
Reinforcement Learning of Implicit and Explicit Control Flow in Instructions

Ethan A. Brooks¹ Janarthanan Rajendran¹ Richard L. Lewis² Satinder Singh¹

Abstract

Learning to flexibly follow task instructions in dynamic environments poses interesting challenges for reinforcement learning agents. We focus here on the problem of learning control flow that deviates from a strict step-by-step execution of instructions—that is, control flow that may skip forward over parts of the instructions or return backward to previously completed or skipped steps. Demand for such flexible control arises in two fundamental ways: explicitly when control is specified in the instructions themselves (such as conditional branching and looping) and implicitly when stochastic environment dynamics require re-completion of instructions whose effects have been perturbed, or opportunistic skipping of instructions whose effects are already present. We formulate an attention-based architecture that meets these challenges by learning, from task reward only, to flexibly attend to and condition behavior on an internal encoding of the instructions. We test the architecture’s ability to learn both explicit and implicit control in two illustrative domains—one inspired by Minecraft and the other by StarCraft—and show that the architecture exhibits zero-shot generalization to novel instructions of length greater than those in a training set, at a performance level unmatched by three baseline recurrent architectures and one ablation architecture.

1. Introduction

An important goal in artificial intelligence is developing flexible and autonomous agents capable of accomplishing tasks that humans specify in forms that are expressive to the agent and convenient for the human user. In this work we focus on the reinforcement learning problem of following

¹ Department of Computer Science, University of Michigan ² Weinberg Institute for Cognitive Science, Departments of Psychology and Linguistics, University of Michigan. Correspondence to: Ethan Brooks <ethanbro@umich.edu>.

task instructions that require the agent to learn control flow either because the instructions themselves include explicit conditionals such as the if- and while- statements familiar from programming languages, or because the need arises implicitly when stochastic events in the environment require parts of the instructions to be redone or allow them to be skipped. In a classical programming language interpreter, the logic of control flow is fixed in advance and determined completely by the program itself. In contrast, in our interactive reinforcement-learning (RL) setting, while the reward is determined by the instructions, the control flow arises dynamically from interactions between the instructions and the stochastic dynamics of the environment.

Our primary contribution is a novel neural network architecture, the Control Flow Comprehension Architecture - Scan (CoFCA-S)¹, that successfully learns, from task reward signals alone, how to follow instructions that require subtasks to be performed in a nonlinear fashion. More specifically, we introduce two novel elements in our architecture that facilitate generalisation to novel and longer instructions during testing relative to during training: 1) an attentional mechanism that learns to maintain a pointer into the instructions and moves that pointer based on where the pointer should go next rather than how far the pointer should move, and 2) a mechanism that processes the instructions relative to the current pointer, thereby capturing a kind of coarse translation invariance present in our tasks.

We compare our method to four baselines and one ablation, and demonstrate better performance on two distinct domains, one designed to showcase explicit control-flow, and the other to showcase implicit control-flow. We argue that the improvements on these baselines are a result of our pointer architecture, the freedom of pointer movement that it affords, and the techniques that our architecture employs to transfer logic learned on shorter instructions to instructions of much greater length.

2. Related Work

This section reviews prior work on neural architectures for *instruction following*. We also present a brief overview of

¹Source code may be accessed from <https://github.com/ethanabrooks/CoFCA-S>

work on *program execution* because some of that work can handle explicit forms of control flow and because our learning architecture builds on architectures used in this line of work. However, our main focus in this paper is on methods that can learn, through trial-and-error interaction with an environment, the flow of control needed to obtain reward in the face of stochastic events in the environment both with and without explicit control flow in the instructions.

Neural architectures for instruction following. Our work is closely related to *Zero-shot Task Generalization with Multi-Task Deep Reinforcement Learning* (Oh et al., 2017) (henceforth, OLSK), which presents a novel architecture that accepts instructions in the form of a sequence of symbols denoting actions and objects whose semantics are learned with RL. The architecture maintains an attentional program pointer in an approach similar to ours. An explicit analogy mechanism supports generalizations to entirely novel action-object combinations, a kind of generalization that is not the focus of our work. The instructions in OLSK adhere to linear ordering, where tasks must be performed in exactly the same order that they occur in the instructions, but the OLSK architecture can be applied to our instructional task setting and we therefore use it as an instructive baseline. The work of Sun et al. (2020), whose domain inspired the “Minecraft” domain in our work, considers task specifications with rigid control-flow which their architecture navigates using a pre-specified parser and program interpreter. They train a classifier to evaluate control-flow predicates using supervised learning—e.g., to determine whether an if-condition passes based on the number of some resource in the environment—and an RL agent to interpret instructions based on observations of the environment and outputs of the classifier and parser. The *neural subtask graph solver* (NSGS) of Sohn et al. (2018) accepts task specifications in the form of *subtask graphs* encoding the pre-condition dependencies among subtasks and rewards along the way; optimal task execution requires finding graph traversals that maximize reward. The NSGS is trained with RL and exhibits zero-shot generalization to novel graphs. The graph specification is more expressive than our sequential instructions language in that it makes explicit precondition dependencies, but it does not deal with the non-linear control flow that is our concern here.

Like our work, the preceding publications focus on instructions comprising sequences or collections of subtasks. A much broader literature has taken up the question of single-task instructions. Yu et al. (2018a) propose a novel deep architecture for combining the instruction sentence with an high-dimensional observation. Yu et al. (2018b) extends this work to focus on generalization to instruction sentences comprising novel combinations of words (these describe new tasks, not new orderings of subtasks as in our case).

Bahdanau et al. (2018) use an adversarial framework to generate reward functions from instructions, thereby reducing the need for hand-engineering.

