ACTFORMER

A Novel Neural Architecture for Learning and Predicting Actions

Beyond Next-Token Prediction to True Computational Learning

Architecture Design Document

Version 1.0

Abhishek Shah & Z.ai Research

2025

Abstract

We introduce Actformer, a novel neural network architecture that fundamentally reimagines how machines learn computation. Unlike Transformers, which predict next tokens through attention mechanisms without truly learning underlying operations, Actformers learn and predict actions - discrete computational steps that transform state. This paradigm shift enables the model to genuinely learn algorithms rather than pattern-matching surface forms. We present the complete architectural specification, including the Action Space design, Working Memory system, Action Execution Engine, and Differentiable Operation primitives. Our approach draws from and extends key insights in Neural Turing Machines, Differentiable Neural Computers, Neural Program Synthesis, and Neurosymbolic AI, while addressing their scalability limitations. We demonstrate how Actformers can learn arithmetic operations like addition and multiplication from small examples and generalize perfectly to arbitrarily large numbers - a feat that remains fundamentally challenging for current Transformer architectures. This document serves as a comprehensive technical specification for implementing Actformer models across various scales and domains.

Table of Contents

1. Introduction and Motivation

The Transformer architecture has revolutionized artificial intelligence, achieving remarkable success across natural language processing, computer vision, and multimodal understanding. However, despite these impressive achievements, Transformers harbor a fundamental limitation that constrains their path toward artificial general intelligence: they do not learn to compute; they learn to pattern-match. This distinction, while subtle in everyday applications, becomes glaringly obvious when we examine tasks requiring genuine algorithmic reasoning.

1.1 The Fundamental Problem with Transformers

Consider a simple task that any elementary school student can perform: adding two 20-digit numbers. A student who has learned the addition algorithm can correctly sum any pair of numbers, regardless of their size, because they have internalized the underlying computational procedure. In stark contrast, a Transformer model, even one with hundreds of billions of parameters, struggles mightily with this task. Research has consistently demonstrated that Transformers trained on arithmetic operations fail to generalize beyond the numerical ranges seen during training. When asked to add numbers with more digits than encountered in their training data, their accuracy precipitously declines.

This limitation stems from the very mechanism that makes Transformers powerful: attention-based next-token prediction. When a Transformer 'predicts' the sum of two numbers, it is not performing addition in any meaningful sense. Rather, it is pattern-matching against its training distribution, retrieving statistically probable continuations. The model has learned correlations between digit patterns and output sequences, but it has not learned the algorithm of addition itself. This is why Transformers can appear to perform arithmetic on numbers similar to those in their training set while failing catastrophically on out-of-distribution inputs.

The implications of this limitation extend far beyond arithmetic. If Transformers cannot truly learn simple algorithms like addition, how can we expect them to master the complex algorithmic reasoning required for scientific discovery, mathematical proof, software engineering, or strategic planning? The pattern-matching paradigm, while extraordinarily useful for many tasks, fundamentally constrains the type of reasoning that current AI systems can achieve.

1.2 Our Core Insight: Learning Actions, Not Tokens

Actformer emerges from a fundamental reconceptualization of what neural networks should learn. Rather than predicting the next token in a sequence, Actformers predict the next action in a computational process. This shift from token prediction to action prediction changes everything about how the model learns and generalizes.

An action, in our framework, is a discrete computational operation that transforms the current state of a problem. When adding two numbers, the relevant actions include: read digit from position i, retrieve carry bit, compute sum, determine new carry, write result digit, and advance to next position. By learning to predict these actions rather than output tokens, the model internalizes the actual algorithm. Crucially, the actions learned on small numbers transfer perfectly to numbers of any size because the underlying algorithm is identical.

This approach draws inspiration from several research threads while synthesizing them into a novel architecture. Neural Turing Machines (Graves et al., 2014) demonstrated that neural networks could learn to use external memory through differentiable operations. Differentiable Neural Computers (Graves et al., 2016) extended this with more sophisticated memory addressing. Neural Program Synthesis has shown that networks can learn to generate programs. Chain-of-Thought prompting revealed that decomposing problems into intermediate steps dramatically improves reasoning. Actformer builds upon all these insights while addressing their key limitations, particularly around scalability and integration with modern deep learning infrastructure.

