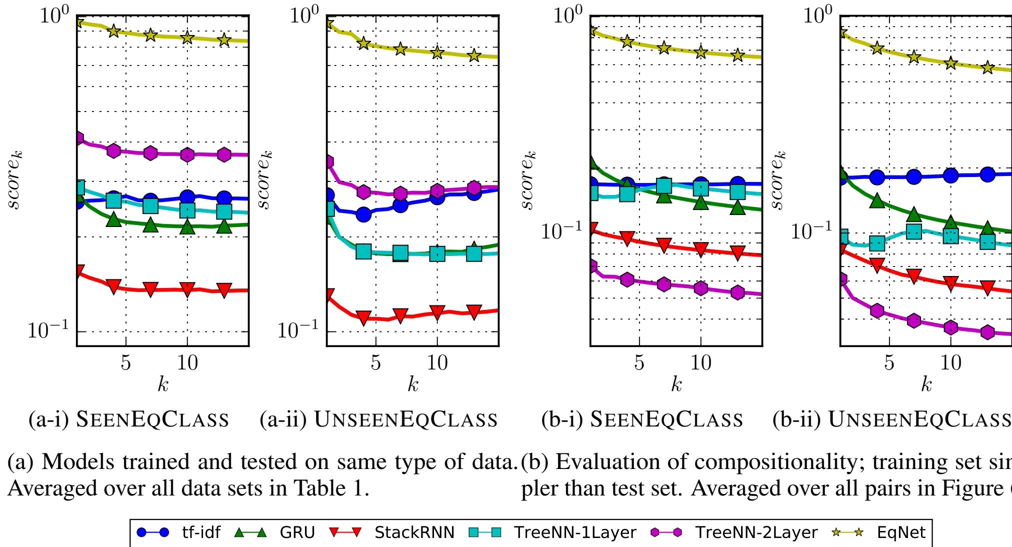
3.1 QUANTITATIVE EVALUATION

Metric. To evaluate the quality of the learned representations we count the proportion of k nearest neighbors of each expression (using cosine similarity) that belong to the same equivalence class. More formally, given a test query expression q in an equivalence class c we find the k nearest neighbors $\mathbb{N}_k(q)$ of q across all expressions, and define the score as

$$score_k(q) = \frac{|\mathbb{N}_k(q) \cap c|}{\min(k, |c|)}. \tag{4}$$

To report results for a given testset, we simply average $score_k(q)$ for all expressions q in the testset.

Evaluation. Figure 2 presents the average $score_k$ across the datasets for each model. Table 1 shows $score_5$ of UNSEENEQCLASS for each dataset. Detailed plots can be found in Appendix B. It can be clearly seen that EQNET performs better for all datasets, by a large margin. The only exception is POLY5, where the two-layer TREENN performs better. However, this may have to do with the small size of the dataset. The reader may observe that the simple datasets (containing fewer operations and variables) are easier to learn. Understandably, introducing more variables increases the size of the represented space reducing performance. The tf-idf method performs better in settings where more variables are included, because it captures well the variables and operations used. Similar observations can be made for sequence models. The one and two layer TREENNs have mixed performance; we believe that this has to do with exploding and diminishing gradients due to the deep and highly compositional nature of TREENNs. Although Zaremba & Sutskever (2014) consider a different problem to us, they use data similar to the ONEV-POLY datasets with a traditional TREENN architecture. Our evaluation suggests that EQNETs perform much better within the ONEV-POLY setting.



(a-i) SEENEQCLASS (a-ii) UNSEENEQCLASS (b-i) SEENEQCLASS (b-ii) UNSEENEQCLASS

(a) Models trained and tested on same type of data. Averaged over all data sets in Table 1.

(b) Evaluation of compositionality; training set simpler than test set. Averaged over all pairs in Figure 6.

Figure 2: Average $score_k$ (y-axis in log-scale). Markers are shown every three ticks for clarity. TREENN refers to Socher et al. (2012). Detailed, per-dataset, plots can be found in Appendix B.

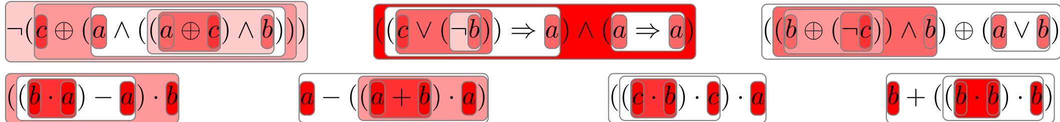


Figure 3: Visualization of $score_5$ for all expression nodes for three BOOL10 and four POLY8 test sample expressions using EQNET. The darker the color, the lower the score, *i.e.* white implies a score of 1 and dark red a score of 0.

