Artificial Human Intelligence: The Programmer’s Apprentice

Tom Dean
Google Research

In this presentation, the phrase “Artificial Human Intelligence” refers to AI systems that employ architectures modeled after the human brain, by leveraging ideas from developmental psychology and cognitive neuroscience. We start off with a discussion of the overall project with an emphasis on the underlying cognitive architecture motivated in large part by the basic linguistic and problem solving skills required of an assistant collaborating with a software engineer.

Goal: Build an end-to-end system for an individual personal assistant that focuses on a specific area of expertise, namely software engineering, that learns from experience, works collaboratively with an expert programmer and that provides value from day one.
The Programmer's Apprentice was the name of a project started by Charles Rich and Richard Waters at the MIT AI lab in 1987. The goal of the project was to develop a theory of how expert programmers analyze, synthesize, modify, explain, specify, verify and document programs, and, if possible, implement it.

The authors made it clear they did not believe that fully automatic programming was possible. They pointed out the shortcomings of the current AI systems and suggested that a completely general system would require superhuman skill. Their research plan was to build prototypes of the apprentice incrementally.

Our research plan also involves making incremental steps. However, we will make substantially larger steps by exploiting and contributing to the powerful AI technologies developed during the intervening 30 years with our primary focus on recent advances in applied machine learning and artificial neural networks.
Artificial Human Intelligence

Cognitive and systems neuroscience provide clues to engineers interested in exploiting what we know concerning how humans think about and solve problems. Fundamental to our understanding of human cognition is the essential tradeoff between fast, highly-parallel, context-sensitive, distributed connectionist-style computations and slow, serial, systematic, combinatorial symbolic computations.

The fields of cognitive and systems neuroscience provide clues to engineers interested in applying what we’ve learned about how humans communicate and solve problems. Given the difficulty of recording from human brains these clues are primarily in the form of gross architectural details and high-level functional descriptions.

Fundamental to our understanding of human cognition is the essential tradeoff between fast, parallel, contextual, distributed, connectionist computational models and slow, serial, combinatorial, symbolic computational models. Human intelligence is considered to be a hybrid of these two complementary computational strategies.
We are interested in a systems level understanding of cognition and for the purpose of this presentation we can ignore almost all of the detail shown on the left hand side of this slide.

To the extent we discuss the anatomy at all, we will focus on relatively simple models that concentrate on the location and extent of functional areas and their patterns of connectivity.

For example, the graphics labeled A, B and C shown on the right provide summaries of how information and computation relating to vision and language propagate through the cortex.
Our goal in developing systems that incorporate characteristics of human intelligence is twofold: humans provide a complete solution that we can use as a basic blueprint and then improve upon, and the resulting AI systems are likely to be well suited to developing assistants that complement and extend human intelligence while operating in a manner comprehensible to our understanding.

The study of human cognitive function at the systems level primarily consists of human subjects performing cognitive tasks in an fMRI scanner that measures brain activity reflected in changes associated with blood flow.

These studies localize cognitive activity in space and time to construct cognitive models by measuring the correlation between regions of observed brain activity and the steps carried out by subjects in solving problems.

In addition to localizing brain activity correlated with cognitive functions, Diffusion Tensor Imaging (DTI) can be used for tractographic reconstructions to infer white matter connections between putative functional regions.
The resulting theories often take the form of block diagrams that abstract from anatomy allowing various forms of computational modeling. Inset A shows a model of cortical executive function applied to the Programmer’s Apprentice including functional blocks for the Global Workspace [labeled GW] as well as blocks for ingesting programs as abstract syntax trees [labeled AST] and then representing them in a Differentiable Neural Computer [labeled DNC] that serves to interface the assistant with an integrated development environment enabling the embodiment or gamification of automatic programming. The three insets — labeled B, C and D — on the bottom of the slide represent detailed computational models for key pieces of the larger model shown in inset A, developed by Randall O’Reilly, his students and collaborators working on Leabra — acronym for “Local, Error-driven and Associative, Biologically Realistic Algorithm”.

Stanislas Dehaene and his colleagues at the Collège de France in Paris developed a computational model of consciousness that provides a practical framework for thinking about consciousness that is sufficiently detailed for much of what an engineer might care about in designing digital assistants.