2. Background and Related Work

2.1 Neural Turing Machines and Differentiable Neural Computers

Neural Turing Machines (NTMs), introduced by Graves et al. in 2014, represented a paradigm shift in how we think about neural network architectures. Unlike conventional networks with fixed computational graphs, NTMs incorporate an external memory matrix that the network can read from and write to through attention-based addressing mechanisms. This design explicitly separates computation (performed by a controller network) from storage (handled by external memory), mirroring the architecture of classical von Neumann computers.

The controller in an NTM produces read and write weights that determine how information flows to and from memory. These weights are computed through a combination of content-based addressing (finding memory locations with similar content) and location-based addressing (accessing specific memory positions). The entire process remains differentiable, enabling end-to-end training via gradient descent.

Differentiable Neural Computers (DNCs), introduced in 2016, refined this architecture with more sophisticated memory management. DNCs added temporal memory linking (tracking the order in which memory locations were written) and dynamic memory allocation (reusing freed memory locations). These enhancements enabled DNCs to learn more complex algorithms, including graph traversal and question answering on structured data.

However, both NTMs and DNCs faced significant challenges that limited their adoption. Training proved unstable and sensitive to hyperparameter choices. The architectures struggled to scale to larger problems and never matched the performance of simpler models on practical tasks. Perhaps most critically, they emerged just before the Transformer revolution, and attention-based sequence models proved easier to train and scale.

2.2 Neural Program Synthesis

Neural Program Synthesis addresses the challenge of teaching neural networks to generate executable programs. This research direction recognizes that programs are precise specifications of algorithms and that generating programs represents a form of guaranteed correct computation. Key contributions include Neural Programmer-Interpreters (Reed & de Freitas, 2016), which learned to execute and compose programs, and various approaches to generating code from natural language specifications.

The Neural GPU (Kaiser & Sutskever, 2016) demonstrated that neural networks could learn binary addition and multiplication algorithms that generalize to arbitrary precision. This work showed that algorithmic generalization is achievable but required careful architectural design and training procedures. The architecture used convolutional recurrence with gating mechanisms to propagate information across sequence positions.

Neural Execution Engines (Velickovic et al., 2020) focused on learning to execute algorithmic subroutines. By training networks to imitate individual steps of algorithms like sorting and shortest path computation, they showed that neural networks could learn to perform complex algorithmic operations. However, these approaches required explicit supervision of intermediate algorithmic states.

2.3 Chain-of-Thought and Scratchpad Methods

Chain-of-Thought (CoT) prompting, introduced by Wei et al. in 2022, demonstrated that large language models perform significantly better on reasoning tasks when prompted to show their work step-by-step. Rather than directly producing an answer, the model generates intermediate reasoning steps before concluding. This approach dramatically improved performance on arithmetic, commonsense, and symbolic reasoning tasks.

The Scratchpad technique extended this idea by explicitly training models to output intermediate computation steps. By augmenting training data with detailed work traces, models learned to generate more systematic reasoning processes. Research showed that models trained with scratchpad annotations could solve longer arithmetic problems more accurately.

However, both CoT and Scratchpad methods remain fundamentally limited by the underlying Transformer architecture. The intermediate steps generated are still sequences of tokens produced through pattern matching, not genuine computational actions. While these methods improve performance on in-distribution problems, they do not solve the fundamental generalization challenge.

2.4 Program-Aided Language Models

Program-Aided Language Models (PAL), introduced by Gao et al. in 2023, take a different approach. Rather than attempting to perform computation within the neural network itself, PAL uses the language model to generate executable Python code that performs the actual computation. This neurosymbolic approach combines the flexibility of neural networks for understanding problems with the precision of program execution for solving them.

Toolformer (Schick et al., 2023) extended this paradigm by teaching language models to call external APIs. Models learned to determine when tool use was beneficial and how to format API calls correctly. This approach achieved the best of both worlds: neural understanding with external computational precision.

