Research Code Sample: In-Context Learning of Finite Automata

Author: Trenton O'Bannon
Role: Undergraduate Research, UC Berkeley CS 182
Project: Investigating Neural Architectures' Ability to In-Context Learn Deterministic Finite Automata
Repository: https://github.com/Toba4366/CS182_FinalProject.git
Date: December 2025

Overview

This document presents a research codebase I co-developed for studying **in-context learning (ICL)** capabilities across neural architectures. The project investigates how Transformers, LSTMs, and RNNs learn deterministic finite automata (DFAs) from demonstration sequences—a task inspired by computational social science research on measuring cognitive complexity.

Research Question: Can neural networks infer the transition function of a finite automaton from observed state-action sequences, and how do different architectures compare?

Key Contributions: - Synthetic data pipeline for generating DFAs and sampling trajectories - Unified training framework supporting multiple architectures - Curriculum learning implementation for complex automata - Comprehensive evaluation with length extrapolation tests

---

1. Finite State Machine Generation

The core data generation involves creating random connected DFAs and sampling trajectories for in-context learning sequences.

1.1 DFA Generator (`src/fsm/generator.py`)

```
"""
Finite State Machine (FSM) generator utilities.
Creates random connected DFAs with configurable complexity.

Vocabulary layout:
- States: 0 to num_states-1 (always in range [0, MAX_STATES-1])
- Actions: MAX_STATES to MAX_STATES+max_actions-1
"""

from dataclasses import dataclass
```

```python
from typing import Dict, Optional
import random

# Type alias for clarity: FSM maps state -> {action -> next_state}
FSM = Dict[int, Dict[int, int]]

@dataclass
class FSMGeneratorConfig:
    """Configuration for DFA generation."""
    num_states: int = 5
    min_actions: int = 3
    max_actions: int = 8
    seed: Optional[int] = None

class FSMGenerator:
    """Random fully-connected DFA generator."""

    def __init__(self, config: FSMGeneratorConfig):
        # Validate configuration
        assert config.num_states >= 2, "DFA requires at least two states"
        assert config.min_actions >= 1, "Each state needs at least one action"
        assert config.max_actions >= config.min_actions

        self.config = config
        self.rng = random.Random(config.seed)
        self.action_ids = list(range(MAX_STATES, MAX_STATES + config.max_actions))

    def generate(self) -> FSM:
        """
        Generate a random connected DFA.

        Algorithm:
        1. Build spanning tree to ensure connectivity
        2. Fill remaining transitions randomly

        Returns:
            Dictionary mapping state -> {action -> next_state}
        """
        num_states = self.config.num_states
        fsm: FSM = {state: {} for state in range(num_states)}

        # Step 1: Build spanning tree for connectivity
        remaining_states = list(range(1, num_states))
        connected_states = [0]

        while remaining_states:
```

```
            src = self.rng.choice(connected_states)
            dst = remaining_states.pop(self.rng.randrange(len(remaining_states)))

            action_id = self._sample_unused_action(fsm[src])
            fsm[src][action_id] = dst
            connected_states.append(dst)

        # Step 2: Fill in remaining transitions
        for state in range(num_states):
            for action_id in self.action_ids:
                if action_id not in fsm[state]:
                    next_state = self.rng.randrange(num_states)
                    fsm[state][action_id] = next_state

        return fsm

    def generate_with_absorbing_state(self) -> FSM:
        """Generate DFA with one absorbing state (all transitions loop back)."""
        fsm = self.generate()
        absorbing = self.rng.randrange(self.config.num_states)

        for action_id in self.action_ids:
            fsm[absorbing][action_id] = absorbing
        return fsm
```

Design Rationale: - **Spanning tree construction** guarantees all states are reachable - **Deterministic transitions**: Every (state, action) pair maps to exactly one state - **Absorbing states** support experiments on automata complexity (inspired by Oprea, 2020)