Evaluation of Compositionality. We evaluate whether the EQNETs have successfully learned to compute compositional representations, rather than overfitting to expression trees of a small size. We evaluate this by considering a type of transfer setting, in which we train on simpler datasets, but tested on more complex ones; for example, training on the training set of BOOL5 but testing on the testset of BOOL8. We average over 11 different train-test pairs (full list in Figure 6) and present the results in Figure 2b-i and Figure 2b-ii (note the differences in scale to the two figures on the left). These graphs again show that EQNETs are dramatically better than any of the other methods, and indeed, performance is only a bit worse than in the non-transfer setting.

Impact of EQNET Components EQNETs differ from traditional TREENNs in two major components, which we analyze here. First, SUBEXPFORCE has a positive impact on performance. When training the network with and without subexpression forcing, on average, the area under the curve (AUC) of the $score_k$ decreases by 16.8% on the SEENEQCLASS and 19.7% on the UNSEENEQCLASS. This difference is smaller in the transfer setting of Figure 2b-i and Figure 2b-ii, where AUC decreases by 8.8% on average. However, even in this setting we observe that SUBEXPFORCE helps more in large and diverse datasets. The second key difference to traditional TREENNs is the output normalization at each layer. Comparing our model to the one-layer and two-layer TREENNs again, we find that output normalization results in important improvements (the two-layer TREENNs have on average 60.9% smaller AUC).

3.2 QUALITATIVE EVALUATION

Table 2 and Table 3 shows expressions whose SEMVEC nearest neighbor is of an expression of another equivalence class. Manually inspecting boolean expressions, we find that EQNET confusions happen more when a XOR or implication operator is involved. In fact, we fail to find any confused expressions for EQNET *not* involving these operations in BOOL5 and in the top 100 expressions in BOOL10. As expected, tf-idf confuses expressions with others that contain the same operators and

Table 2: *Non* semantically equivalent first nearest-neighbors from BOOL8. A checkmark indicates that the method correctly results in the nearest neighbor being from the same equivalence class.

Expression	$a \wedge (a \wedge (a \wedge (\neg c)))$	$a \wedge (a \wedge (c \Rightarrow (\neg c)))$	$(a \wedge a) \wedge (c \Rightarrow (\neg c))$
tfidf	$c \wedge ((a \wedge a) \wedge (\neg a))$	$c \Rightarrow (\neg((c \wedge a) \wedge a))$	$c \Rightarrow (\neg((c \wedge a) \wedge a))$
GRU	✓	$a \wedge (a \wedge (c \wedge (\neg c)))$	$(a \wedge a) \wedge (c \Rightarrow (\neg c))$
1L-TREENN	$a \wedge (a \wedge (a \wedge (\neg b)))$	$a \wedge (a \wedge (c \Rightarrow (\neg b)))$	$(a \wedge a) \wedge (c \Rightarrow (\neg b))$
EQNET	✓	✓	$(\neg(b \Rightarrow (b \vee c))) \wedge a$

Table 3: *Non* semantically equivalent first nearest-neighbors from POLY8. A checkmark indicates that the method correctly results in the nearest neighbor being from the same equivalence class.

Expression	$a + (c \cdot (a + c))$	$((a + c) \cdot c) + a$	$(b \cdot b) - b$
tf-idf	$a + (c + a) \cdot c$	$(c \cdot a) + (a + c)$	$b \cdot (b - b)$
GRU	$b + (c \cdot (a + c))$	$((b + c) \cdot c) + a$	$(b + b) \cdot b - b$
1L-TREENN	$a + (c \cdot (b + c))$	$((b + c) \cdot c) + a$	$(a - c) \cdot b - b$
EQNET	✓	✓	$(b \cdot b) \cdot b - b$

variables ignoring order. In contrast, GRU and TREENN tend to confuse expressions with very similar symbolic representation differing in one or two deeply nested variables or operators. In contrast, EQNET tends to confuse fewer expressions (as we previously showed) and the confused expressions tend to be more syntactically diverse and semantically related.