Dehaene’s work extends the Global Workspace Theory of Bernard Baars. Dehaene’s version of the theory combined with Yoshua Bengio’s concept of a consciousness prior suggests a model for constructing and maintaining the cognitive states that arise and persist during complex problem solving.
Global Workspace Theory accounts for both conscious and unconscious thought with the primary distinction for our purpose being that the former has been selected for attention and the latter has not been so selected. Sensory data arrives at the periphery of the organism. The data is initially processed in the primary sensory areas located in posterior cortex, propagates forward and is further processed in increasingly-abstract multi-modal association areas. Even as information flows forward toward the front of the brain, the results of abstract computations performed in the association areas are fed back toward the primary sensory cortex. This basic pattern of activity is common in all mammals.
The human brain has evolved to handle language, or, some would say, language has co-evolved with the primate brain. We have a large frontal cortex that includes machinery responsible for conscious awareness and that depends on an extensive network of specialized neurons called spindle cells that span a large portion of the posterior cortex allowing circuits in the frontal cortex to sense activity throughout this large area and then manage this activity by creating and maintaining the persistent state vectors that are necessary when working on complex problems that require juggling many component concepts at once.
The architecture of the apprentice sensory cortex including the layers corresponding to abstract, multi-modal representations handled by the association areas can be realized as a multi-layer hierarchical neural network model consisting of standard neural network components whose local architecture is primarily determined by the sensory modality involved. This graphic shows these components as encapsulated in thought bubbles of the sort often employed in cartoons to indicate what some cartoon character is thinking. Analogously, the technical term “thought vector” is used to refer to the activation state of the output layer of such a component.
All of these capabilities can be combined in a relatively simple computational architecture that represents the apprentice’s global workspace and incorporates a model of attention that surveys activity throughout somatosensory and motor cortex, identifies the activity relevant to the current focus of attention and then maintains this state of activity so that it can readily be utilized in problem solving. In the case of the apprentice, new information is ingested into the model at the system interface, including dialog in the form of text, visual information in the form of editor screen images, and a collection of programming-related signals originating from a suite of software development tools.

Single-modality sensory information feeds into multi-modal association areas to create rich abstract representations. Attentional networks in the prefrontal cortex take as input activations occurring throughout the posterior cortex. These networks are trained by reinforcement learning to identify areas worth attending to and the resulting policy selects areas to attend to and sustain. This attentional process is guided by a prior that prefers low-dimensional thought vectors corresponding to statements about the world that are either true, highly probable or very useful for making decisions. Humans can sustain only a few such activations at a time. The apprentice need not be so constrained.
The global workspace summarizes recent experience in terms of sensory input, its integration, abstraction and inferred relevance to the context in which the underlying information was acquired. To exploit the knowledge encapsulated in such experience, the apprentice must identify and make available relevant experience. The apprentice’s experiential knowledge is encoded as tuples in a Neural Turing Machine (NTM) memory that supports associative recall. We’ll ignore the details of the encoding process to focus on how episodic memory is organized, searched and applied to solving problems.

In the biological analog of an NTM the hippocampus and entorhinal region of the frontal cortex play the role of episodic memory and several subcortical circuits including the basal ganglia comprise the controller. The controller employs keys in the form of low-dimensional vectors generated from activations highlighted in the global workspace to access related memories that are then actively maintained in the prefrontal cortex and serve to bias processing throughout the brain but particularly in those circuits highlighted in the global workspace.

Relevant technical notes from related work mentioned or quoted above:

A Neural Turing Machine (NTM) architecture contains two basic components: a neural network controller and a memory bank. Like most neural networks, the controller interacts with the external world via input and output vectors. Unlike a standard network, it also interacts with a memory matrix using selective read and write operations. By analogy to the Turing machine we refer to the network outputs that
parametrise these operations as “heads.” [...] Crucially, every component of the architecture is differentiable, making it straightforward to train with gradient descent. We achieved this by defining ‘blurry’ read and write operations that interact to a greater or lesser degree with all the elements in memory (rather than addressing a single element, as in a normal Turing machine or digital computer). The degree of blurriness is determined by an attentional “focus” mechanism that constrains each read and write operation to interact with a small portion of the memory, while ignoring the rest. Because interaction with the memory is highly sparse, the NTM is biased towards storing data without interference. Excerpt From: Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. CoRR, arXiv:1410.5401, 2014.