Another important body of literature considers human-generated natural language instructions (as opposed to the programmatically generated instructions in our domains). In general, these approaches use supervised learning or consider instructions that induce simpler policies than those required by our domain. Chaplot et al. (2018) propose a multiplication-based mechanism for combing instructions with observations, which facilitates following of natural-language instructions in 3d domains. Misra et al. (2018) develop a supervised-learning approach for mapping natural language statements to sequences of actions by marginalizing across possible goal locations implied by the instruction. Fried et al. (2018) consider a similar problem, but introduce a “speaker” network that infers the probability of instructions given action trajectories. They use the “speaker” to augment the dataset with synthetic instructions and to rank candidate action-sequences produced by a “follower” network. Hill et al. (2020) use embeddings from a pre-trained language model to encode a single-sentence instruction. The result is fed into an RL agent for execution in a simulated 3d-domain.

Neural architectures for program execution. Graves et al. (2014) was an important early contribution to the problem of program execution using end-to-end differentiable architectures. Our work builds on the seminal attention-based architecture from this paper but as we detail below we needed a more sophisticated attention mechanism to allow for learning of the kinds of nonlinear control flow required in our reinforcement learning tasks. Reed & De Freitas (2015) develops an architecture that learns to execute complex programs via a supervised training regime that paired programs and execution traces. This work does not consider the problem of following instructions with many parts whose ordering must be learned from trial-and-error experience. Bošnjak et al. (2017) extend this work with supervised learning of tasks specified using the Forth programming language. Bieber et al. (2020) develop an architecture that predicts the output of a program.

Our work thus exists at the intersection of work that learns to follow instructions comprising multiple subtasks—(Oh et al., 2017; Sun et al., 2020; Sohn et al., 2018)—and work that focuses on the problem of non-linear task specifications—(Bošnjak et al., 2017; Bieber et al., 2020). Our work departs from prior work in several ways: 1) unlike the work on program execution/interpretation, in our setting it is not possible to determine the sequence of subtasks to be performed solely by looking at the instructions, rather it has to be learned through trial and error interaction with a stochastic environment, and 2) we are in the RL setting

in which the only feedback to the agent is in the form of a terminal (and therefore delayed) reward as opposed to much of the prior work that has focused on variations within the supervised learning setting.

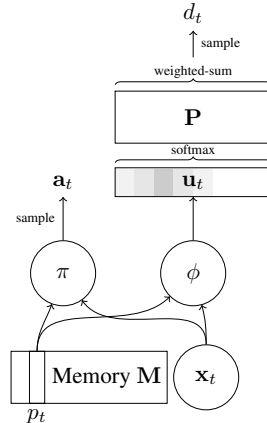
3. CoFCA-S: The Control Flow Comprehension Architecture - Scan

In our setting, it is not possible to map the instructions and current observation to the desired action because the current observation may not record all the subtasks that have already been performed and those that have been undone by stochastic events. One way to deal with this would be to learn a mapping from the instruction and the history of interactions to the action. However this can be challenging in complex settings with long instructions where each line requires the agent to take several actions in the environment. The underlying assumption in our architecture is that much of this challenge can be handled by maintaining a pointer to a line/subtask in the instructions, and one can learn a mapping from the line/subtask pointed to by the pointer and the current observation to a desired action. Of course, this requires a procedure for updating the pointer after taking the action and receiving an observation.

One way to handle pointer movement would be to learn a distribution over backward and forward pointer movements during training. The challenge is that one can only learn about movement distances encountered during training. One of our main contributions is to learn *where* we want to move the pointer instead of *how far* we want to move the pointer. The ‘‘Scan’’ mechanism we introduce to do this computes the probability of a pointer movement as a function of the features of the lines around the line to which we are considering a move. This allows pointer movement to be free of distances and allows generalization to longer instructions requiring longer distance pointer movements during testing than seen during training.

Furthermore we introduce a mechanism in our architecture to exploit the following invariance property present in our tasks: chunks of the instructions, e.g., a while loop, require the agent to take the same actions no matter where that chunk is in the instructions. We accomplish this by always processing the embeddings of the instructions from the pointer’s position in our architecture. This simple mechanism captures the chunk-translation-invariance property and facilitates transfer of learned behavior from one position in the instructions to other positions in the instructions. This contributes to our results on generalising to longer instructions during testing.

Our architecture comprises the following primary components. \mathbf{M} is an encoding of the instructions. p_t is the integer pointer into \mathbf{M} . π , implemented as a neural network,



CoFCA-S Algorithm

-
- Require:** pointer p_t
Require: observation \mathbf{x}_t
Require: instruction \mathbf{I}
- 1: $\mathbf{M} \leftarrow \text{bag-of-words}_{\theta}(\mathbf{I})$
 - 2: $\mathbf{H}_t \leftarrow \text{BI-GRU}([\mathbf{M}, \mathbf{x}_t])$
 {BI-GRU starts at line p_t }
 - 3: Generate \mathbf{P} (equation 1)
 - 4: $\mathbf{a}_t \sim \pi(\mathbf{x}_t, \mathbf{M}_{p_t})$
 - 5: $\mathbf{u}_t \leftarrow \phi(\mathbf{x}_t, \mathbf{M}_{p_t}, \mathbf{a}_t)$
 - 6: $\tilde{\mathbf{u}}_t \leftarrow \text{softmax}(\mathbf{u}_t)$
 - 7: $d_t \sim \text{Cat}(\mathbf{P}\tilde{\mathbf{u}}_t)$
 - 8: $c_t \sim \psi(\mathbf{x}_t, \mathbf{M}_{p_t}, \mathbf{a}_t)$
 - 9: $p_{t+1} \leftarrow p_t + c_t d_t$
-

Figure 1. (Left) Depicts the flow of information every time step from memory \mathbf{M} , pointer p_t , and observation \mathbf{x}_t to actions \mathbf{a}_t and pointer movements d_t . (Right) Pointer update pseudocode.

combines information from \mathbf{M} and the observation, \mathbf{x}_t to produce actions. \mathbf{P} is a collection of possible pointer movement distributions. ϕ , implemented as a neural network, combines information from \mathbf{M} and \mathbf{x}_t to choose among these distributions. Fig. 1 identifies these components and their relationships. We now describe them in detail.