Figure 3 shows a visualization of $score_5$ for each node in the expression tree. One may see that as EQNET knows how to compose expressions that achieve good score, even if the subexpressions achieve a worse score. This suggests that for common expressions, (*e.g.* single variables and monomials) the network tends to select a unique location, without merging the equivalence classes or affecting the upstream performance of the network. Larger scale interactive t-SNE visualizations can be found online.

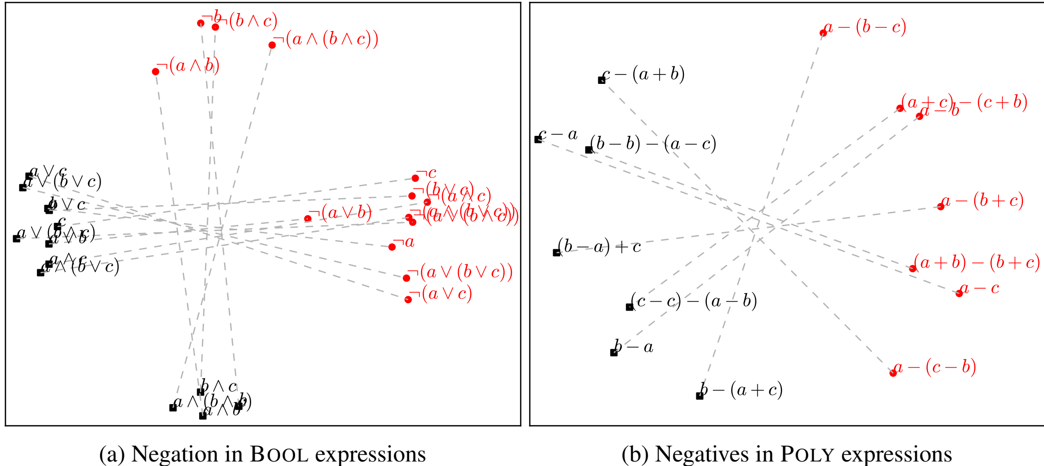
Figure 4 presents two PCA visualizations of the learned embeddings of simple expressions and their negations/negatives. It can be easily discerned that the black dots and their negations (in red) are easily discriminated in the semantic representation space. Figure 4b shows this property in a very clear manner: left-right discriminates between polynomials with a and $-a$, top-bottom between polynomials that contain b and $-b$ and the diagonal $y = x$ between c and $-c$. We observe a similar behavior in Figure 4a for boolean expressions.

4 RELATED WORK

Researchers have proposed compilation schemes that can transform any given program or expression to an equivalent neural network (Gruau et al., 1995; Neto et al., 2003; Siegelmann, 1994). One can consider a serialized version of the resulting neural network as a representation of the expression. However, it is not clear how we could compare the serialized representations corresponding to two expressions and whether this mapping preserves semantic distances.

Recursive neural networks (TREENN) (Socher et al., 2012; 2013) have been successfully used in NLP with multiple applications. Socher et al. (2012) show that TREENNs can learn to compute the *values* of some simple propositional statements. EQNET's SUBEXPFORCE may resemble recursive autoencoders (Socher et al., 2011) but differs in form and function, encoding the whole parent-children tuple to force a clustering behavior. In addition, when encoding each expression our architecture does *not* use a pooling layer but directly produces a single representation for the expression.

Mou et al. (2016) use tree convolutional neural networks to classify code into 106 student submissions tasks. Although their model learns intermediate representations of the student tasks, it is a way of learning task-specific features in the code, rather than of learning semantic representations of programs. Piech et al. (2015) also learn distributed matrix representations of programs from student submissions. However, to learn the representations, they use input and output program states and do not test over program equivalence. Additionally, these representations do not necessarily represent program equivalence, since they do not learn the representations over the exhaustive set of all possible input-output states. Allamanis et al. (2016) learn variable-sized representations of



(a) Negation in BOOL expressions (b) Negatives in POLY expressions

Figure 4: A PCA visualization of some simple *non*-equivalent boolean and polynomial expressions (black-square) and their negations (red-circle). The lines connect the negated expressions.

source code snippets to summarize them with a short function-like name. This method aims to learn summarization features in code rather than to learn representations of symbolic expression equivalence.