The large-scale architectural organization of Leabra includes three major brain systems: the posterior cortex, specialized for perceptual and semantic processing using slow, integrative learning; the hippocampus, specialized for rapid encoding of novel information using fast, arbitrary learning; and the frontal cortex/basal ganglia complex, specialized for active and flexible maintenance of goals and other context information, which serves to control or bias processing throughout the system. This latter system also incorporates various neuromodulatory systems, such as dopamine, norepinephrine, and acetylcholine, that are driven by cortical and subcortical areas (e.g., the amygdala, ventral tegmental area, substantia nigra pars compacta, and locus ceruleus) involved in emotional and motivational processing. These neuromodulators are important for regulating overall learning and decision-making characteristics of the entire system. Excerpt from: David Jilk, Christian Lebiere, Randall O’Reilly, and John R. Anderson. SAL: An explicitly pluralistic cognitive architecture. Journal of Experimental and Theoretical Artificial Intelligence, 20:197-218, 2008.
You can think of the episodic memory encoded in the hippocampus and entorhinal cortex as RAM and the actively maintained memories in the prefrontal cortex as the contents of registers in a conventional von Neumann architecture. Since the activated memories have different temporal characteristics and functional relationships with the contents of the global workspace, we implement them as two separate NTM memory systems each with its own special-purpose controller.

In the NTM paper appearing in Science, the authors point out that an associative key that only partially matches the content of a memory location can still be used to attend strongly to that location. This enables a form of pattern completion whereby the value recovered by reading the memory location includes information that is not present in the key. In general, key-value retrieval provides a rich mechanism for navigating associative data structures stored in the NTM memory, because the content of one address can effectively encode references to other addresses. The contents of memory consist of thought vectors that can be composed with other thought vectors to shape the global context for interpretation.

Additional resources for Stanford students:

Lecture by Randall O’Reilly in CS379C Spring 2018:

https://web.stanford.edu/class/cs379c/calendar_invited_talks/lectures/04/12/index.htm
Importantly, a key that only partially matches the content of a memory location can still be used to attend strongly to that location. This enables a form of pattern completion whereby the value recovered by reading the memory location includes additional information that is not present in the key. In general, key-value retrieval provides a rich mechanism for navigating associative data structures in the external memory, because the content of one address can effectively encode references to other addresses. [...] However, there are interesting parallels between the memory mechanisms of a DNC and the functional capabilities of the mammalian hippocampus. DNC memory modification is fast and can be one-shot, resembling the associative long-term potentiation of hippocampal CA3 and CA1 synapses. The hippocampal dentate gyrus, a region known to support neurogenesis, has been proposed to increase representational sparsity, thereby enhancing memory capacity. Excerpt from: Alex Graves, Greg Wayne, [...], Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538:471-476, 2016.

When trained with supervised learning, we demonstrate that a DNC can successfully answer synthetic questions designed to emulate reasoning and inference problems in natural language. We show that it can learn tasks such as finding the shortest path between specified points and inferring the missing links in randomly generated graphs, and then generalize these tasks to specific graphs such as transport networks and family trees. When trained with reinforcement learning, a DNC can complete a moving blocks puzzle in which changing goals are specified by sequences of symbols. Taken together, our results demonstrate that DNCs have the capacity to solve complex, structured tasks that are inaccessible to neural networks without external read–write memory. Excerpt from: Alex Graves, Greg Wayne, [...], Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538:471-476, 2016.
Here is a schematic sketch of the apprentice architecture illustrating the components mentioned so far. There are two key components missing from this picture. Both components concern how the apprentice thinks about programs and programming. The first component is concerned with how experience is encoded and subsequently applied to solve problems. The second component addresses the problem of how programs are represented and manipulated internally so as to effectively apply connectionist methods.
In the mammalian brain, information pertaining to sensing and motor control is topographically mapped to reflect the intrinsic structure of that information required for interpretation. This was early recognized in the work of Hubel and Wiesel on the visual cortex and Wilder Penfield in his work developing the notion of a *cortical homunculus* in the primary motor and somatosensory areas of the cortex as shown here. Such maps relate to the idea of embodied intelligence.

