Machine Learning Projects for Iterated Distillation and Amplification

Owain Evans[a], William Saunders[b], Andreas Stuhlmüller[c]

July 3, 2019

Abstract

Iterated Distillation and Amplification (IDA) is a framework for training ML models. IDA is related to existing frameworks like imitation learning and reinforcement learning, but it aims to solve tasks for which humans cannot construct a suitable reward function or solve directly.

This document reviews IDA and proposes three projects that explore aspects of IDA. Project 1 applies IDA to problems in high-school mathematics and investigates whether learning to decompose problems can improve performance over supervised learning. Project 2 applies IDA to neural program interpretation, where neural nets are trained on the internal behavior (execution traces) of traditional computer programs. Project 3 investigates whether adaptive computation time (varying compute at inference time as a function of the input) can improve the robustness and efficiency of IDA.

Our goal in outlining these projects is to generate discussion and encourage research on IDA. We are not (as of June 2019) working on these projects, but we are interested in collaboration.

Contents

0 Background on IDA **2**
- 0.1 What is IDA? 2
- 0.2 Examples 4
- 0.3 Related Work 6
- 0.4 Project Goals 7

1 Project 1: Amplifying Mathematical Reasoning **8**
- 1.1 Motivation 8
- 1.2 Project Directions 8
- 1.3 IDA for High School Mathematics 10
- 1.4 Related Work 12

[a]University of Oxford
[b]University of Toronto
[c]Ought. Correspondence to andreas@ought.org

2 Project 2: IDA for Neural Program Interpretation **12**
2.1 Motivation 12
2.2 Project Directions 13
2.3 Related Work 14

3 Project 3: Adaptive Computation **16**
3.1 Motivation 16
3.2 Project Directions 17
3.3 IDA, Fast and Slow 17
3.4 Related Work 18

0 Background on IDA

0.1 What is IDA?

Iterated Distillation and Amplification (IDA) is a framework for training models from data [1, 2]. IDA is related to and builds on existing frameworks like supervised learning, imitation learning, and reinforcement learning. It is intended for tasks where:

1. The goal is to outperform humans at the task or to solve hard instances.
2. It is not feasible to provide demonstrations or reward signals for superhuman performance at the task.[1]
3. Humans have some high-level understanding of the task and can also provide demonstrations or reward signals for easy instances of the task.

The idea behind IDA is to bootstrap using an approach similar to AlphaZero [3], but with a learned model of human reasoning steps taking the place of the fixed game simulator.

We will explain IDA in abstract terms and then describe concrete examples. For broader discussion of IDA, including its relevance to the value alignment problem, see [1, 2, 4, 5, 6, 7].

0.1.1 Technical description of IDA

We consider the following learning problem: We want to train a model (e.g. a neural net) to solve tasks from the set T, where T contains a series of tasks that get progressively harder for humans to solve. Formally, let $T = \bigcup_{i=0}^{N} T_i$, where tasks in T_n are harder than tasks in T_{n-1} for all n.

We are given a training set which includes solutions to the easiest class of problems T_0 and human demonstrations of *decomposing* tasks T_n into finitely many slightly easier tasks in T_{n-1}.

In IDA, the initial training steps are:

[1] More precisely: it is infeasible to provide large numbers of demonstrations or sufficiently dense reward signals for methods like imitation learning or RL to work well.

1. M is trained by supervised learning[2] to reproduce the answers to the easiest tasks T_0.

2. M is trained by supervised learning to imitate the human demonstrations for **decomposing** a task $x \in T_n$ into a set of tasks in T_{n-1} and then **aggregating** the solutions to solve x.

Steps (1) and (2) are analogous to the two parts of a recursive algorithm: the *base case* and the *recursive step*. After initial training, M can solve tasks in T_1 by first decomposing them into tasks in T_0. M is then trained by supervised learning on its *own* solutions to tasks in T_1, enabling M to solve T_1 tasks *directly* (i.e. without decomposition). Solving a task in T_1 directly involves a single call to M, while solving by decomposition into tasks in T_0 requires M to be called on each of the T_0 tasks.

This process of "training on its own solutions" can be iterated. M is trained by supervised learning to directly solve increasingly hard tasks in T, where the target solutions (i.e. labels for supervised learning) are produced by M itself via decomposition into tasks M can already solve. If supervised training works perfectly at each iteration, then eventually M can solve any task in T directly (with only single call to M). (This depends on the strong assumption that humans can decompose all tasks T_n into tasks in T_{n-1} — see [8, 11] for discussion.)

We can now summarize IDA (see Figure 1). After training on steps (1) and (2), the "base case" and "recursive step", the following steps are repeated (for $n > 0$):

Amplification Step
M solves tasks in T_n by decomposing them into tasks in T_{n-1}, which it solves directly (without decomposition).

Distillation Step
M is trained by supervised learning to solve tasks in T_n directly, with target solutions coming from the Amplification Step.

It is called the "Amplification Step" because it *amplifies* the capability of model M. While M can only solve tasks in T_{n-1} directly, M can be used (via decomposition and aggregation) to solve tasks in T_n.

In the "Distillation Step", the slower amplified model (which makes multiple calls to M) is *distilled* into a faster process with a single call to M. This is like distillation for neural nets [12], where a large net (or ensemble of nets) is "distilled" into a smaller net that tries to capture the behavior of the large net. In general, it is unlikely that distillation of the slower process will be perfect. This can be addressed using RL-based distillation or by selectively choosing when to use fast M directly and when to fall back to the amplified model (see Project 3).

[2] We describe IDA based on supervised learning, similar to [8], since this is what the three projects in this document focus on. This can be substituted with RL or other training schemes, see [9, 10]. We view answering questions, problem decomposition, and aggregation of subproblem answers all as *sequence-to-sequence* problems, so we can train a single model M to solve them. We could also train distinct models.