More closely related is the work of Zaremba & Sutskever (2014) who use a recursive neural network (TREENN) to guide the tree search for more efficient mathematical identities, limited to homogeneous single-variable polynomial expressions. In contrast, EQNETs consider at a much wider set of expressions, employ subexpression forcing to guide the learned SEMVECs to better represent equivalence, and do *not* use search when looking for equivalent expressions. Alemi et al. (2016) use RNNs and convolutional neural networks to detect features within mathematical expressions and speed the search for premise selection during automated theorem proving but do not explicitly account for semantic equivalence. In the future, SEMVECs may find useful applications within this work.

Our work is also related to recent work on neural network architectures that learn controllers/programs (Gruau et al., 1995; Graves et al., 2014; Joulin & Mikolov, 2015; Grefenstette et al., 2015; Dyer et al., 2015; Reed & de Freitas, 2015; Neelakantan et al., 2015; Kaiser & Sutskever, 2016). In contrast to this work, we do not aim to learn how to evaluate expressions or execute programs with neural network architectures but to learn continuous semantic representations (SEMVECs) of expression semantics irrespectively of how they are syntactically expressed or evaluated.

5 DISCUSSION & CONCLUSIONS

In this work, we presented EQNETs, a first step in learning continuous semantic representations (SEMVECs) of procedural knowledge. SEMVECs have the potential of bridging continuous representations with symbolic representations, useful in multiple applications in artificial intelligence, machine learning and programming languages.

We show that EQNETs perform significantly better than state-of-the-art alternatives. But further improvements are needed, especially for more robust training of compositional models. In addition, even for relatively small symbolic expressions, we have an exponential explosion of the semantic space to be represented. Fixed-sized SEMVECs, like the ones used in EQNET, eventually limit the capacity that is available to represent procedural knowledge. In the future, to represent more complex procedures, variable-sized representations would seem to be required.

ACKNOWLEDGMENTS

This work was supported by Microsoft Research through its PhD Scholarship Programme and the Engineering and Physical Sciences Research Council [grant number EP/K024043/1]. We thank the University of Edinburgh Data Science EPSRC Centre for Doctoral Training for providing additional computational resources.

REFERENCES

Alex A Alemi, Francois Chollet, Geoffrey Irving, Christian Szegedy, and Josef Urban. DeepMath–deep sequence models for premise selection. *arXiv preprint arXiv:1606.04442*, 2016.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *ICML*, 2016.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

Sumit Chopra, Raia Hadsell, and Yann LeCun. Learning a similarity metric discriminatively, with application to face verification. In *CVPR*, 2005.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. Transition-based dependency parsing with stack long short-term memory. In *ACL*, 2015.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *NIPS*, 2015.

Frédéric Gruau, Jean-Yves Ratajszczak, and Gilles Wiber. A neural compiler. *Theoretical Computer Science*, 1995.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. 2015.

Łukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. In *ICLR*, 2016.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, 2016.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. 2015.

João Pedro Neto, Hava T Siegelmann, and J Félix Costa. Symbolic processing in neural networks. *Journal of the Brazilian Computer Society*, 8(3):58–70, 2003.

Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas J Guibas. Learning program embeddings to propagate feedback on student code. In *ICML*, 2015.

Scott Reed and Nando de Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Scott Reed and Nando de Freitas. Neural programmer-interpreters. 2016.

Sebastian Riedel, Matko Bošnjak, and Tim Rocktäschel. Programming with a differentiable forth interpreter. *arXiv preprint arXiv:1605.06640*, 2016.

Hava T. Siegelmann. Neural programming language. In *Proceedings of the 12th National Conference on Artificial Intelligence*, 1994.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

Richard Socher, Jeffrey Pennington, Eric H Huang, Andrew Y Ng, and Christopher D Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *EMNLP*, 2011.

Richard Socher, Brody Huval, Christopher D Manning, and Andrew Y Ng. Semantic compositionality through recursive matrix-vector spaces. In *EMNLP*, 2012.

Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

A SYNTHETIC EXPRESSION DATASETS

Below are sample expressions within an equivalence class for the two types of datasets we consider.