The integrated development environment and its associated software engineering tools constitute an extension of the apprentice’s capabilities in much the same way that a piano or violin extends a musician. The extension becomes an integral part of the person possessing it and over time their brain creates a topographic map that facilitates interacting with the extension.

As engineers designing the apprentice, part of our job is to create tools that enable the apprentice to learn its trade and eventually become an expert. Conventional, IDE tools simplify the job of software engineers in designing software. The IDE that we build for the apprentice (FIDE) will be integrated into the apprentice’s cognitive architecture so that tasks like stepping a debugger or setting breakpoints are as easy for the apprentice as balancing parentheses and checking for spelling errors in a text editor is for us.
The apprentice employs several cognitive prostheses that serve as physical extensions and constitute its embodiment. The simplest of these prostheses corresponds to a browser window and editor buffer shared with the programmer. The most sophisticated corresponds to a fully instrumented integrated development environment called FIDE. The API providing the interface with these extensions supports the representation of information relating to programming and execution of commands for carrying out activities associated with manipulating this information. These extensions constitute the body of the apprentice and are implemented as a Differentiable Neural Computer (DNC). Think of them as the analog of an Atari game console for programming. They enable us to essentially gamify the process of analyzing, synthesizing and executing computer programs.
This slide summarizes the architectural components introduced so far in a single model. Data in the form of text transcriptions of ongoing dialogue, source code and related documentation and output from the integrated development environment are the primary input to the system and are handled by relatively standard neural network models. The Q-network for the attentional RL system is realized as a multi-layer convolutional network. The two DNC controllers are straightforward variations on existing network models with a second controller responsible for maintaining a priority queue encodings of relevant past experience retrieved from episodic memory. The nondescript box labeled motor “cortex” includes the machinery for managing dialogue and handling tasks related to programming and code synthesis. In the remainder of this talk, I’ll briefly touch on one aspect of dialogue management and then Rishabh will provide an overview of our strategy for tackling code synthesis.
Programs along with examples of their input and output have several representations within the apprentice. Rishabh will make this more concrete in his part of the presentation. Suffice it to say for now that the apprentice works with syntactic representations in the form of abstract syntax trees, semantic representations derived from execution traces and a variety of text descriptions including natural and formal language program specifications and related programming knowledge.
Representing and Reasoning About Structured Programming

As an example of how the apprentice architecture simplifies collaboration with the programmer, the two share an interactive FIDE editor window that displays the source code for the current program and a shell for running FIDE programming tools. The apprentice sees exactly what the programmer sees. Access to the corresponding bit map allows the apprentice to understand visual references like "the assignment statement inside the for loop" or "the above procedure definition".

Both the apprentice and the programmer can modify or make references to text appearing in the FIDE window by pointing to items or highlighting regions of the source code. The apprentice, however, also represents and manipulates the current program internally as an abstract syntax tree (AST) within the FIDE. The text and AST versions of the programs represented in the FIDE are automatically synchronized so that the program under development is forced to adhere to certain syntactic invariants.

As a first step in simplifying use of the IDE for coding the apprentice can manipulate programs as abstract syntax trees and easily move back and forth between the AST representation and the original source code in collaborating with the programmer. We believe the highly-parallel, contextual, connectionist computations that dominate in human information processing will complement the primarily-serial, combinatorial, symbolic computations that characterize conventional information processing and will have a considerable positive impact on the development of practical automatic programming methods.
Relevant technical notes from related work mentioned or quoted above:

[Most approaches to representing structured programs attempt] to transfer natural language methods and [do] not capitalize on the unique opportunities offered by code's known semantics. For example, long-range dependencies induced by using the same variable or function in distant locations are often not considered. **We propose to use** graphs to represent both the syntactic and semantic structure of code and use graph-based deep learning methods to learn to reason over program structures. **Excerpt from:** Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.
To support this hypothesis, we are developing distributed representations for programs that enable the apprentice to efficiently search for solutions to programming problems by allowing the apprentice to easily move back and forth between the two paradigms, exploiting both conventional approaches to program synthesis and recent work on machine learning and inference in artificial neural networks. Neural Turing Machines coupled with reinforcement learning are capable of learning simple programs. We are interested in representing structured programs expressed in modern programming languages. Our approach is to alter the NTM controller and impose additional structure on the NTM memory designed to support procedural abstraction.