3.1. Instruction preprocessing.

First, our architecture uses a lookup table to embed each line of the instruction (line 1 of the pseudocode in Fig. 1). If an instruction line comprises several symbols, we embed each symbol separately and sum them. The result is $\mathbf{M} \in \mathbb{R}^{N \times E}$, where N is the number of lines in the instructions and E the size of the embedding. Next, we concatenate \mathbf{x}_t to each line of \mathbf{M} and pass the result through a bidirectional Gated Recurrent Unit (GRU) (Chung et al., 2014), a variant of the bidirectional Recurrent Neural Network (Graves et al., 2013). Importantly, we start the forward pass at line p_t (recall that p_t is the pointer into memory), instead of the first line of the instruction. This simple change enables the architecture to exploit the chunk-translation-invariance property, by allowing learning from earlier lines of the instruction to be reused on later lines. The GRU outputs an L dimensional vector for each line, with higher weights increasing the probability of moving forward to that line (more explanation in the next paragraph). We do the same with a backward GRU starting at line $p_t - 1$, this time for backward movement. Combining the two we get a weight matrix $\mathbf{H}_t \in \mathbb{R}^{2N \times L}$. L encodes the number of possible pointer distributions which we subsequently choose among using information derived from the observation. We pass \mathbf{H}_t through a sigmoid function squashing it between 0 and 1. These steps correspond to line 2 of the pseudocode.

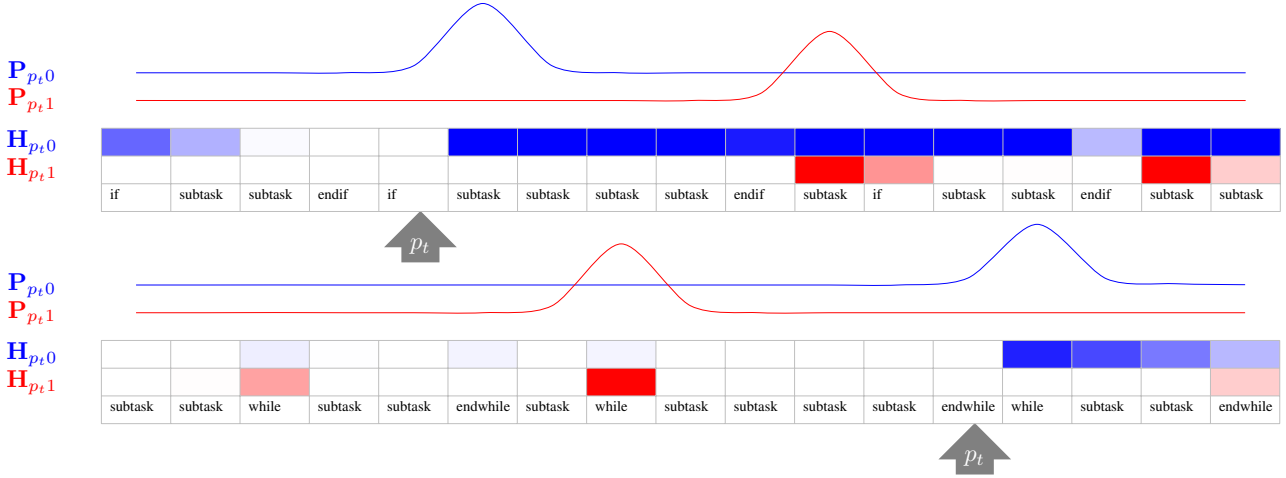


Figure 2. (Upper) Here, $p_t = 4$. The agent will choose the distribution $\mathbf{P}_{p_t,0}$ if the condition succeeds and $\mathbf{P}_{p_t,1}$ if it fails. To generate these distributions $\mathbf{H}_{p_t,0}$ has “flagged” all lines except those immediately preceding p_t . $\mathbf{H}_{p_t,1}$ has flagged lines following *endif*. (Lower) Here, $p_t = 12$. The agent will use the $\mathbf{P}_{p_t,1}$ distribution to return the pointer to the start of the while loop in order to inspect the *while* condition, whose evaluation will determine the next distribution that the agent chooses (as in the upper figure).

3.2. The Scan Mechanism.

This mechanism transforms \mathbf{H}_t into a collection of distributions over pointer movements. To simplify our explanation of this mechanism, we first assume that $L = 1$. In this case, the distribution we use to generate pointer movements is equivalent to a geometric distribution generated by the following process. We scan through each line in the order $p_t + 1, p_t - 1, p_t + 2, p_t - 2, \dots$. At each line we flip a coin, with heads-probability determined by the sigmoid output described in the preceding paragraph. We stop at the first line where the coin comes up heads. When $L > 1$, we repeat this process L times. The result is the matrix

$$\mathbf{P}_{ij} = \sigma(\mathbf{H}_{ij}) \prod_{k \in \{1, -1, 2, -2, \dots, N, -N\}, k \neq i} (1 - \sigma(\mathbf{H}_{(N+k)j})) \quad (1)$$

Like \mathbf{H}_t , \mathbf{P} is in $\mathbb{R}^{2N \times L}$. We convert \mathbf{P} into a single distribution using \mathbf{u}_t , weighting over the different distributions in \mathbf{P} , as follows:

$$\mathbf{u}_t = \phi(\mathbf{x}_t, \mathbf{M}_{p_t}, \text{Emb}(\mathbf{a}_t)) \quad (2)$$

$$d_t \sim \sum_{i=1}^L \text{softmax}(\mathbf{u}_t)_i \mathbf{P}_{(\cdot)i} \quad (3)$$

where ϕ is a neural network (details in §3.5) and d_t is a pointer movement in the form of a delta to add to p_t . These steps correspond to lines 5 through 7 of the pseudocode.