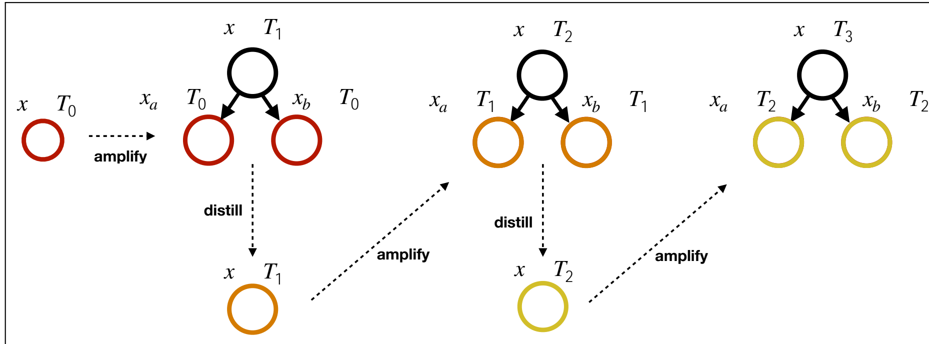

Figure 1: Diagram showing the first few Amplification and Distillation steps in IDA training. First (far left), M is trained by supervised learning on each task $x \in T_0$. Second, M solves each task $x \in T_1$ by decomposing into tasks $\{x_a, x_b\} \in T_0$ and solving these directly. This eventually produces a dataset of solved tasks $x \in T_1$, which M is trained to solve directly by supervised learning (leftmost "distill" step). This process is then repeated.

To simplify the exposition, we have presented M as being trained to decompose tasks and learn all base case solutions *before* the iterative training by Amplification and Distillation begins. In practice, these processes (learning to decompose harder problems and gathering additional base case solutions from humans, and learning to solve harder problems directly) would happen in parallel [8].

0.2 Examples

Having given an outline of IDA, we will describe two toy examples of solving problems with IDA: integer multiplication and shortest path in graphs.[3] These examples are intended to build intuition for IDA. They are not themselves practically relevant use cases for IDA.

Example 1: Multiplication

This toy example shows how one would use IDA to train a neural net M to multiply large integers. We assume that M has been pre-trained to add large integers.

Following the pattern in the previous section, the initial training set contains (1) simple multiplications (*base case*), and (2) demonstrations of decomposing multiplications into simpler multiplications (*recursive step*). M is trained as follows:

1. Train M on simple multiplications: single-digit multiplication and multiplication by 10. For example: 5 6 = 30, 10 234 = 2340, 9 8 = 72.

2. Train M to **decompose** multiplications into simpler multiplications by distribution and other algebraic rules. For example:
   8 17 = 8 10 + 8 7
   223 4 = 200 4 + 23 4 = (20 10) 4 + (20 4) + (3 4)

[3] The Multiplication example has not been implemented, but a version of the Shortest Path example is implemented in [8].

After training M successfully on (1) and (2), it would be possible to solve large multiplications exactly by recursively decomposing down to the base case. This is similar to how exact multiplication is computed by traditional algorithms or by humans. A tree illustrating this is shown in Figure 2:

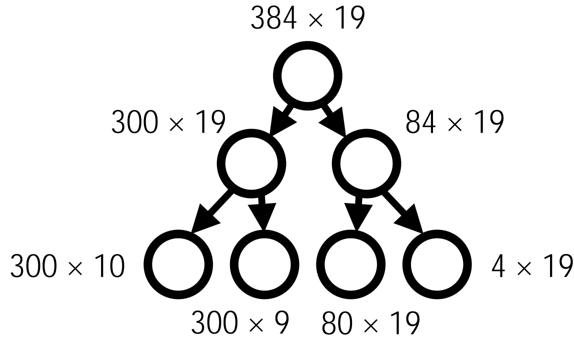

Figure 2: A multiplication problem could eventually be solved by decomposing all the way down to simple multiplications (without any distillation). Only the first two levels of decomposition are shown here.

Recursing all the way to the base case would require many calls to the neural net M. Instead, M could be trained incrementally using the IDA scheme described above. If this was successful, M could eventually compute 384×19 or 79332×2927 in a single call, without the need to ever instantiate the fully expanded slow process for large problems during training. This is a key property of IDA, since recursive decompositions can (in general) result in a number of calls to M that scales *exponentially* in the problem size, and so would be infeasible to instantiate explicitly.

This example shows that IDA can easily fail if we use a standard neural net as our model M. For example, a small MLP is not capable of learning to multiply large integers in a single call. This is the case even for training the MLP by supervised learning on ground-truth examples, e.g. training on pairs $((m, n), m \times n)$ for $m, n < 10^6$. Training by IDA will generally make learning *more* difficult, because M will be trained by supervised learning on its own answers to multiplication problems (instead of ground-truth answers).

This example also shows that IDA can fail regardless of the model M. It is not generally possible to distill an exponential tree of calls to M into a single call to M. However, there are many AI problems where research aims to better *approximate* an exponential-time computation. AlphaZero uses an IDA-like algorithm to distill an exponential game-tree expansion into a feedforward neural net. The neural net does not distill perfect play for Go or chess, but it achieves impressive performance relative to humans (and formidable performance when combined with MCTS).

Example 2: Shortest path in a graph

A recent paper by Christiano et al. [8] implements IDA and applies it to discrete algorithms problems including union find, wildcard search, and shortest path.

To apply IDA to finding the shortest path between nodes s and t in a directed graph, we need an initial training set that covers the base case and recursive step. These include:

1. A dataset of solutions to the easiest shortest path problems (for which nodes s and t are adjacent).
2. A set of demonstrations of decomposing shortest path problems into smaller shortest path problems and aggregating the results.