BOOL8

$(\neg a) \wedge (\neg b)$	$(\neg a \wedge \neg c) \vee (\neg b \wedge a \wedge c) \vee (\neg c \wedge b)$	$(\neg a) \wedge b \wedge c$
$a\neg((\neg a) \Rightarrow ((\neg a) \wedge b))$	$c \oplus (((\neg a) \Rightarrow a) \Rightarrow b)$	$\neg((\neg b) \vee ((\neg c) \vee a))$
$\neg((b \vee (\neg(\neg a))) \vee b)$	$\neg((b \oplus (b \vee a)) \oplus c)$	$((a \vee b) \wedge c) \wedge (\neg a)$
$(\neg a) \oplus ((a \vee b) \oplus a)$	$\neg((\neg(b \vee (\neg a))) \oplus c)$	$(\neg((\neg(\neg b)) \Rightarrow a)) \wedge c$
$(b \Rightarrow (b \Rightarrow a)) \wedge (\neg a)$	$((b \vee a) \oplus (\neg b)) \oplus c)$	$(c \wedge (c \Rightarrow (\neg a))) \wedge b$
$((\neg a) \Rightarrow b) \Rightarrow (a \oplus a)$	$(\neg((b \oplus a) \wedge a)) \oplus c$	$b \wedge (\neg(b \wedge (c \Rightarrow a)))$
False	$(\neg a) \wedge (\neg b) \vee (\wedge c)$	$\neg a \vee b$
$(a \oplus a) \wedge (c \Rightarrow c)$	$(a \Rightarrow (\neg c)) \oplus (a \vee b)$	$a \Rightarrow ((b \wedge (\neg c)) \vee b)$
$(\neg b) \wedge (\neg(b \Rightarrow a))$	$(a \Rightarrow (c \oplus b)) \oplus b$	$\neg(\neg((b \vee a) \Rightarrow b))$
$b \wedge ((a \vee a) \oplus a)$	$b \oplus (a \Rightarrow (b \oplus c))$	$(\neg a) \oplus (\neg(b \Rightarrow (\neg a)))$
$((\neg b) \wedge b) \oplus (a \oplus a)$	$(b \vee a) \oplus (x \Rightarrow (\neg a))$	$b \vee (\neg((\neg b) \wedge a))$
$c \wedge ((\neg(a \Rightarrow a)) \wedge c)$	$b \oplus ((\neg a) \vee (c \oplus b))$	$\neg((a \Rightarrow (a \oplus b)) \wedge a)$

POLY8

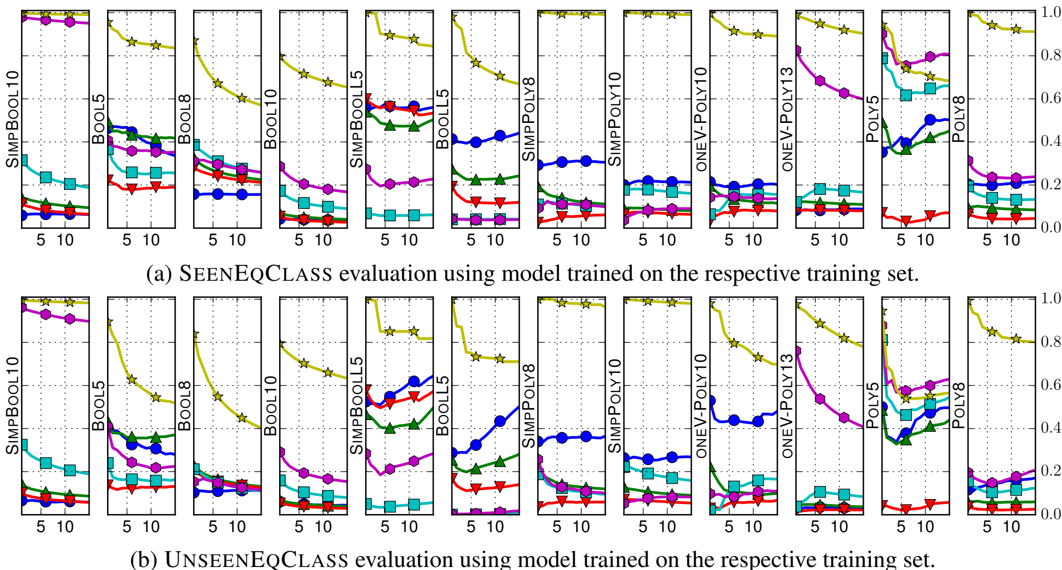
$-a - c$	c^2	b^2c^2
$(b - a) - (c + b)$	$(c \cdot c) + (b - b)$	$(b \cdot b) \cdot (c \cdot c)$
$b - (c + (b + a))$	$((c \cdot c) - c) + c$	$c \cdot (c \cdot (b \cdot b))$
$a - ((a + a) + c)$	$((b + c) - b) \cdot c$	$(c \cdot b) \cdot (b \cdot c)$
$(a - (a + a)) - c$	$c \cdot (c - (a - a))$	$((c \cdot b) \cdot c) \cdot b$
$(b - b) - (a + c)$	$c \cdot c$	$((c \cdot c) \cdot b) \cdot b$
c	$b \cdot c$	$b - c$
$c - ((c - c) \cdot a)$	$(c - (b - b)) \cdot b$	$(a - (a + c)) + b$
$c - ((a - a) \cdot c)$	$(b - (c - c)) \cdot c$	$(a - c) - (a - b)$
$((a - a) \cdot b) + c$	$(b - b) + (b \cdot c)$	$(b - (c + c)) + c$
$(c + a) - a$	$c \cdot ((b - c) + c)$	$(b - (c - a)) - a$
$(a \cdot (c - c)) + c$	$(b \cdot c) + (c - c)$	$b - ((a - a) + c)$