The motivation for the distribution expressed in equation 1 is that it allows the size of pointer movements to depend on features at the destination line, not on the size of the jump. This is critical to enable the agent to perform pointer movements larger than those performed during training.

E.g., if \mathbf{H}_0 corresponds to an *if* line, the GRU might learn to flag subsequent *endif* lines by assigning large values to \mathbf{H}_{ij} for all indices i corresponding to subsequent *endif* lines. As long as all values of \mathbf{H}_{kj} are nearly 0 for $|k| < |i|$, the pointer will be able to move to line i , even if i is much larger than any jump that the agent has performed during training.

3.3. Gating of pointer movement.

While performing an individual subtask, the agent should not move the instruction memory pointer—it should learn to wait for the subtask to be completed before advancing. The agent learns to *gate* changes to the memory pointer to accomplish this waiting. The gate is a binary value c_t sampled from a learned distribution $\psi(\mathbf{x}_t, \mathbf{M}_{p_t}, \text{Emb}(\mathbf{a}_t))$. ψ is a feed-forward neural network as detailed in §3.5. p_t only changes position when the gate’s value is 1: $p_{t+1} = p_t + c_t d_t$. This corresponds to lines 8 and 9 of the pseudocode.

3.4. Action sampling mechanism.

Actions depend on information from the instruction and from the environment. In one of our domains, actions also depend on the history of previous actions. We derive information from the instruction by using the pointer to index into the encoded representation of the instruction, \mathbf{M}_{p_t} . We derive information about the environment from an encoding of the current observation \mathbf{x}_t (see §3.5). Where relevant, we encode the action history using a GRU: $\mathbf{h}_t = \text{GRU}(\mathbf{a}_t, \mathbf{h}_{t-1})$. Thus the policy is $\pi(\mathbf{x}_t, \mathbf{M}_{p_t})$ or $\pi(\mathbf{x}_t, \mathbf{M}_{p_t}, \mathbf{h}_t)$.

3.5. Network architectures.

Both ϕ , the network responsible for producing \mathbf{u}_t , and π use a neural “torso” with shared weights, which is special-

ized to handle the distinct observation spaces of each of our domains (§4.2 and §4.3). The torso uses a convolutional neural network for 3d components of the observation, a lookup table of neural embeddings for integer components, and a linear projection for all other components. The torso concatenates the results of these operations, applies a Rectified Nonlinear Unit (Nair & Hinton, 2010), and passes the result to the heads corresponding to π and ϕ . These heads are implemented using linear projections followed by a softmax which transforms the outputs into a probability distribution for sampling actions (in the case of π) and for choosing columns of \mathbf{P} (in the case of ϕ).

3.6. Failure buffer.

During training the agent may learn a suboptimal policy which works most of the time but fails for a small subset of instructions and episode starting conditions. In order to encourage the agent to learn a policy that is robust to these challenges, we modify the distribution of training episodes to increase the frequency of difficult episodes. We accomplish this by saving the random seed used to generate unsuccessful episodes to a “failure buffer.” We also maintain a moving average of the agent’s success-rate and, each episode, with probability proportional to this success-rate, we sample the seed from the failure buffer to retry a previously unsuccessful episode.

3.7. Training details.

We train the agent using Proximal Policy Optimization algorithm (Schulman et al., 2017), a state-of-the-art on-policy RL algorithm. The learning objective is

$$L(\theta) = \mathbb{E}[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)] + \alpha\mathcal{H} \quad (4)$$

$$r_t(\theta) = \frac{\Pr(\mathbf{a}_t, d_t, c_t | \mathbf{x}_t, \mathbf{M}_{p_t}, \mathbf{a}_t, \theta)}{\Pr(\mathbf{a}_t, d_t, c_t | \mathbf{x}_t, \mathbf{M}_{p_t}, \mathbf{a}_t, \theta_{\text{old}})} \quad (5)$$

Here A_t denotes the advantage on time step t , θ denotes network parameters, θ_{old} represents the pre-update parameters, α refers to an entropy coefficient, \mathcal{H} refers to the entropy over the probability distribution in the numerator of (5), and \Pr refers to the joint probability of the action choice, pointer movement and gate, calculated as the product of their individual probabilities. For tuning hyper-parameters of all algorithms, we searched for good common values for hidden size, kernel size, and stride, for both convolutions and for all hidden sizes used by neural networks. We also tuned entropy coefficient values (used to encourage exploration), number of distributions L in \mathbf{P} , and learning rate.

4. Experiments

In this section we present results from three generalization experiments in two instruction-following domains, in which agents are trained on short instructions and evaluated on longer instructions or instructions containing unseen combinations of explicit control-flow blocks. The first domain is inspired by the *StarCraft* video game and is designed to impose challenges in learning *implicit* control flow. The second domain is inspired by *Minecraft* and is designed to impose challenges in learning *explicit* control flow. We compare our architecture to three baselines, one using recurrence to maintain a memory of progress through the instructions, another using the OLSK (Oh et al., 2017) architecture described above, and a third using a modified version of OLSK with extended pointer-movement range. We also compare our architecture to an ablation, CoFCA which removes the Scan mechanism from CoFCA-S. In all results, error bands and bars indicate standard error across 4 distinct random seeds. To anticipate our main results: we find in both domains that CoFCA-S outperforms the baselines as well as CoFCA in generalization, especially as instruction length increases in both domains.