While PAL and Toolformer represent significant practical advances, they delegate computation to external systems rather than enabling the model to learn computation intrinsically. Actformer aims to achieve the best of both approaches: models that can both call external tools when available and perform computation internally when needed.

2.5 State Space Models and Transformer Alternatives

Recent work on State Space Models (SSMs), particularly Mamba (Gu & Dao, 2023), has emerged as a promising alternative to Transformer attention. Mamba achieves linear time complexity in sequence length through selective state spaces while maintaining competitive performance on language modeling benchmarks. The architecture introduces a selection mechanism that allows the model to filter and propagate information efficiently.

While SSMs address the quadratic attention complexity of Transformers, they do not fundamentally address the computation learning problem. The state space still performs pattern recognition and sequence modeling rather than learning discrete computational actions. However, the efficient state management mechanisms in SSMs provide valuable insights for Actformer's working memory design.

3. Actformer Architecture Overview

3.1 Core Design Principles

Actformer is founded on four core design principles that distinguish it from all existing neural architectures:

Principle 1: Action-Centric Learning - The fundamental unit of prediction is the action, not the token. Actions are discrete computational operations that transform problem state. By predicting actions, the model learns computational procedures rather than surface patterns.

Principle 2: Explicit State Management - The architecture maintains an explicit working memory that represents the current state of computation. This state is directly observable and modifiable, enabling systematic algorithm execution and debugging.

Principle 3: Differentiable Operations - All operations, including state modifications, are differentiable where needed for training. Non-differentiable operations are handled through gradient estimation or auxiliary losses, enabling end-to-end learning.

Principle 4: Compositional Generalization - The action vocabulary is designed for compositional reuse. Actions learned in one context transfer to novel problems, enabling systematic generalization beyond training distributions.

3.2 High-Level Architecture

The Actformer architecture comprises four primary components that work together to enable action-based learning:

Component	Function	Key Innovation
Action Space	Defines vocabulary of computational operations	Hierarchical, learnable action tokens
Working Memory	Maintains explicit state during computation	Differentiable read/write with attention
Action Execution Engine	Implements action semantics and state transitions	Hybrid symbolic-neural execution
Action Predictor	Predicts next action given context and state	Attention over action history and state

Table 1. Core Components of the Actformer Architecture

The information flow in Actformer proceeds as follows: First, the input (e.g., a problem specification or question) is encoded into the initial working memory state. The Action Predictor then examines the current state and action history to predict the next action. The Action Execution Engine implements this action, modifying the working memory accordingly. This cycle repeats until a termination action is predicted or a maximum step count is reached. The final state then yields the output.

4. Core Components

4.1 Action Space and Tokenization

The Action Space is the foundational abstraction in Actformer. It defines what operations the model can learn and execute. Unlike token vocabularies in language models, which are derived from data statistics, the action vocabulary is designed to capture fundamental computational primitives.

4.1.1 Action Categories

We organize actions into four hierarchical categories, each serving a distinct computational purpose:

Category	Examples	Role
Primitive Operations	ADD, SUBTRACT, MULTIPLY, COMPARE, READ, WRITE	Basic computational building blocks
Memory Operations	LOAD, STORE, POINTER_MOVE, ALLOCATE, FREE	Working memory management
Control Flow	IF, ELSE, WHILE, FOR, BREAK, CONTINUE	Algorithmic structure
Meta Operations	CALL_TOOL, EMIT_OUTPUT, HALT, RECURSE	System-level operations

Table 2. Action Categories and Their Roles

4.1.2 Action Token Representation

Each action is represented as a composite token that encodes both the operation type and its parameters. The representation follows a structured format:

```
ACTION_TOKEN = [OP_CODE, ARG_1, ARG_2, ..., ARG_N, MODIFIERS]
```

For example, an addition action might be represented as:

```
ADD [reg_a] [reg_b] -> [reg_c] // Add values in reg_a and reg_b, store in reg_c
```

This structured representation enables the model to learn the semantics of operations in a compositional manner. The operation code (ADD) determines the type of computation, while the arguments specify the data locations. Crucially, the same ADD operation applies regardless of the magnitude of the numbers being added.