B DETAILED EVALUATION

Figure 5 presents a detailed evaluation for our k-NN metric for each dataset. Figure 6 shows the detailed evaluation when using models trained on simpler datasets but tested on more complex ones, essentially evaluating the learned compositionality of the models. Figure 9 show how the performance varies across the datasets based on their characteristics. As expected as the number of variables increase, the performance worsens (Figure 9a) and expressions with more complex operators tend to have worse performance (Figure 9b). In contrast, Figure 9c suggests no obvious correlation between performance and the entropy of the equivalence classes within the datasets. The results for UNSEENEQCLASS look very similar and are not plotted here.

C MODEL HYPERPARAMETERS

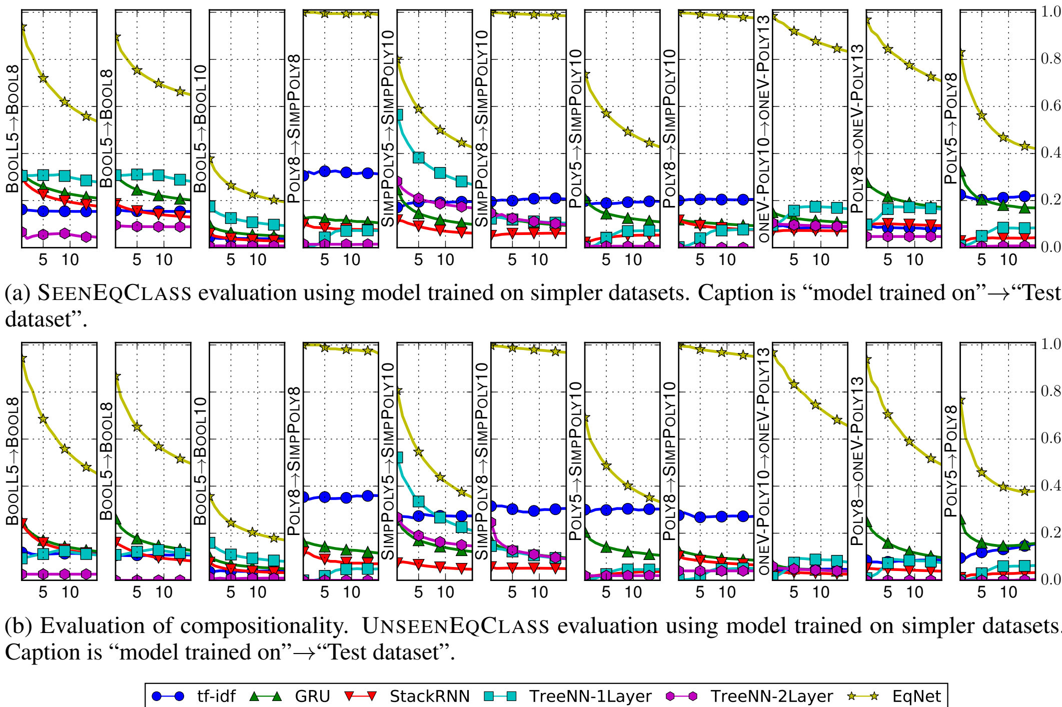
The optimized hyperparameters are detailed in Table 4. All hyperparameters were optimized using the Spearmint (Snoek et al., 2012) Bayesian optimization package. The same range of values was used for all common model hyperparameters.