We use pointers to represent programs as abstract syntax trees and partition the NTM memory, as in a conventional computer, into program memory and a LIFO execution (call) stack to support recursion and reentrant procedure invocations, including call frames for return addresses, local variable values and related parameters. The NTM controller manages the program counter and LIFO call stack to simulate the execution of programs stored in program memory. Program statements are represented as embedding vectors and the system learns to evaluate these representations in order to generate intermediate results that are also embeddings. It is a simple matter to execute the corresponding code in the FIDE and incorporate any of the results as features in embeddings.

What could we do with such a representation? It is important to understand why we don’t work with some intermediate representation like bytecodes. By working in the
target programming language, we can take advantage of both the abstractions afforded by the language and the expert knowledge of the programmer about how to exploit those abstractions. The apprentice is bootstrapped with several statistical language models: one trained on a natural language corpus and the other on a large code repository. Using these resources and the means of representing and manipulating program embeddings, we intend to train the apprentice to predict the next expression in a partially constructed program by using a variant of imagination-based planning. As another example, we will attempt to leverage NLP methods to generate proposals for substituting one program fragment for another as the basis for code completion.

Relevant technical notes from related work mentioned or quoted above:

Based on the context, the pointer mixture network learns to either generate a within-vocabulary word through an RNN component, or regenerate an OoV word from local context through a pointer component. [...] Besides the traditional context attention, we also propose a parent attention for the AST-based code completion. Intuitively, different hidden states within the context window should have different degrees of relevance to the current prediction. Excerpt from: Jian Li, Yue Wang, Irwin King, and Michael R. Lyu. Code completion with neural attention and pointer networks. CoRR, arXiv:1711.09573, 2017.

Here we introduce the “Imagination-based Planner”, the first model-based, sequential decision-making agent that can learn to construct, evaluate, and execute plans. Before any action, it can perform a variable number of imagination steps, which involve proposing an imagined action and evaluating it with its model-based imagination. All imagined actions and outcomes are aggregated, iteratively, into a “plan context” which conditions future real and imagined actions. Excerpt from: Razvan Pascanu, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sébastien Racanière, David P. Reichert, Theophane Weber, Daan Wierstra, and Peter Battaglia. Learning model-based planning from scratch. CoRR, arXiv:1707.06170, 2017.

The Differentiable Neural Program (DNP) representation and associated NTM controller for managing the call stack and single-stepping through such programs allow us to exploit the advantages of distributed vector representations to predict the next statement in a program under construction. This model makes it easy to take advantage of supplied natural language descriptions and example input/output pairs plus incorporate semantic information in the form of execution traces generated by utilizing the IDE to evaluate each statement and encoding information about local variables on the stack.

This slide illustrates how we make use of input/output pairs as program invariants to narrow search for the next statement in the evolving target program. At any given moment the call stack contains the trace of a single conditioned path through the developing program. A single path is unlikely to provide sufficient information to account for the constraints implicit in all of the sample input/output pairs and so we intend to use a limited lookahead planning system to sample multiple execution traces in order to inform the prediction of the next program statement.