4.1.3 Learnable Action Embeddings

While the action vocabulary has a defined structure, we also learn continuous embeddings for each action. These embeddings capture semantic relationships between actions and enable gradient-based learning. The embedding for an action token is computed as:

```
e_action = E_op(op_code) + SUM(E_arg(arg_i)) + E_mod(modifiers)
```

where E_op, E_arg, and E_mod are learned embedding matrices for operation codes, arguments, and modifiers respectively. This factorized embedding enables generalization to novel action compositions not seen during training.

4.2 Working Memory System

The Working Memory is where Actformer maintains explicit state during computation. Unlike the implicit state in recurrent networks or the context window in Transformers, working memory is directly addressable and modifiable by actions.

4.2.1 Memory Architecture

The working memory consists of three interconnected components:

Register File - A set of fixed-size vectors that hold intermediate computational results. Registers provide fast, direct access to frequently used values. The number and size of registers is a hyperparameter, typically ranging from 16 to 256 registers of 64 to 512 dimensions each.

Scratchpad Memory - A larger, addressable memory for storing variable-length data. The scratchpad uses attention-based addressing for both read and write operations, enabling the model to store and retrieve complex data structures.

Pointer Network - A set of learned pointers that track positions within data structures. Pointers enable sequential access patterns essential for algorithms like addition (process digits from right to left) and sorting (track current positions).

4.2.2 Memory Operations

Memory operations in Actformer are designed to be differentiable while maintaining precise semantics. Key operations include:

Operation	Description	Differentiability
READ(addr)	Retrieve value from memory address with content-based attention	Fully differentiable
WRITE(addr, val)	Store value at address using differentiable write weights	Fully differentiable
POINTER_MOVE(ptr, delta)	Move pointer by delta positions (discrete)	Straight-through estimator
ALLOCATE(size)	Allocate new memory region of specified size	Gradient through allocation score

Table 3. Memory Operations and Their Differentiability Properties

4.3 Action Execution Engine

The Action Execution Engine (AEE) is the component that implements the actual semantics of actions. It bridges the gap between the discrete action predictions and the continuous state modifications required for differentiable training.

4.3.1 Hybrid Symbolic-Neural Execution

A key innovation in Actformer is the hybrid execution model that combines symbolic operations with neural computations. For operations with well-defined mathematical semantics (like addition), we use exact symbolic implementations. For operations requiring learned behavior (like recognizing patterns or making judgments), we use neural networks.

This hybrid approach provides several advantages. First, it ensures correctness for algorithmic operations - when the model executes an ADD action, the result is guaranteed to be mathematically correct. Second, it enables learning for fuzzy operations that cannot be precisely specified. Third, it provides interpretability - we can trace exactly which operations were performed to reach a conclusion.

4.3.2 Action Selection and Execution Pipeline

The execution pipeline proceeds in stages:

Stage 1 - Action Prediction: The Action Predictor examines the current working memory state and action history to produce a probability distribution over possible next actions.

Stage 2 - Action Selection: During training, we sample from this distribution; during inference, we typically select the highest-probability action (with optional beam search).

Stage 3 - Parameter Binding: The selected action's parameters are bound to specific memory locations based on the model's attention over registers and scratchpad.

Stage 4 - Execution: The Action Execution Engine carries out the operation, producing new memory states and any external outputs.

Stage 5 - State Update: Working memory is updated with the results, and the action is appended to the history for context in subsequent predictions.

4.4 Differentiable Operation Primitives

The Differentiable Operation Primitives are the atomic computations that actions can invoke. These primitives form the instruction set that the model learns to compose into complex algorithms.