The decomposition in (2) is similar[4] to the following recursion, which is also used in the dynamic programming algorithm for shortest path:

$$\text{min-dist}(s, t) = \min\left(\{\text{min-dist}(s, x) + \text{dist}(x, t) \mid x \text{ adjacent to } t\}\right)$$

Here "$\text{min-dist}(s, t)$" is the minimum path length between nodes s and t, and "dist" is the distance between adjacent nodes. Christiano et al. use a Transformer model [13] as their model M and they compare two ways of training the Transformer to compute the shortest path. The first approach is the IDA model just described, which only gets labeled examples for the smallest shortest path problems and must bootstrap to solve larger problems. The second approach is regular supervised learning, where the model gets a large set of labeled examples of all sizes. The main result is that the IDA approach is successful in getting close to the performance of the supervised model. While this is a toy example, it illustrates the general aim for IDA, which is training a model for tasks where (a) there are only labels/demonstrations for easy problems, and (b) humans can provide decompositions for going from harder to easier problems.

0.3 Related Work

As noted above, there are various discussions of IDA and its relevance to AI alignment [1, 2, 6, 7]. These discussions are valuable as background but they mostly abstract away the practical details of implementing an IDA system using ML. The only paper that actually implements IDA is the aforementioned Christiano et al. However, the AlphaZero algorithm [3] is very similar to IDA [1] and work applying AlphaZero (and related algorithms AlphaGo [14] and Expert Iteration [15]) to chess/Go and to graph coloring [16] are relevant to thinking about IDA experiments. IDA depends on bootstrapping and function approximation, which are core topics in reinforcement learning [17, 18]. Recent work on Deep Q-learning is especially relevant [19].

For IDA to help address value alignment problems for advanced ML systems, it would likely need to apply to tasks (and use decompositions) that involve sophisticated reasoning in natural language. There is currently no published research that is targeted at natural language tasks. Ought and OpenAI have conducted preliminary experiments in an IDA-like setting with humans in place of ML models. These experiments are aimed at shedding light on whether different incentive structures for IDA-like approaches (e.g. different objective functions and reward signals) lead to aligned behavior. See examples from Ought [20] and OpenAI [21] and stay in touch with Ought[5] for the latest information on experiments.

[4] Christiano et al. use a slightly more complicated decomposition.

[5] https://ought.org

In the two toy examples above, IDA trains a model to imitate behavior. This is an extension of supervised learning and imitation learning, and does not involve reinforcement learning. However, as mentioned above, IDA is compatible with RL [10, 9]. There is also a framework called "Debate", which is closely related to IDA and draws on self-play reinforcement learning as a method of bootstrapping. Irving et al. [22] introduces Debate, explores analogies to computational complexity theory, and describes links between Debate and IDA.

0.4 Project Goals

The rest of this document outlines three projects that could help clarify aspects of IDA. These projects aim to extend Christiano et al [8] in a few different ways:

> The projects train a single model M to perform a wide range of types of decompositions. By contrast, Christiano et al. train a fresh model for each algorithmic problem (i.e. one for each of union find, shortest path, and wildcard search). The decompositions for these algorithmic problems have dynamic programming structure, so each model decomposes a problem instance into smaller instances of the same kind of problem (e.g. a shortest path problem is broken down into smaller shortest path problems).
>
> The projects are well suited to investigating extrapolation and generalization out of distribution. Projects 1 and 2 build on prior work in ML which tests generalization performance in mathematics and neural programming tasks. Project 3 is focused on general approaches (adaptive computation time and calibration) to improving robust generalization.
>
> In Christiano et al. the main goal is for a model trained by IDA to match the performance of a model trained by supervised learning. The IDA model has less labeled data and is trained by bootstrapping on its own labels. While this is one possible goal for our projects, we also consider the goal of improving *test-time* performance by making multiple calls to the model using amplification. Improving test-time performance by amplification is similar to the way AlphaZero uses MCTS during competitive matches to improve performance over the policy net.

There are many other possible projects on IDA. Research projects that push in the following directions seem particularly valuable:

> Produce theory and empirical knowledge about training IDA systems, analogous to knowledge of how to train supervised learning or reinforcement learning systems.
>
> Connect IDA to existing work in ML. The projects below connect to language modeling, neural programming, calibration for neural networks, and adaptive computation time. IDA is also related to semi-supervised learning, deep reinforcement learning, dynamic programming, belief propagation, etc. A research project could explore and develop any of these connections.
>
> Solve problems with IDA that can't be solved by other approaches. This is the ultimate goal of IDA, and would draw interest from the larger ML

community. The projects below are not primarily aimed to achieve this, but they may provide a useful first step. Two ways in which IDA could solve problems that can't be solved by other approaches are:

- By distilling large (i.e. exponential in problem size) trees to a fast machine learning model (as in AlphaZero).
- By learning decomposition steps from human data. This is for domains (e.g. common-sense reasoning) where we are not able to write down an algorithm to decompose problems.

1 Project 1: Amplifying Mathematical Reasoning

1.1 Motivation

Decomposition is a fundamental strategy for solving problems in mathematics. Consider the following problem from high-school mathematics:

```
g(y) = y-2
f(x) = x*g(x) + 3x^3
Find the derivative of f at x=1.
```

We can solve the problem by decomposing it into sub-problems which only depend on parts of the whole problem. Here is one possible decomposition:

```
Sub-problem 1:
g(y) = y-2
f(x) = x*g(x) + 3x^3
Write f as a polynomial in x in standard form.
Solution: f(x) = 3x^3 + x^2 - 2x

Sub-problem 2:
f(x) = 3x^3 + x^2 - 2x
Differentiate f.
Solution: f'(x) = 9x^2 + 2x - 2

Sub-problem 3:
f'(x) = 9x^2 + 2x - 2
Compute f'(1).
Solution: 9
```

These sub-problems could themselves be decomposed: the first sub-problem decomposes to substituting the function g and expanding the resulting expression.