1.2 Trajectory Sampler (`src/fsm/trajectory_sampler.py`)

```
"""
Trajectory sampler for generating random walks over FSMs.
Each trajectory records both states and actions for reconstruction.
"""

from dataclasses import dataclass
from typing import Dict, List, Optional, Sequence

Trajectory = Dict[str, Sequence[int]]
# {"states": [s0, s1, ..., sT], "actions": [a0, a1, ..., aT-1]}

@dataclass
class TrajectorySamplerConfig:
    num_states: int = 5
```

```
    min_actions_per_state: int = 3
    max_actions_per_state: int = 8
    num_trajectories: int = 3
    trajectory_length: int = 64
    seed: Optional[int] = None

class TrajectorySampler:
    """Sampler that generates rollouts from FSMs."""

    def __init__(self, config: TrajectorySamplerConfig):
        self.config = config
        self.rng = random.Random(config.seed)
        self.generator = FSMGenerator(config.to_generator_config())
        self.fsm: FSM = self.generator.generate()

    def rollout(self, traj_length: int, start_state: Optional[int] = None) -> Trajectory:
        """Perform one random walk over the FSM."""
        state = start_state if start_state is not None else \
                self.rng.choice(list(self.fsm.keys()))

        states = [state]
        actions = []

        for _ in range(traj_length):
            transitions = self.fsm[state]
            action = self.rng.choice(list(transitions.keys()))
            next_state = transitions[action]

            actions.append(action)
            states.append(next_state)
            state = next_state

        return {"states": states, "actions": actions}
```

Output Example:

```
trajectory = {
    "states":  [0, 1, 2, 1, 3],   # Length T+1
    "actions": [8, 9, 10, 11],    # Length T
}
# Represents: s0 --a8--> s1 --a9--> s2 --a10--> s1 --a11--> s3
```

---

2. In-Context Learning Dataset

2.1 Sequence Format

Each training sample follows an ICL paradigm:

```
[Demo 1] S  A  S  A  S  ... <eos>
[Demo 2] S  A  S  A  S  ... <eos>
[Demo 3] S  A  S  A  S  ... <eos>
<query> S  A  [?] A  [?] A  [?] <eos>
```

The model observes demonstration trajectories, then predicts unknown states in the query.

2.2 Dataset Implementation (`src/datasets/moore_dataset.py`)

```python
"""
Dataset builder for in-context learning of finite automata.
Generates tokenized sequences with proper loss masking.
"""

class MooreICLDataset(Dataset):
    """
    Dataset storing trajectories for in-context learning.

    Vocabulary layout (fixed for reproducibility):
        - 0 ... MAX_STATES-1     : state IDs
        - MAX_STATES ... +MAX_ACTIONS-1 : action IDs
        - eos_token              : end-of-sequence
        - query_token            : marks query segment start
        - pad_token              : padding for batching
    """

    def __init__(self, sample_indices, all_samples, sampler_config, max_seq_len):
        self.samples = all_samples
        self.sample_ids = sample_indices
        self.max_seq_len = max_seq_len

        # Fixed vocabulary layout
        self.action_offset = MAX_STATES
        self.eos_token = MAX_STATES + MAX_ACTIONS
        self.query_token = self.eos_token + 1
        self.pad_token = self.eos_token + 2
        self.vocab_size = self.pad_token + 1

    def __getitem__(self, idx: int) -> Dict[str, torch.Tensor]:
        """
        Build input sequence, targets, and loss mask.
```

```
        Example:
            sequence = [0, 8, 1, 8, 2, <eos>, ..., <query>, 2, 9, 0, 8, 1, <eos>]
            loss_mask = [F, F, F, F, F, F, ..., F, F, F, T, F, T, F]

        Loss is computed ONLY on query state predictions (True positions).
        """
        sample = self.samples[self.sample_ids[idx]]
        demos = sample["demos"]
        query = sample["query"]

        # Encode demonstrations
        sequence_tokens = []
        sequence_mask = []

        for demo in self._select_demos(demos, num_samples=3):
            for token in self._trajectory_to_tokens(demo):
                sequence_tokens.append(token)
                sequence_mask.append(False)  # No loss on demo tokens
            sequence_tokens.append(self.eos_token)
            sequence_mask.append(False)

        # Encode query with masked states
        sequence_tokens.append(self.query_token)
        sequence_mask.append(False)

        query_tokens, query_mask = self._encode_query(query)
        sequence_tokens.extend(query_tokens)
        sequence_mask.extend(query_mask)  # True for state predictions

        # Build shifted input/target for autoregressive training
        input_tokens = [sequence_tokens[0]] + sequence_tokens[:-1]
        target_tokens = sequence_tokens

        return {
            "input_ids": torch.tensor(input_tokens, dtype=torch.long),
            "target_ids": torch.tensor(target_tokens, dtype=torch.long),
            "loss_mask": torch.tensor(sequence_mask, dtype=torch.bool),
        }

    def _encode_query(self, traj: Dict[str, List[int]]) -> Tuple[List[int], List[bool]]:
        """
        Encode query trajectory with state prediction masks.

        tokens = [S , A , S , A , S ]
        mask   = [F,  F,  T,  F,  T]  # True = predict this state
```

```
    """
    tokens, mask = [], []
    states, actions = traj["states"], traj["actions"]

    tokens.append(states[0])
    mask.append(False)  # Starting state is given

    for idx, action in enumerate(actions):
        tokens.append(action)
        mask.append(False)  # Actions are given

        tokens.append(states[idx + 1])
        mask.append(True)  # Predict this state

    return tokens, mask
```