4.1. Baselines

The **Unstructured Memory** (UM) baseline uses the recurrent state of a recurrent neural network to track its progress through the instruction. Like CoFCA-S, this algorithm runs a bidirectional GRU length-wise across the instructions. However, instead of retaining all of the outputs of the GRU and maintaining a pointer, it feeds the concatenated last outputs of this bidirectional GRU into a second GRU, along with \mathbf{x}_t and the embedded action \mathbf{a}_t . Note that the length-wise GRU encoding of the instruction is necessary to facilitate generalization from shorter to longer instructions, which would not be possible if simpler methods like concatenation were used instead. Thus the first GRU is responsible for encoding the variable-length instruction, whereas the second is responsible for preserving state information across time steps. The baseline must use the recurrent hidden state \mathbf{h}_t of the second GRU to perform the functions that \mathbf{M} and p_t perform in CoFCA-S, tracking the agent’s progress through the instructions. The policy π maps \mathbf{h}_t and the observation \mathbf{x}_t to a distribution from which the architectures samples the \mathbf{a}_t .

The **OLSK** baseline reproduces the algorithm described in OLSK (Oh et al., 2017). Each time step, ϕ maps \mathbf{x}_t , \mathbf{M}_t , and a hidden state \mathbf{h}_t to a distribution over discrete pointer movements in $\{-1, 0, +1\}$ and a new hidden state \mathbf{h}_{t+1} (per the hard-attention scheme described in (Oh et al., 2017)). Thus, OLSK can only move the pointer one step forward or one step backward and so has to do this repeatedly without acting in the world to move the pointer many steps, in con-

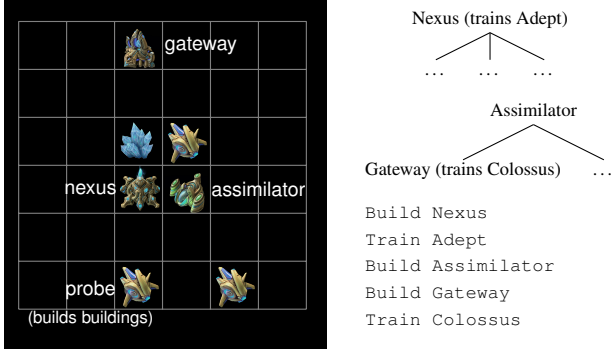


Figure 3. Example of StarCraft-inspired environment and instructions. The instructions indicate that Nexus is the building which trains the Adept unit, the Assimilator is a prerequisite for the Gateway, and the Gateway trains the Colossus. The randomly generated build-tree (upper right) is the basis for the instructions, but the agent only sees the instructions, not the tree, and the instructions only contain information relevant to the production of the required units (the Adept and the Colossus, in this example).

trast to CoFCA-S, which can move the pointer between any two lines in the instruction in one step.

The **extended-range OLSK** (OLSK-E) extends the range of OLSK’s pointer movement to facilitate pointer movements in $\{-N, \dots, +N\}$, where N is the length of the instruction. Note that OLSK does not use the bidirectional GRU to preprocess the instruction and it does not take advantage of the “Scan” mechanism described in §3.2. Note that the absence of this preprocessing mechanism limits the ability of both OLSK and extended-range OLSK to operate in a context-aware manner.

The **CoFCA** baseline is an ablation of CoFCA-S. As the omission of “-S” suggests, CoFCA ablates the Scan mechanism described by equation 1. Recall that CoFCA-S passes the embedded instructions M through a bidirectional GRU. CoFCA retains only the final output of this GRU and uses a single-layer neural network followed by a softmax layer to project the output to a distribution over forward/backward pointer movements, up to the maximum length instruction to be evaluated. Note that while CoFCA can in principle move between any two lines, even in the longer evaluation instructions, it will only have the opportunity during training to perform jumps equal to or less than the size of the instruction.

4.2. StarCraft-inspired domain: implicit control flow

StarCraft is a real-time strategy game in which players construct buildings to produce units and research technologies, and use those units to destroy an opponent. In our StarCraft-inspired domain the agent assumes the role of a support player that receives an order for units from an allied player;

these orders are the instructions given to the agent. To satisfy the order for a type of unit, the agent must construct the buildings that produce that type of unit and then train the units. The agent cannot construct a building until a prerequisite building has been constructed. Each building depends on at most one building, though each building may serve as a prerequisite for several others, forming a *build tree* of dependencies. Figure 3 provides an example.

Two aspects of the environment dynamics interact to introduce implicit control flow learning challenges. The first is that enemy attacks on buildings and ambushes on units happen stochastically, requiring buildings to be reconstructed and units to be retrained. The second is that the type of buildings that produce each unit type as well as the build-tree dependencies are randomly generated for each episode, and this information is encoded in the instructions, requiring the agent to learn to extract these relationships from the instructions instead of learning them from experience. Also, when the enemy ambushes units or attacks buildings, the agent must learn to respond to these events and re-execute just those parts of the instructions whose effects were undone.

Episodes play out in a 6×6 gridworld. The agent begins each episode with three Probes (units for constructing buildings) and a random endowment of pre-constructed buildings. If these buildings correspond to buildings required by the instruction, the agent may opportunistically skip the corresponding lines of instruction. The agent’s observation includes a top-down view of the gridworld with channels devoted to buildings, units, and terrain.