1.2 Project Directions

The aim of this project is to train a model via IDA to solve mathematics problems. Following Christiano et al [8], we could use IDA at training time to

bootstrap from a small labeled dataset. This is like a student learning math by solving problems from a textbook with no solutions at the back of the book. Another possible goal is to use amplification at test time to apply more compute to harder problems, and so achieve better test-time performance than a standard supervised model.

There are many possible choices of mathematics problem, including:

1. Mathematical problems in a formal language (e.g. theorem proving [23]).
2. High-school algebra (in a mix of natural language and math notation [24]).
3. So-called "word problems", which are problems in natural language that need to be translated into math. For example, many problems on the American GMAT or GRE exams [25].
4. Advanced mathematics problems: competition or Olympiad math, university-level proof-based math [26].

Problems in (1) do not require working with natural language. Many such problems have a natural decomposition via brute-force search, similar to game-tree search in Chess. These problems are a good testing ground for some aspects of IDA and we encourage research on them. However this document focuses on problems in (2)-(4) and especially (2). Problems in (2)-(4) are natural language problems which don't necessarily have an obvious decomposition strategy. This makes them more similar to problems in areas outside mathematics such as science, philosophy, and common-sense reasoning.

How can we tackle natural language mathematics problems in IDA? The main question is how to produce decompositions. There are two basic options:

1. Have human experts produce decompositions of the problems.
2. Write an algorithm that solves problems by decomposition as in Christiano et al [8]. This algorithm is likely to resemble classical AI approaches ("GOFAI").

Using human data is the more general approach, as it applies outside mathematics. However, it is unlikely that the usual way people solve problems would yield the most useful decompositions. So, part of the research effort is to work out which kinds of decompositions help most and train human experts to produce them.

For option (2) above, there are two kinds of algorithm for decomposition. The first kind is an efficient algorithm that solves all problems in the class of interest (as in the multiplication and shortest path examples). A neural net trained using IDA is unlikely to perform better than such an algorithm. Experiments with IDA would aim not at state-of-the-art performance but instead at investigating certain aspects of IDA. We discuss this in the next section. The second kind of algorithm solves problems in the class *inefficiently* and so can only solve small problems in practice. In this case, it is possible that IDA can achieve state-of-the-art performance by learning to distill decompositions into a much more efficient neural algorithm. However, for advanced mathematics problems in natural language, it is not clear what this inefficient algorithm

would look like.[6] So there is a separate research project in investigating such algorithms.

In the next section, we outline a project that uses high-school math problems in natural language. We suggest starting with algorithmically generated decompositions and later extending this to decompositions provided by humans.

1.3 IDA for High School Mathematics

1.3.1 Task and Dataset

Saxton et al. [24] introduce a dataset of high-school level mathematics problems in natural language. The problems cover arithmetic, algebra, differentiation, probability, and number theory. Here are some examples:

```
Question: Solve -42*r + 27*c = -1167 and 130*r + 4*c = 372 for r.
Answer: 4

Question: Simplify sqrt(200)*2 + sqrt(200) + sqrt(200) + -4.
Answer: -4 + 40*sqrt(2)

Question: Let u(n) = -n^3 - n^2. Let e(c) = -2*c^3 + c. Let
f(j) = -118*e(j) + 54*u(j). What is the derivative of f(a)?
Answer: 546*a^2 - 108*a - 118

Question: Three letters picked without replacement from
qqqkkklkqkkk. Give prob of sequence qql.
Answer: 1/110

Question: What are the prime factors of 235232673?
Answer: 3, 13, 19, 317453

Question: Let j = -5 - 28. Is j/6*(-14) a composite number?
Answer: True
```

The problems are generated by an algorithm, so there is unlimited training data. The dataset includes test sets for both *interpolation* and *extrapolation*. The extrapolation questions have quantities that vary outside of the range encountered on the training set.

The paper includes baselines for models trained by supervised learning. The questions and answers are both represented as strings, so it can be treated as a sequence-to-sequence problem. The best performing model is a standard Transformer with 30M parameters, which achieves 76% model accuracy (probability of a correct answer) on interpolation and 50% on extrapolation.

1.3.2 Approach with IDA

How should one generate decompositions for training IDA to solve these math problems? One option is to write an algorithm for decomposing problems, rather than collecting human decompositions. The problems were generated using a compositional algorithm: consider the last example in the list above, which

[6] If mathematics problems are fully formalized, we can search over formal proofs. But if the mathematics is informal, it is much harder to provide an algorithm that would eventually (given arbitrary amounts of time and compute) solve the problems.

combines algebra (specifying an integer using equations) and number theory (checking if the integer is composite). This algorithm can be run in "reverse" to help generate decompositions. However, it is not obvious how close the resulting decompositions would be to decompositions that are a good fit for IDA training. (For instance, the best decompositions could be more or less fine-grained.)

One objective would be to do better than the supervised baseline at test time by applying amplification (i.e. decomposing the problem and using multiple calls to the model M). In particular, amplification promises to do better at extrapolation to bigger problems or problems with larger numerical quantities. Another objective is to train from a smaller number of labels using distillation and try to rival the performance of the supervised baseline.

1.3.3 Non-Amplification Baselines

The simplest baseline is supervised learning (as in Saxton et al [24]). The math problems and solutions are represented as strings, and the model is trained to map strings to strings. Another baseline would make use of the same decomposition training data as IDA. Instead of training the model to decompose problems, we could use the decomposition as an auxiliary objective. The model would be trained to produce both the decomposition and the answer. An alternative approach is to train a model to take strings as input and then produce output suitable to be fed into a symbolic math system (e.g. SymPy).