These so-called imagination-augmented agents implement a novel architecture for reinforcement learning that balances exploration and exploitation using imperfect models to generate trajectories from some initial state using actions sampled from a rollout policy. These trajectories are then combined and fed to an output policy along with the action proposed by a model-free policy to make better decisions. There are related reinforcement learning architectures that perform Monte Carlo Markov chain search to apply and collect the constraints from multiple input/output pairs.
Relevant technical notes from related work mentioned or quoted above:


Unlike images and text, a program has an unambiguous semantic meaning that can be difficult to capture by only considering its syntax (i.e. syntactically similar programs can exhibit vastly different run-time behavior), which makes syntax-based program embeddings fundamentally limited. This paper proposes a novel semantic
program embedding that is learned from program execution traces. Our key insight is that program states expressed as sequential tuples of live variable values not only captures program semantics more precisely, but also offer a more natural fit for Recurrent Neural Networks to model. Excerpt from: Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. *International Conference on Learning Representations*, 2018.
Graph nets are a neural network framework for constructing, modifying and performing inference on differentiable encodings of graphical structures. Li et al. describe a model known as a Gated Graph Sequence Neural Network (GGS-NN) that operates on graph nets to produce sequences from graph-structured input. Daniel Johnson introduced the Gated Graph Transformer Neural Network (GGT-NN), an extension of GGS-NNs that uses graph-structured data as an intermediate representation. The model can learn to construct and modify graphs in sophisticated ways based on textual input, and also to use the graphs to produce a variety of outputs.

The network on left demonstrates how to package the five general transformations described in Johnson’s paper to provide a Swiss-army-knife utility that can be used to manipulate abstract syntax trees in code synthesis simplifying the construction of differentiable neural programs introduced earlier. This graph-nets utility could be integrated into a reinforcement-learning code synthesis module that would learn how to repair programs or perform other forms of synthesis by learning how to predict the best alterations on the program under construction.
The imagination-based planning for reinforcement learning framework developed by engineers at DeepMind serves as an example for how the code synthesis module might be implemented. The IBP model combines three separate adaptive components: (a) the CONTROLLER + MEMORY system which maps a state $s$ in $S$ and history $h$ in $H$ to an action $a$ in $A$; (b) the MANAGER maps a history $h$ in $H$ to a route $u$ in $U$ that determines whether the system performs an action in the world (an operation in the FIDE, e.g., single-step the current code) or performs an imagination step (generates a proposal for modifying the existing code under construction); the IMAGINATION MODEL is form of dynamical systems model that maps a pair consisting of a state $s$ in $S$ and an action $a$ in $A$ to an imagined next state $s'$ in $S$ and scalar-valued reward $r$ in $R$.

The IMAGINATION MODEL is implemented as an interaction network that could implemented using the graph-nets framework introduced on the previous slide. The three components are trained by three distinct, concurrent, on-policy training loops. The IBP framework allows code synthesis to alternate between acting by actually modifying and running code, and applying the model to investigate and analyze what would happen if you did act. The IBP framework shown here allows code synthesis to alternate between acting by actually modifying and running code, and applying the model to investigate and analyze what would happen if you did act. The MANAGER chooses whether to execute a command or predict (imagine) its result and can produce a tree $ht$ of imagined results. The CONTROLLER takes this tree plus the compiled history and chooses an action (command) to carry out in the FIDE.
Relevant technical notes from related work mentioned or quoted above:

The *manager* is a discrete policy which maps a history, $h$, to a route, $u$ in $U$. The $u$ determines whether the agent will execute an action in the environment, or imagine the consequences of a proposed action. If imagining, the route can also select which previously imagined, or real, state to imagine from. The *controller* is a contextualized action policy which maps a state $s$ in $S$ and a history to an action, $a$ in $A$. The state which is provided as input to the controller is determined by the manager’s choice of $u$. If executing, the actual state, $s_j$, is always used. If imagining, the state $s_j^l$ is used, as mentioned above. The *imagination* is a model of the world, which maps states, $s$ in $S$, and actions, $a$ in $A$, to consequent states, $s_0$ in $S$, and scalar rewards, $r$ in $R$. The *memory* recurrently aggregates the external and internal data generated from one iteration, $d$ in $D$, to update the history, $h$. Excerpt from: Razvan Pascanu, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sébastien Racanière, David P. Reichert, Theophane Weber, Daan Wierstra, and Peter Battaglia. Learning model-based planning from scratch. *CoRR*, arXiv:1707.06170, 2017.