Instructions. Each line in the instruction is encoded by an integer indicating either a building type or a unit type. A building followed by a building indicates that the first is a prerequisite of the second. A building followed by a unit indicates that the building must be used to build the unit.

Attacks and ambushes. Enemy attacks and ambushes occur each time-step with 10% probability. An attack wipes out a random subset of the existing buildings. An ambush destroys one of the units the agent has produced. If the instructions indicated that the destroyed unit was required, the agent must produce the unit again. The agent may also need to rebuild the buildings required to construct that unit if the building was previously destroyed in an enemy attack, requiring the agent to consult earlier sections of the instructions. Messages to the agent notifying it about unit ambushes assume the form of a single integer, which identifies the type of unit ambushed.

Reward and termination. The agent receives a reward of 0 on each timestep and 1 for completing the instructions (producing all required units). The episode terminates when the instructions are completed or when a time limit runs out.

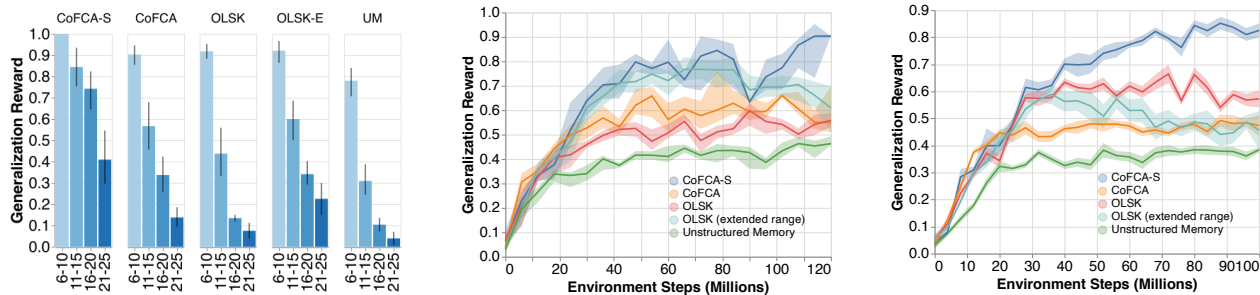


Figure 4. Results for experiments in the StarCraft-inspired domain (§4.2). Y-axis represents cumulative reward on evaluation episodes. (Left) Generalization to longer instructions (§4.2.1), binned by instruction length, aggregated from the last 5 million steps. (Middle) Learning curves for generalization to longer instruction lengths. (Right) Learning curves for generalization to longer instruction lengths and deeper build trees (§4.2.2).

The time limit is $30\times$ the length of the instructions.

Modifications to the architectures to handle the large action space. There are 1512 build commands (3 Probes \times 36 coordinates \times 14 buildings) and 108 go-to commands (3 Probes \times 36 coordinates). Buildings are chosen by coordinate and therefore there are as many as 576 train-unit commands (36 coordinates \times 16 unit types). To handle this action space we adopt an autoregressive policy (Metz et al., 2017), which allows the agent to select a variable-length *sequence* of actions between environment interactions. The agent first chooses a Probe or a coordinate. Choosing a coordinate containing a building selects that building and allows the agent to produce any unit that the building is capable of producing. Choosing a Probe allows the agent to construct buildings using that Probe or to move the Probe to a coordinate. Constructing a building requires choosing a building then an unoccupied coordinate. Invalid choices (e.g. telling a Probe to construct a building for which the prerequisite has not been constructed) result in a no-op. This dependence on the history of actions motivates the addition of the action GRU described in §3.4.

4.2.1. GENERALIZATION TO LONGER INSTRUCTIONS UNDER IMPLICIT CONTROL FLOW DEMANDS

Training and evaluation. The aim of this experiment is to test the ability of the agents to learn implicit control flow strategies from short instructions and generalize to longer ones. All agents were trained in the StarCraft-inspired domain on randomly sampled instructions of length 1 to 5. Every million frames, we evaluated agent performance on 150 complete episodes with instructions of length 6 to 25. In this way we tested the ability of the agent to learn strategies for re-executing parts of the instructions affected by disruptive events, in a way that generalizes to longer instructions.

Results. Figure 4 shows the performance of all five architectures on the evaluation instructions, with performance

binned by instructions length. CoFCA-S outperforms the baselines on all instructions lengths. We conjecture that Unstructured Memory has difficulty tracking its place in non-sequential control flow, and OLSK has difficulty performing large pointer movements through a sequence of single-steps. We conjecture CoFCA-S performs best because it can learn a logic of pointer movement that is *independent* of the size of the movement. For example, it might learn to scan backward to the first of a series of buildings destroyed by the enemy. Such logic can transfer from the short training instructions to longer evaluation instructions.

4.2.2. GENERALIZATION TO LONGER INSTRUCTIONS AND DEEPER BUILD TREES

Training and Evaluation. This experiment reproduces the setup of the previous (§4.2.1) but during training, it restricts the depth of build-trees to three. Though the agent is free to perform larger jumps, the agent need only perform jumps of size three or less to complete these tasks. Meanwhile, we evaluate the agents on trees with unrestricted depth (up to 16, if the tree forms a chain). Thus these experiments increases the pressure on agents to perform longer jumps during evaluation than during training.

Results. This experiment highlights the advantages of the Scan mechanism which is designed to facilitate generalization from shorter jumps during training to longer jumps in evaluation. OLSK’s second-best performance makes sense given that it does not perform jumps at all, and is therefore less susceptible to overfitting on shorter jumps learned during training.