1.3.4 Questions to investigate

The project would seek to investigate some of the following questions:

- What is the test-time performance of applying amplification vs. a baseline that makes a single call to the model (distillation) vs. a baseline that was trained by supervised learning?
- What is the performance on extrapolation and on off-distribution problems?
- What is the performance of a distilled model trained by IDA (from a small number of labeled examples) vs. the supervised baseline (with a large labeled training set)?
- How does performance vary with different kinds of decomposition strategies?
- How robust is the amplified model to noise in the training data and to approximation error in the neural net?
- Can we find a decomposition strategy that makes the model more robust to errors/noise?
- When training a model to solve sub-problems, we need to pick some distribution over sub-problem examples. How does this impact performance? Are there principles for generating training data in the most useful way for IDA?

Using amplification at test time is one way to vary the amount of compute used to solve problems. Another approach is to use recurrent models and adaptive computation time. How does this compare to IDA? (See Project 3 for more discussion and references).

Can we train the model from decompositions provided by humans? What kind of decompositions should we use? Can these be augmented by algorithmically generated decompositions?

1.4 Related Work

Analysing Mathematical Reasoning Abilities of Neural Models (Saxton et al.) [24]
Dataset: https://github.com/deepmind/mathematics_dataset.
The paper by Saxton et al. discussed above.

Neural Arithmetic Logic Units (Trask et al.) [27]
A specialized neural net architecture for doing arithmetic. It achieved state-of-the-art results (as of 2018) on various tasks, including pure arithmetic and arithmetic combined with vision/natural language.

Program Induction by Rationale Generation (Ling et al.) [25]
Dataset: https://github.com/deepmind/AQuA
This paper introduces a dataset of mathematical word problems (based on US standardized tests). Humans had to "show their work" while solving the problems. The paper has a model that learns to generate these intermediate steps (in addition to learning to solving the problems).

Sigma Dolphin
microsoft.com/en-us/research/project/sigmadolphin-2/
Dataset of natural language math problems, taken from Yahoo Answers.

HOList: An Environment for Machine Learning of Higher-Order Theorem Proving (Bansal et al.) [23].
This dataset contains fully formalized proofs for a large number of theorems and a framework for training ML systems to produce proofs.

2 Project 2: IDA for Neural Program Interpretation

2.1 Motivation

Many algorithms (e.g. matrix arithmetic, shortest path, sorting) involve decomposing problems into progressively smaller problems and then aggregating results. More generally, computer programs in high-level languages decompose tasks into progressively smaller tasks, and ultimately into the primitive operations of the language. We can explore the capabilities of IDA by training a model on the decompositions used in the execution of computer programs.

Training IDA on decompositions from programs is closely related to research on Neural Program Interpretation (see Reed and de Freitas [28] and related work below) or "NPI". In NPI, a model is trained to mimic the *internal* behavior of an algorithm and not just its input-output behavior. Moreover, the model trains not just on the primitive operations of the algorithm but on its hierarchical decomposition (i.e. the way procedures call other procedures). As with IDA, one motivation for learning this internal behavior is to achieve stronger generalization. Another motivation is to integrate hierarchical discrete computation with the sort of pattern recognition in high-dimensional spaces enabled by machine learning (see third experiment in [28]).

2.2 Project Directions

The project could focus on either of the following areas:

1. **Distillation of programs**
   In contrast to most work on NPI, the goal of IDA experiments could be to use program decompositions as training data to learn more efficient "neural" programs. The idea is to distill elaborate computations into a single call to a neural net, or to combine the exact (slow) computation with distillation (as in AlphaZero).

2. **NPI within a more general framework for learning from decompositions**
   Much of the NPI work uses environments and architectures designed specially for NPI. For IDA, we would aim to replicate NPI results, but in a framework that would also allow learning from human decompositions in natural language.

We expect distillation of programs to be challenging. If we take a complicated algorithm and try to distill it into a neural net, there is likely to be approximation error—and errors will usually be larger for inputs that are off the training distribution. This will cause problems both during IDA's iterative training procedure and also at test time. Part of the project would be to investigate how well distillation works for different kinds of programs and for different ways of organizing the training curriculum. If the distilled model was *calibrated*, then IDA could recognize when distillation was likely to fail and fall back on using amplification. Project 3 (below) explores this "adaptive computation" or "meta-reasoning" approach.

2.2.1 Decision Points

There are many choices to make in devising IDA experiments in the NPI setting:

> *Which programs should we try to learn?*
> The research on NPI has focused on basic algorithms for tasks like integer arithmetic and sorting. Applying IDA to these basic algorithms could be a good starting point, as they allow comparison to existing work. However, it isn't clear how much experience with these algorithms would generalize to other applications of IDA. It could be good to consider algorithms that work with databases or knowledge bases, or to consider algorithms that

operate on human-readable structures like natural languages or images. We also think that pure functional programming is a better fit for IDA than imperative programming. See [11, 29] for relevant discussion.

What kind of built-in operations and environments should we use?
In existing work on NPI, the neural net is given outputs that correspond to basic operations on data. This makes it easier to learn algorithms that depend on those basic operations. For IDA, it would be ideal to learn these operations from examples. (If we were learning from human decompositions, we might not know about these "basic operations on data" ahead of time).

What kind of performance objective should we focus on?
Some work on NPI has focused on getting perfect performance on narrow algorithmic tasks. It's not clear if this is the right objective for IDA. We might care about (a) generalizing well to much larger inputs *most* of the time (but not in the worst case), (b) being robust to distribution shift, and (c) having one neural net learn a wide variety of algorithms.