Here we introduce the *interaction network*, a model which can reason about how objects in complex systems interact, supporting dynamical predictions, as well as inferences about the abstract properties of the system. Our model takes graphs as input, performs object- and relation-centric reasoning in a way that is analogous to a simulation, and is implemented using deep neural networks. We evaluate its ability to reason about several challenging physical domains: n-body problems, rigid-body collision, and non-rigid dynamics. Excerpt from: Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics. *CoRR*, arXiv:1612.00222, 2016.
In addition to communicating by jointly editing and running code, natural language is the primary conduit for conveying programming knowledge. We are experimenting with the notion of using a special-purpose programming language for representing hierarchical plans as an executable intermediary representation between natural language and the native language supported by the IDE, possibly even directly decoding thought vectors into this intermediary language for both writing programs and producing dialog. Understanding language is more challenging.

Understanding language is all about reference and context. Words and phrases are defined by the things they refer to and the company they keep. In dialog, the context of a word or phrase might include the sentence, paragraph and document in which the word or phrase appears. More generally, the context encompasses the larger geographical, historical and conceptual context that we share with one another. We are essentially defined by a subset of this larger context that we have experienced directly or been exposed to indirectly through language and stored in memory.

Practically speaking much of what we know about programming is stored in episodic memory in the form of conversations we’ve had, books and papers we’ve read, code that we’ve written, studied and, often, modified for our use. One of the challenging problems associated with the design of the programmer’s apprentice involves how this personal episodic information is initially represented and stored, retrieved when appropriate, and adapted and applied in solving a particular programming problem. We provide a general solution to this problem that serves as the foundation for the apprentice’s personality and special expertise.
The assistant agent is designed to distinguish between three voices: the voice of the programmer, the voice of the apprentice’s automated tutor and its own voice. These three voices are associated with three separate agents. The first two are tracked and their behavior and interactions with the programmer recorded in a relatively simple but practical Theory of Mind model developed by Neil Rabinowitz and his colleagues at DeepMind. A full fledged Theory of Mind would have the ability to attribute mental states such as beliefs and intentions to all three of the agents in order to keep track of what the apprentice thinks each agent believes about their shared context and intends to do next. The Programmer’s Apprentice application requires only a very basic facility in order to keep track of who said what when.
The tutor is a reinforcement learning system responsible for administering a curriculum designed to bootstrap basic language and programming skills. At any given time, a meta-control system in concert with a reinforcement-learning-trained policy determines a curricular goal constraining the tutor's choice of specific lesson. This policy is implemented using a variant of the scheduled auxiliary control paradigm described by Riedmiller et al. Having selected a subset of lessons relevant to the current curricular goal, the meta-controller cedes control to the tutor which selects a specific lesson and a suitable plan to oversee interaction with the agent over the course of the lesson.

Most lessons will require a combination of spoken dialogue and interactive signaling that may include both the agent and the tutor pointing, highlighting, performing edits and controlling the FIDE by executing code and using developer tools like the debugger to change state, set break points and single step the interpreter. The curriculum for mastering the basic referential modes is divided into three levels of mastery in keeping with Terrence Deacon's description of Charles Sanders Peirce's (semiotic) theory of signs. The tutor will start at the most basic level, continually evaluating performance to determine when it is time to graduate to the next level or when it is appropriate to revert to an earlier level to provide additional training in order to master the less demanding modes of reference.

Relevant technical notes from related work mentioned or quoted above:

We propose a new learning paradigm scheduled auxiliary control in the context of
reinforcement learning that enables learning of complex behaviors — from scratch — in the presence of multiple sparse reward signals. To this end, the agent is equipped with a set of general auxiliary tasks, that it attempts to learn simultaneously via off-policy RL. The key idea behind our method is that active (learned) scheduling and execution of auxiliary policies allows the agent to efficiently explore its environment - enabling it to excel at sparse reward RL. Excerpt from: Martin A. Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing - solving sparse reward tasks from scratch. CoRR, arXiv:1802.10567, 2018.
Supplementary Material

Episodic Memory Formation and Recall

[Diagram showing the regions of the brain involved in episodic memory, including Posterior Cortex, Medial Temporal Lobe, and Hippocampal Formation.]

LEARNING DEEP GENERATIVE MODELS OF GRAPHS

Representing and Reasoning About Structured Programming

LEARNING DEEP GENERATIVE MODELS OF GRAPHS