(a) SeenEqClass evaluation using model trained on the respective training set.

(b) UnseenEqClass evaluation using model trained on the respective training set.

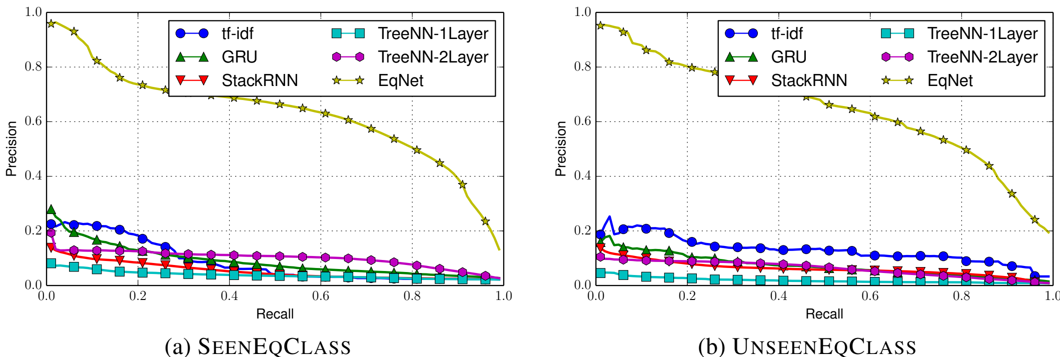
Figure 5: Evaluation of $score_x$ (y axis) for $x = 1, \ldots, 15$. on the respective SeenEqClass and UnseenEqClass where each model has been trained on. The markers are shown every five ticks of the x-axis to make the graph more clear. TreeNN refers to the model of Socher et al. (2012).



(a) SeenEqClass evaluation using model trained on simpler datasets. Caption is "model trained on"→"Test dataset".

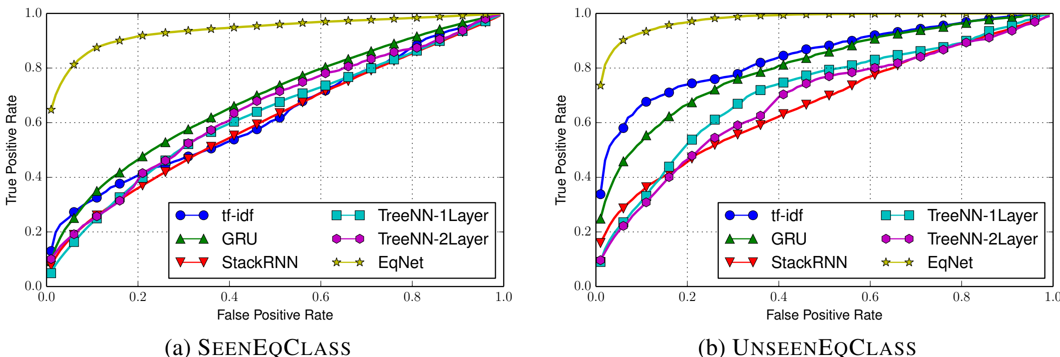
(b) Evaluation of compositionality. UnseenEqClass evaluation using model trained on simpler datasets. Caption is "model trained on"→"Test dataset".

Figure 6: Evaluation of compositionality. Evaluation of $score_x$ (y axis) for $x = 1, \ldots, 15$. The markers are shown every five ticks of the x-axis to make the graph more clear. TreeNN refers to the model of Socher et al. (2012).



(a) SEENEQCLASS (b) UNSEENEQCLASS

Figure 7: Precision-Recall curves averaged across datasets.

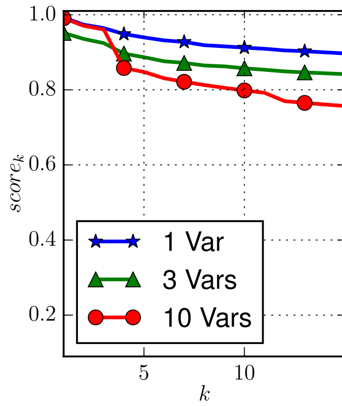


(a) SEENEQCLASS (b) UNSEENEQCLASS

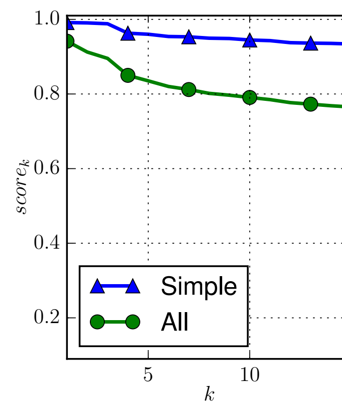
Figure 8: Receiver operating characteristic (ROC) curves averaged across datasets.