2.2.2 Non-Amplification Baselines

For learning to predict the next step of a program from examples, simpler ML methods (random forests, logistic regression, etc.) may perform better than neural networks. For performing the same task as a traditional program, any neural program interpreter working with polynomial-sized trees will add significant overhead and so is unlikely to *improve* performance. One could also consider tasks that require ML to process the inputs (e.g. tasks involving images). The baseline in this case would be a program that defers some decisions to a classifier.

2.3 Related Work

Neural Programmer-Interpreters (Reed and de Freitas) [28]

A quote from the introduction about the motivation for the paper:

> "We may envision two approaches to providing supervision. In one, we provide a very large number of labeled examples, as in object recognition, speech and machine translation. In the other, the approach followed in this paper, the aim is to provide far fewer labeled examples, but where the labels contain richer information allowing the model to learn compositional structure. While unsupervised and reinforcement learning play important roles in perception and motor control, other cognitive abilities are possible thanks to rich supervision and curriculum learning. This is indeed the reason for sending our children to school."

Summary of the approach:

> The model (called the "neural programmer-interpreter") has a single inference core for executing three different programs (addition, sorting, rotating CAD models). So one set of LSTM parameters are used to execute

all programs. However, the different programs are stored as different embeddings, stored in a learnable persistent memory.

The sequence of the model's actions depends on the environment state and action history.

For each program (addition, sorting, rotating CAD models), the model is given a specific environment and set of actions in that environment. The model is trained to compose these actions. For integer addition, there are 1-D arrays and read-only pointers (for reading inputs), as well as 2-D scratchpads and output arrays. For rotating CAD models, there's a CAD renderer with controllable elevation and azimuth movements.

The LSTM input of the previous computation step is a vector embedding, rather than text.

Making Neural Programming Architectures Generalize via Recursion (Cai et al.) [30]

Builds on Reed and de Freitas (above) and has no new machinery. They simply allow a function to call itself. This means it can solve instances of arbitrary size using recursive function calls, each of which have bounded length. This allows for generalization off the training distribution. The hidden state of the LSTM controller is reset (to zero) at each subprogram call, but the environment state is not reset. They learn the recursion termination condition. They achieve 100% generalization on all tasks (albeit for simple tasks).

Parametrized Hierarchical Procedures for Neural Programming (Fox et al.) [31]

Their model is related to IDA for neural programming (it learns PHPs that can recursively call other PHPs). Their tasks are limited to addition and to a building a tower in gridworld. They provide a "weak supervision" motivation: learn from a mix of traces that show what information should be remembered from previous states and also from traces that omit that information.

Neural Program Lattices (Li et al.) [32]

This paper has a "weak supervision" motivation: learn from mix of full traces and traces without program calls/arguments. They generalizes to 500-digit addition, but not to 1000-digit addition.

Improving the Universality and Learnability of Neural Programmer-Interpreters with Combinator Abstraction (Xiao et al.) [33]

Adds combinator abstraction from functional programming. One motivation is to use reinforcement learning to learn programs without supervision. The combinator makes the search space smaller.

Adaptive Neural Compilation (Bunel et al.) [34]

Takes programs, translates them into a differentiable form, then uses backpropagation to optimize them. They optimize programs to be more efficient on a restricted training distribution of problems.

Recent Advances in Neural Program Synthesis (Kant) [35]

This paper provides a summary of different approaches to NPI and Neural Program Synthesis.

Learning Compositional Neural Programs with Recursive Tree Search and Planning (Pierrot et al.) [36]

NPI approach that "incorporates the strengths of Neural Programmer-Interpreters (NPI) and AlphaZero". While relevant to IDA, this approach differs in important ways.

3 Project 3: Adaptive Computation

3.1 Motivation

An ML model exhibits "adaptive computation" if it intelligently varies its computations for different inputs. For example:

1. The model selects which type of computation to run: e.g. between a slow tree search, a large neural net, and a small neural net.
2. The model prioritizes possible computations: e.g. which node to expand next in a tree search.
3. The model determines how long to run a fixed computation: e.g. how many MCTS samples, how many steps to run an RNN, etc.

A principled way to adapt computations is by "meta-level control" or "meta-reasoning". Meta-level control means applying ideas from optimal control and Bayesian decision theory to selecting computations. The idea is to treat the choice of computations as just another learning and planning problem. The choice of computations can be optimized using end-to-end supervised learning [37], model-free reinforcement learning [38], or Bayesian decision theory [39, 40, 41].

Adaptive computation and meta-level control are not a major focus in current deep learning research. Yet human cognition is often adaptive: people dynamically decide how much time to spend on a task based on its perceived difficulty. This is important when humans try to develop new ideas, which can require anything from hours to years of thinking [42, 43]. By contrast, many applications of ML have the following profile:

1. **Training time:** Large amounts of compute and time are permitted. For example, training might take months and use many CPUs and GPUs.

2. **Test/deployment time:** There are very strong constraints on compute. For example, the trained model might be part of a web application and so must perform inference in a fraction of a second.

Not all ML algorithms have this profile. When AlphaZero [3] plays competitive matches, it has time to make many calls to a neural net for each move. AlphaZero uses MCTS to investigate promising moves from the current position. As ML is applied to more tasks for which humans spend a long time thinking (e.g. mathematics, science, business strategy), the profile of AlphaZero may become more common.

IDA is well suited to adaptive computation. Training by IDA produces the following algorithms:

A quick but potentially inaccurate algorithm for solving problems, produced by distillation. This corresponds to a single call to the model M (using the terminology from Section 0.1).

An "anytime" algorithm for solving the same problems, which produces more accurate or reliable answers as a function of doing more compute. This is obtained by decomposing the problem using the learned decomposition strategy ("Amplification") and involves making multiple calls to M.