4.4.1 Arithmetic Primitives

Arithmetic primitives operate on numerical values stored in registers. We implement these using differentiable approximations where exact operations are non-differentiable:

Primitive	Implementation	Gradient Flow
ADD(a, b)	a + b (exact)	Direct gradients
MULTIPLY(a, b)	a * b (exact)	Direct gradients
COMPARE(a, b)	sigmoid(k*(a-b)) with temperature k	Soft gradients
MAX(a, b)	softmax-weighted sum	Attention gradients
DIGIT_EXTRACT(n, pos)	floor(n / 10^pos) mod 10	Straight-through estimator

Table 4. Arithmetic Primitives and Their Implementation Details

4.4.2 Control Flow Primitives

Control flow primitives enable the model to learn conditional and iterative algorithms. These require special handling to maintain differentiability:

```
IF condition THEN action_seq ELSE action_seq
```

For conditional execution, we use a soft branching mechanism where both branches are executed but their outputs are weighted by the condition probability. This maintains gradient flow through both paths during training while enabling discrete decisions during inference.

```
WHILE condition DO action_seq
```

Loops are handled through unrolling with a maximum iteration limit. The condition is evaluated at each step, and execution terminates when the condition becomes false or the limit is reached. Backpropagation through time accumulates gradients across all loop iterations.

5. Mathematical Formalization

5.1 State Transition Formalism

We formalize Actformer's computation as a state transition system. Let S denote the state space (the working memory), A denote the action space, and T: S x A -> S denote the transition function implemented by the Action Execution Engine.

A computation trace is a sequence of state-action pairs:

```
tau = [(s_0, a_0), (s_1, a_1), ..., (s_T, a_T)]
```

where s_{t+1} = T(s_t, a_t) for each t. The initial state s_0 is determined by encoding the input, and the final state s_T contains the output when a HALT action is executed.

5.2 Action Probability Distribution

The Action Predictor defines a policy pi that maps states to distributions over actions:

```
pi(a | s, h) = softmax(f_theta(s, h))_a
```

where h is the action history (previous actions and their contexts), and f_theta is a neural network parameterized by theta. The network architecture uses attention over both the working memory state and the action history:

```
f_theta(s, h) = MLP(concat(Attention(q_s, K_s, V_s), Attention(q_h, K_h, V_h)))
```

This dual attention mechanism enables the model to consider both the current computational state and the sequence of actions taken to reach it when predicting the next action.

5.3 Training Objective

We train Actformer using a combination of supervised learning (when action traces are available) and reinforcement learning (when only final outputs are available). The overall training objective is:

```
L = L_supervised + lambda * L_RL
```

The supervised loss encourages the model to predict correct action sequences:

```
L_supervised = -SUM_t log pi(a_t^* | s_t, h_t)
```

where a_t^* is the target action at step t from the training data.

The reinforcement learning loss optimizes for correct final outputs:

```
L_RL = -E[reward(s_T)] where reward measures output correctness
```

6. Training Paradigm

6.1 Curriculum Learning Strategy

Training Actformer effectively requires careful curriculum design. We propose a progressive curriculum that builds complexity:

Phase 1 - Primitive Actions: Train on individual action execution. Given a state and target action, learn to correctly modify state. This phase teaches the model the semantics of each action type.

Phase 2 - Action Sequences: Train on short action sequences for simple tasks. For addition, this might be sequences that add single-digit numbers. The model learns to predict coherent action chains.

Phase 3 - Algorithm Learning: Train on full algorithms for moderately complex problems. The model learns to compose actions into complete algorithms that handle various input cases.

Phase 4 - Generalization: Train on problems with varying input sizes and edge cases. The model learns to generalize algorithms beyond the specific instances seen in earlier phases.

Phase 5 - Multi-Task Learning: Train on diverse tasks simultaneously. The model learns to select appropriate algorithms and transfer knowledge between related tasks.

6.2 Reinforcement Learning Integration

For tasks where optimal action sequences are not available as supervision, we employ reinforcement learning. The key challenge is the discrete nature of actions and the sparse reward structure (rewards typically only at task completion).

We address these challenges through several techniques. First, we use a baseline value function to reduce variance in policy gradient estimates. Second, we employ curriculum-shaped rewards that provide intermediate feedback for partial progress. Third, we use experience replay to stabilize training and prevent catastrophic forgetting.