4.3. Minecraft-inspired domain: explicit control flow

In the prior StarCraft-inspired domain the instructions themselves imposed a simple linear execution order, and the interesting non-linear control flow challenges arose from stochastic environment dynamics. In this section we explore



Figure 5. Example of instructions and environment state in the Minecraft-inspired domain. Correct execution begins at the first line with evaluation of the *while* condition, which evaluates to true because there is more *iron* than *gold*. To perform the *mine iron* subtask, the agent must navigate to an iron resource (the orange arrow) and perform a *mine* action. This removes the iron resource from the environment but the *while* condition still evaluates to true because there is still more iron than gold. After mining a second iron resource (second orange arrow), the numbers of iron and gold resources are equal and the *while* condition evaluates to false. At this point, the number of *merchants* in the environment (depicted as llamas) exceeds the number of iron resources and the *if* condition evaluates to true. The agent must then execute *inspect iron* and *sell gold* but must skip the *mine wood* subtask. To execute *sell gold* the agent must navigate to a gold resource and mine it (first violet arrow), navigate to a merchant, and then sell the gold (second violet arrow), terminating the episode with a reward of 1.

non-linear control flow that is imposed by the instructions themselves through explicit control flow elements.

Figure 5 provides an example of our domain (inspired by Sun et al. (2020) and the Minecraft video game). The agent navigates a 6×6 gridworld in which resources, terrain, and merchants spawn randomly each episode. In each episode the agent is given new instructions containing subtasks interspersed with control-flow statements. The agent’s goal is to perform these subtasks in the order specified by the control flow. The agent’s observation includes a top-down view of this grid with channels encoding the presence of objects in the gridworld. The resources are *gold*, *iron*, and *wood*. The terrain includes impassible *walls* and *water* which can be bridged only if the agent possesses wood in its inventory.

Instructions. At the start of each episode, the agent receives instructions comprised of a list of lines, each corresponding to a subtask or a control-flow keyword: *if*, *else*, *endif*, *while*, or *endwhile*. These have the procedural semantics familiar from programming languages. (The indentation in Figure 5 is a visual aid not available to the agent.) Each line in the instructions consists of three symbols: the first encodes the control-flow keyword or indicates that the line is a subtask; the second encodes an action; the third encodes either the target resource for the action or the condition for an *if* or *while* line that requires resource comparisons;

e.g. *more iron than gold* is true if the amount of iron in the environment exceeds the amount of gold. While the agent observes the entire instructions on each time-step, nothing in the observation indicates the agent’s progress through the instructions.

Actions and inventory. The agent acts by issuing verb-noun commands to a worker. The possible verbs are *mine*, *sell*, and *inspect*, while the possible nouns correspond to resources: *iron*, *gold*, and *wood*. The action-space of the agent is the cross-product of nouns and verbs. The movements of the worker in response to these commands are determined by a pre-trained RL agent. In response to a *mine resource* command, the worker navigates to the resource and executes a *mine* action, removing it from the map and adding it to the inventory. For a *sell resource* command, the worker executes a *sell* interaction on a merchant grid, which decrements the resource inventory. If the worker does not already possess the resource, it must first collect it using the *mine* action. To complete an *inspect resource* command, the worker must execute an *inspect* action on a resource grid (this does not change or remove the resource). If the agent’s inventory contains wood, the agent may move into a water grid, decrementing the wood in its inventory and removing the water grid (building a bridge). The agent observes the inventory as a list of integers encoding the quantity of each resource type.

Reward and termination. The agent receives a reward of 1 upon completion of all subtasks in the instructions, in the order specified by control flow. All other time steps provide a reward of 0. An episode terminates when the instructions are completed or if the worker performs a *mine* or *sell* action out of the order specified by the instructions, or if the agent exceeds a time-limit equal to $30 \times$ the length of the instructions.

4.3.1. GENERALIZATION TO LONGER INSTRUCTIONS WITH EXPLICIT CONTROL FLOW

Training and evaluation. The aim of this experiment is to test the agents’ ability to learn how to follow explicit control flow from short instructions and generalize to longer instructions. We trained the architectures on instructions of length 1 to 10 in our Minecraft-inspired domain, randomly sampled from a generative grammar (see Appendix). Every 100 gradient steps, we evaluated the performance of the agent for 500 time steps on instructions of length 11 to 50, also sampled from the same grammar.

Results. Figure 6 (left and middle) provides the evaluation results. CoFCA-S and CoFCA outperform the three baselines, all of which fail to generalize entirely. In the case of OLSK this is again likely because the pointer cannot

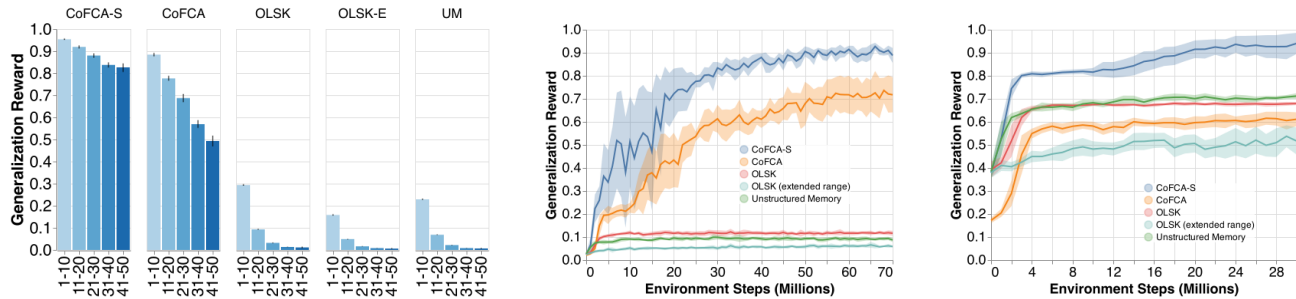


Figure 6. Results for experiments in the Minecraft-inspired domain. Y-axes represents cumulative reward on evaluation episodes. (Left) Experiment 2: Generalization to longer **instructions length**, binned by instruction length (Middle) Experiment 2: Learning curves for generalization to **longer instructions**. (Right) Experiment 3: Learning curves for generalization to instructions with **novel compositions of control-flow**.