3.2 Project Directions

Adaptive computation for IDA can be applied either during the iterative training scheme or at test/deployment time:

1. **Training time**
   The goal during training is to bootstrap by using amplification to solve progressively harder problems. Adaptive computation time could be applied to select how much computation to apply to a given problem in the training set. For example, if the distilled model already performs perfectly on some class of problems, there is no need to run the slow amplified process (with many calls to model M) on this class. It's also possible to apply active learning – automatically selecting problems that are more informative about the objective.

2. **Test time**
   The goal is to perform well on a set of test problems given a time budget. If the test set is received in batch, then the algorithm would ideally spend more of its time budget on the harder instances. This requires the algorithm to recognize harder instances that will benefit from more compute time (via amplification). There are also more fine-grained questions about how to use the compute budget. For example, when using amplification, which problems should be decomposed first and how far should the recursive decomposition go?

3.3 IDA, Fast and Slow

There are many ways to use adaptive computation as part of IDA. As a starting point, we describe a simple form of adaptive computation, where the algorithm

decides between a *fast* (distilled) process for solving problems and a *slow* process that uses a fixed amount of amplification. The fast process makes a single call to a neural net, while the slow process makes at most n calls to the same net. For each input problem, the algorithm runs the fast process and has to decide whether to also run the slow process. The goal is to balance the greater accuracy of the slow process with a time/compute cost that is specified as part of the problem statement.

This decision of whether to run the slow process could be trained end-to-end, by always running both processes during training (as in [37]). A different approach would be to use a calibrated model for the fast process and decide whether to call the slow process based on the confidence of this calibrated model.

Which tasks would be a good testing ground for adaptive computation? The mathematics and neural programming tasks (Projects 1 and 2 above) are useful for testing purposes, because it's easy to vary the amount of computation that is necessary and sufficient for solving a problem. It's also valuable to explore tasks which have an "anytime" structure, where approximate solutions to the task can be improved with additional compute. This would include planning, natural language reasoning, chess, and strategic videogames.

How would this project differ from Projects 1 and 2? The aim in those projects is to explore the performance of IDA on challenging tasks. It's not clear that any clever adaptive computation strategy is required to do well on these tasks (though we can't rule it out). In a related example, AlphaZero did not select the number of MCTS samples as a function of the board position, but instead used a fixed proportion of the remaining match time. Project 3, on the other hand, is all about investigating adaptive computation for IDA. The motivation is that adaptive computation is likely to play an important role as IDA is applied to increasingly challenging tasks.

3.3.1 Non-Amplification Baselines

For any adaptive computation approach, it is important to compare against non-adaptive approaches (which use a fixed amount of compute) and also against simple heuristics for scaling up compute for harder instances. Amplification could also be compared to adaptive approaches that do not use amplification (e.g. something like [37]).

3.4 Related Work

Attending to Mathematical Language with Transformers (Wangperawong) [44]

There is a dataset (available at https://github.com/tensorflow/tensor2tensor) and paper for addition/multiplication/subtraction of numbers. The paper explores Transformers that can use more compute for larger instances.

Adaptive Computation Time for Recurrent Neural Networks (Graves) [37]

Adaptive Computation Time for RNNs adds the probability of halting computation at current step to the output of RNN. The idea is to train the adaptive

RNN end-to-end: running long computations at training time, which allows differentiation of the halting probability.

Comparing Fixed and Adaptive Computation Time for Recurrent Neural Networks (Fojo et al.) [45]

This paper presents experiments claiming to show that adaptive computation time for RNNs (as in [37] above) is not needed because similar performance is achieved with a regular RNN (which takes a fixed number of steps between predictions).

Universal Transformers (Dehghani et al.) [46]

The paper applies Adaptive Computation Time to the Transformer architecture. It shows improved performance on bAbI tasks with Adaptive Computation Time

On Calibration of Modern Neural Networks (Guo et al.) [47]

This paper tests the calibration of convolutional neural networks and finds that they are poorly calibrated by default. They show that temperature scaling (learning a temperature parameter that scales the softmax inputs) does a good job of getting the network to be calibrated. They indicate a tradeoff between accuracy and calibration (see their Figure 3).

Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles (Lakshminarayanan et al.) [48]

They get calibrated uncertainty estimates for deep neural nets by:

Using a proper scoring rule for the loss

Performing adversarial training

Using an ensemble of networks

Principles of Metalevel Control (Hay) [39]

Ph.D. thesis on metalevel control and metareasoning. Includes an excellent introduction to the subject and a review of previous work. The technical contributions include applying Bayesian decision theory to Bandit problems and applying RL to tree-search (i.e. learning a tree-search policy rather than just building in MCTS).

Learning to Search with MCTSnets (Guez et al.) [49]

They show how to learn the hyperparameters of Monte-Carlo Tree search end-to-end, by playing single-player gridworld game Sokoban. They learn a more sample efficient search than regular MCTS.

Acknowledgements

We thank Roger Grosse, Paul Christiano, Ondrej Bajgar and David Abel for valuable discussions. This work was supported by a grant from the Future of Life Institute (RFP2-178).

References

[1] Paul Christiano and Ajeya Cotra. Iterated distillation and amplification. https://ai-alignment.com/iterated-distillation-and-amplification-157debfd1616. Accessed: 2019-05-17.

[2] Paul Christiano and Dario Amodei. Learning complex goals with iterated amplification. https://openai.com/blog/amplifying-ai-training/. Accessed: 2019-05-17.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[4] Stuart Russell, Daniel Dewey, and Max Tegmark. Research priorities for robust and beneficial artificial intelligence. *Ai Magazine*, 36(4):105–114, 2015.

[5] Paul Christiano. Prosaic ai alignment. https://ai-alignment.com/prosaic-ai-control-b959644d79c2. Accessed: 2019-05-17.