Table 4: Hyperparameters used in this work.

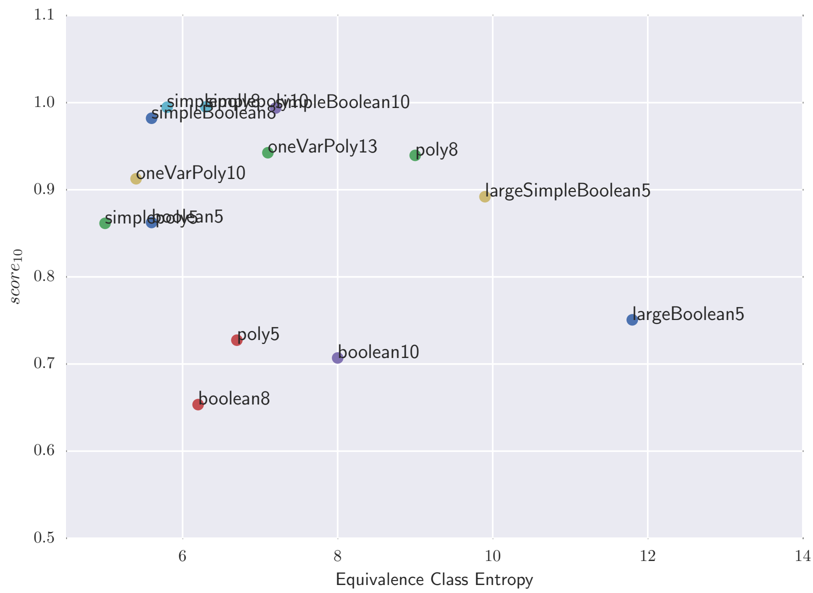
Model	Hyperparameters
EQNET	learning rate $10^{-2.1}$, rmsprop $\rho = 0.88$, momentum 0.88, minibatch size 900, representation size $D = 64$, autoencoder size $M = 8$, autoencoder noise $\kappa = 0.61$, gradient clipping 1.82, initial parameter standard deviation $10^{-2.05}$, dropout rate $.11$, hidden layer size 8, $\nu = 4$, curriculum initial tree size 6.96, curriculum step per epoch 2.72, objective margin $m = 0.5$
1-layer-TREENN	learning rate $10^{-3.5}$, rmsprop $\rho = 0.6$, momentum 0.01, minibatch size 650, representation size $D = 64$, gradient clipping 3.6, initial parameter standard deviation $10^{-1.28}$, dropout 0.0, curriculum initial tree size 2.8, curriculum step per epoch 2.4, objective margin $m = 2.41$
2-layer-TREENN	learning rate $10^{-3.5}$, rmsprop $\rho = 0.9$, momentum 0.95, minibatch size 1000, representation size $D = 64$, gradient clipping 5, initial parameter standard deviation 10^{-4}, dropout 0.0, hidden layer size 16, curriculum initial tree size 6.5, curriculum step per epoch 2.25, objective margin $m = 0.62$
GRU	learning rate $10^{-2.31}$, rmsprop $\rho = 0.90$, momentum 0.66, minibatch size 100, representation size $D = 64$, gradient clipping 0.87, token embedding size 128, initial parameter standard deviation 10^{-1}, dropout rate 0.26
StackRNN	learning rate $10^{-2.9}$, rmsprop $\rho = 0.99$, momentum 0.85, minibatch size 500, representation size $D = 64$, gradient clipping 0.70, token embedding size 64, RNN parameter weights initialization standard deviation 10^{-4}, embedding weight initialization standard deviation 10^{-3}, dropout 0.0, stack count 40



(a) Performance *vs.* Number of Variables



(b) Performance *vs.* Operator Complexity



(c) Entropy H *vs.* $score_{10}$ for all datasets

Figure 9: EQNET performance on SEENEQCLASS for various dataset characteristics