make individual movements greater than ± 1 , and instead must make a series of these small movements. Conversely OLSK-E is capable of making larger movements, but lacks the bidirectional GRU preprocessing step that would give it access to information about lines surrounding the p_i^{th} and enable it to make these larger movements judiciously. Instead, OLSK-E must derive this kind of information from its memory of previously visited lines. Inspecting the trajectories of OLSK-E, we observe that it ultimately gives up on learning to manipulate the pointer, ignoring the instruction and instead learning a prior over the subtasks. There is also a smaller but still significant gap between the generalization performance of CoFCA-S and CoFCA. In Figure 6 (left) we break down the performance of all the agents after training for 70 million time steps as a function of the length of the instructions. As expected each agent’s performance degrades with increasing length of instructions, though for both CoFCA and CoFCA-S the drop in generalization performance from the shortest generalization lengths 10-20 to the longest generalization lengths of 40-50 is much smaller than for the baselines.

4.3.2. GENERALIZATION TO NOVEL COMPOSITIONS OF EXPLICIT CONTROL FLOW

Training and evaluation. To test generalization to novel control flow compositions, we trained the agents in the MineCraft-inspired domain on instructions of length up to 10, excluding those that contain more than one type of control flow (though possibly more than one instance of the same type of control flow). Every 100 gradient steps, we evaluated the performance of the agent for 500 time steps on instructions of length up to 10, but *only* those that contained at least two different types of control flow in them. (The maximum instruction length for testing and training was the same so as not to confound the effects with length.)

Results. All architectures do better in this experiment because the instructions are shorter, but again CoFCA-S domi-

nates. We attribute the weak performance of CoFCA to its dependence on the bidirectional GRU without the benefit of the Scan mechanism. We conjecture that a recurrent network that has only ever trained on instructions of one type is likely to generate noisy outputs when first encountering instructions with multiple types of control flows — e.g., seeing an *if* line following a *while* line. While the CoFCA-S architecture also depends on recurrence, the Scan mechanism ensures that the potentially noisy outputs of the GRU outside any given control-flow block are mostly ignored.

5. Conclusion

This work contributes a neural network architecture with a novel attentional mechanism that moves pointers based on where the pointer should move next rather than how much it should move and also processes instructions using the current pointer position. As a result, our architecture allows RL agents to learn to follow instructions with implicit and explicit control-flow as well as to generalize better to longer and novel instructions. Our empirical work demonstrated this benefit in two domains.

Acknowledgements

This work was made possible by the support of the Lifelong Learning Machines (L2M) grant from the Defense Advanced Research Projects Agency. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the sponsors.

References

Bahdanau, D., Hill, F., Leike, J., Hughes, E., Hosseini, A., Kohli, P., and Grefenstette, E. Learning to understand goal specifications by modelling reward. *arXiv preprint arXiv:1806.01946*, 2018.

- Bieber, D., Sutton, C., Larochelle, H., and Tarlow, D. Learning to execute programs with instruction pointer attention graph neural networks. *arXiv preprint arXiv:2010.12621*, 2020.
- Bošnjak, M., Rocktäschel, T., Naradowsky, J., and Riedel, S. Programming with a differentiable forth interpreter. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 547–556. JMLR. org, 2017.
- Chaplot, D. S., Sathyendra, K. M., Pasumathi, R. K., Rajagopal, D., and Salakhutdinov, R. Gated-attention architectures for task-oriented language grounding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.
- Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.
- Fried, D., Hu, R., Cirik, V., Rohrbach, A., Andreas, J., Morency, L.-P., Berg-Kirkpatrick, T., Saenko, K., Klein, D., and Darrell, T. Speaker-follower models for vision-and-language navigation. *arXiv preprint arXiv:1806.02724*, 2018.
- Graves, A., Jaitly, N., and Mohamed, A.-r. Hybrid speech recognition with deep bidirectional lstm. In *2013 IEEE workshop on automatic speech recognition and understanding*, pp. 273–278. IEEE, 2013.
- Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.
- Hill, F., Mokra, S., Wong, N., and Harley, T. Human instruction-following with deep reinforcement learning via transfer-learning from text. *arXiv preprint arXiv:2005.09382*, 2020.
- Metz, L., Ibarz, J., Jaitly, N., and Davidson, J. Discrete sequential prediction of continuous actions for deep rl. *arXiv preprint arXiv:1705.05035*, 2017.
- Misra, D., Bennett, A., Blukis, V., Niklasson, E., Shatkhin, M., and Artzi, Y. Mapping instructions to actions in 3d environments with visual goal prediction. *arXiv preprint arXiv:1809.00786*, 2018.
- Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.
- Oh, J., Singh, S., Lee, H., and Kohli, P. Zero-shot task generalization with multi-task deep reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 2661–2670. JMLR. org, 2017.
- Reed, S. and De Freitas, N. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- Sohn, S., Oh, J., and Lee, H. Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. In *Advances in Neural Information Processing Systems*, pp. 7156–7166, 2018.
- Sun, S.-H., Wu, T.-L., and Lim, J. J. Program guided agent. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=BkxUvnEYDH>.
- Yu, H., Lian, X., Zhang, H., and Xu, W. Guided feature transformation (gft): A neural language grounding module for embodied agents. In *Conference on Robot Learning*, pp. 81–98. PMLR, 2018a.
- Yu, H., Zhang, H., and Xu, W. Interactive grounded language acquisition and generalization in a 2d world. *arXiv preprint arXiv:1802.01433*, 2018b.