The policy gradient update follows the standard REINFORCE with baseline formulation:

```
nabla_theta J = E[SUM_t (R_t - b(s_t)) * nabla_theta log pi(a_t | s_t)]
```

where R_t is the cumulative reward from time t onward and b(s_t) is the baseline value estimate.

7. Implementation Roadmap

Implementing Actformer requires significant engineering effort. We outline a phased approach that enables iterative development and validation:

Phase	Duration	Key Milestones
Proof of Concept	2-3 months	Basic action space, simple memory, addition task working
Core Architecture	3-4 months	Full action vocabulary, working memory system, multiple arithmetic tasks
Training Pipeline	2-3 months	Curriculum learning, RL integration, distributed training
Scaling	4-6 months	Large-scale training, multi-task learning, transfer learning
Production	2-3 months	Optimization, deployment, API development

Table 5. Implementation Roadmap Phases and Timeline

7.1 Proof of Concept Architecture

For the initial proof of concept, we recommend the following minimal architecture that captures the essential Actformer principles:

```
class ActformerPoC: def __init__(self): # Minimal action space for arithmetic self.action_vocab = ['READ', 'WRITE', 'ADD', 'CARRY', 'OUTPUT', 'HALT'] # Working memory: 8 registers + 64-cell scratchpad self.registers = [0] * 8 self.scratchpad = [None] * 64 self.pointers = {'read_ptr': 0, 'write_ptr': 0} # Action predictor: simple transformer self.predictor = Transformer( d_model=256, nhead=4, num_layers=4 ) # Action embeddings self.action_embeddings = nn.Embedding(len(self.action_vocab), 256) def forward(self, input_numbers): # Encode input to initial state state = self.encode_input(input_numbers) actions_taken = [] for step in range(max_steps): # Predict next action action_logits = self.predictor(state, actions_taken) action = self.select_action(action_logits) # Execute action state = self.execute_action(action, state) actions_taken.append(action) if action == 'HALT': break return self.decode_output(state)
```

8. Conclusion and Future Directions

Actformer represents a fundamental reimagining of how neural networks can learn computation. By shifting from token prediction to action prediction, we enable models to internalize actual algorithms rather than surface patterns. The architecture's explicit working memory, structured action space, and hybrid symbolic-neural execution provide the necessary components for genuine algorithmic learning.

The path forward involves several exciting research directions. First, scaling Actformer to larger action vocabularies and more complex tasks will test the limits of the approach. Second, integrating Actformer with existing language models could provide a powerful combination of natural language understanding and computational precision. Third, extending the action space to include learned operations would enable the model to discover novel algorithms.

The ultimate vision for Actformer is a model that can learn any computable function - a neural network that truly understands computation rather than merely approximating its surface statistics. While significant challenges remain, the architectural foundations laid out in this document provide a clear roadmap toward that goal.

References

Graves, A., Wayne, G., & Danihelka, I. (2014). Neural Turing Machines. arXiv:1410.5401.

Graves, A., Wayne, G., Reynolds, M., et al. (2016). Hybrid computing using a neural network with dynamic external memory. Nature, 538(7626), 471-476.

Kaiser, L., & Sutskever, I. (2016). Neural GPUs Learn Algorithms. ICLR 2016.

Wei, J., Wang, X., Schuurmans, D., et al. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. NeurIPS 2022.

Gao, L., Madaan, A., Zhou, S., et al. (2023). PAL: Program-aided Language Models. ICML 2023.

Schick, T., Dwivedi-Yu, J., Dessi, R., et al. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. NeurIPS 2023.

Gu, A., & Dao, T. (2023). Mamba: Linear-Time Sequence Modeling with Selective State Spaces. arXiv:2312.00752.

Velickovic, P., Ying, R., Padovano, M., et al. (2020). Neural Execution Engines: Learning to Execute Subroutines. NeurIPS 2020.

Reed, S., & de Freitas, N. (2016). Neural Programmer-Interpreters. ICML 2016.

Nye, M., Andreassen, A., Gur-Ari, G., et al. (2021). Show Your Work: Scratchpads for Intermediate Reasoning with Language Models. arXiv:2112.00114.