[6] Paul Christiano. Towards formalizing universality. https://ai-alignment.com/towards-formalizing-universality-409ab893a456. Accessed: 2019-05-17.

[7] Jan Leike, David Krueger, Tom Everitt, Miljan Martic, Vishal Maini, and Shane Legg. Scalable agent alignment via reward modeling: a research direction. *arXiv preprint arXiv:1811.07871*, 2018.

[8] Paul Christiano, Buck Shlegeris, and Dario Amodei. Supervising strong learners by amplifying weak experts. *arXiv preprint arXiv:1810.08575*, 2018.

[9] Paul Christiano. Benign model-free rl. https://ai-alignment.com/benign-model-free-rl-4aae8c97e385. Accessed: 2019-05-17.

[10] William Saunders. Reinforcement learning in the iterated amplification framework. https://www.alignmentforum.org/posts/fq7Ehb2oWwXtZic8S/reinforcement-learning-in-the-iterated-amplification. Accessed: 2019-05-17.

[11] Andreas Stuhlmüller. Factored cognition. https://ought.org/presentations/factored-cognition-2018-05. Accessed: 2019-05-17.

[12] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[14] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[15] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370, 2017.

[16] Jiayi Huang, Mostofa Patwary, and Gregory Diamos. Coloring big graphs with alphagozero. *arXiv preprint arXiv:1902.10162*, 2019.

[17] Dimitri P Bertsekas and John N Tsitsiklis. *Neuro-dynamic programming*, volume 5. Athena Scientific Belmont, MA, 1996.

[18] Sutton Richard, Barto, and G Andrew. *Reinforcement learning: an Introduction*. MIT Press, 2018.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[20] Andreas Stuhlmüller. Ought 2018-12-31 progress update. https://ought.org/updates/2018-12-31-progress-update. Accessed: 2019-05-17.

[21] Geoffrey Irving and Amanda Askell. Ai safety needs social scientists. *Distill*, 2019. https://distill.pub/2019/safety-needs-social-scientists.

[22] Geoffrey Irving, Paul Christiano, and Dario Amodei. Ai safety via debate. *arXiv preprint arXiv:1805.00899*, 2018.

[23] Kshitij Bansal, Sarah M Loos, Markus N Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher-order theorem proving (extended version). *arXiv preprint arXiv:1904.03241*, 2019.

[24] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. *arXiv preprint arXiv:1904.01557*, 2019.

[25] Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. *arXiv preprint arXiv:1705.04146*, 2017.

[26] Mohan Ganesalingam and William Timothy Gowers. A fully automatic theorem prover with human-style output. *Journal of Automated Reasoning*, 58(2):253–291, 2017.

[27] Andrew Trask, Felix Hill, Scott E Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural arithmetic logic units. In *Advances in Neural Information Processing Systems*, pages 8035–8044, 2018.

[28] Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

[29] Paul Christiano. Meta-execution. https://ai-alignment.com/meta-execution-27ba9b34d377. Accessed: 2019-05-17.

[30] Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. *arXiv preprint arXiv:1704.06611*, 2017.

[31] Roy Fox, Richard Shin, Sanjay Krishnan, Ken Goldberg, Dawn Song, and Ion Stoica. Parametrized hierarchical procedures for neural programming. *ICLR 2018*, 2018.

[32] Chengtao Li, Daniel Tarlow, Alexander L Gaunt, Marc Brockschmidt, and Nate Kushman. Neural program lattices. 2016.

[33] Da Xiao, Jo-Yu Liao, and Xingyuan Yuan. Improving the universality and learnability of neural programmer-interpreters with combinator abstraction. *arXiv preprint arXiv:1802.02696*, 2018.

[34] Rudy R Bunel, Alban Desmaison, Pawan K Mudigonda, Pushmeet Kohli, and Philip Torr. Adaptive neural compilation. In *Advances in Neural Information Processing Systems*, pages 1444–1452, 2016.

[35] Neel Kant. Recent advances in neural program synthesis. *arXiv preprint arXiv:1802.02353*, 2018.

[36] Thomas Pierrot, Guillaume Ligner, Scott Reed, Olivier Sigaud, Nicolas Perrin, Alexandre Laterre, David Kas, Karim Beguir, and Nando de Freitas. Learning compositional neural programs with recursive tree search and planning. *arXiv preprint arXiv:1905.12941*, 2019.

[37] Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

[38] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. RL2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[39] Nicholas James Hay. *Principles of Metalevel Control*. PhD thesis, UC Berkeley, 2016.

[40] Sebastian Schulze and Owain Evans. Active reinforcement learning with monte-carlo tree search. *arXiv preprint arXiv:1803.04926*, 2018.

[41] Arthur Guez, David Silver, and Peter Dayan. Efficient bayes-adaptive reinforcement learning using sample-based search. In *Advances in neural information processing systems*, pages 1025–1033, 2012.

[42] Daniel Kahneman. *Thinking, fast and slow*. Macmillan, 2011.

[43] Owain Evans, Andreas Stuhlmüller, Chris Cundy, Ryan Carey, Zachary Kenton, Thomas McGrath, and Andrew Schreiber. Predicting human deliberative judgments with machine learning. Technical report, Technical report, University of Oxford, 2018.

[44] Artit Wangperawong. Attending to mathematical language with transformers. *arXiv preprint arXiv:1812.02825*, 2018.

[45] Daniel Fojo, Víctor Campos, and Xavier Giró-i Nieto. Comparing fixed and adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1803.08165*, 2018.

[46] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

[47] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1321–1330. JMLR. org, 2017.

[48] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in Neural Information Processing Systems*, pages 6402–6413, 2017.

[49] Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to search with mctsnets. *arXiv preprint arXiv:1802.04697*, 2018.