Key Design Decisions: - **Loss masking**: Only backpropagate on query state predictions - **Teacher forcing**: Use ground truth tokens during training - **Variable length demos**: Add ±20% variation for robustness

---

3. Neural Architecture Implementations

All models share a unified interface for fair comparison:

```
def forward(input_ids, targets=None, unknown_mask=None) -> (logits, loss)
```

3.1 Transformer with Rotary Position Embeddings (`src/models/moore_transformer.py`)

```
"""
Causal transformer for predicting DFA states.
Uses Rotary Position Embeddings (RoPE) for position encoding.
"""

@dataclass
class TransformerConfig:
    vocab_size: int
    num_states: int
    max_seq_len: int = 512
    d_model: int = 256
    num_heads: int = 8
    num_layers: int = 4
    d_ff: int = 512
    dropout: float = 0.1
    rope_theta: float = 10000.0  # RoPE frequency base

class RotaryPositionEmbedding(nn.Module):
```

```
    """RoPE: Encodes relative positions in attention."""

    def __init__(self, dim: int, max_seq_len: int = 2048, theta: float = 10000.0):
        super().__init__()
        # Precompute inverse frequencies
        inv_freq = 1.0 / (theta ** (torch.arange(0, dim, 2).float() / dim))
        self.register_buffer("inv_freq", inv_freq, persistent=False)

    def forward(self, x: torch.Tensor, seq_len: int) -> Tuple[torch.Tensor, torch.Tensor]:
        """Return (cos, sin) tensors for rotary embedding."""
        t = torch.arange(seq_len, device=x.device, dtype=self.inv_freq.dtype)
        freqs = torch.outer(t, self.inv_freq)
        emb = torch.cat([freqs, freqs], dim=-1)
        return emb.cos(), emb.sin()

def apply_rotary_pos_emb(q, k, cos, sin):
    """Apply rotary embedding to query and key tensors."""
    def rotate_half(x):
        x1, x2 = x.chunk(2, dim=-1)
        return torch.cat([-x2, x1], dim=-1)

    cos = cos.unsqueeze(0).unsqueeze(0)  # (1, 1, seq_len, head_dim)
    sin = sin.unsqueeze(0).unsqueeze(0)

    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed, k_embed

class CausalSelfAttention(nn.Module):
    """Multi-head attention with RoPE and causal masking."""

    def __init__(self, d_model, num_heads, dropout, rope):
        super().__init__()
        self.num_heads = num_heads
        self.head_dim = d_model // num_heads
        self.rope = rope

        self.q_proj = nn.Linear(d_model, d_model)
        self.k_proj = nn.Linear(d_model, d_model)
        self.v_proj = nn.Linear(d_model, d_model)
        self.out_proj = nn.Linear(d_model, d_model)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        bsz, seq_len, _ = x.size()

        # Project to Q, K, V
```

```
        q = self.q_proj(x).view(bsz, seq_len, self.num_heads, self.head_dim).transpose(1, 2)
        k = self.k_proj(x).view(bsz, seq_len, self.num_heads, self.head_dim).transpose(1, 2)
        v = self.v_proj(x).view(bsz, seq_len, self.num_heads, self.head_dim).transpose(1, 2)

        # Apply RoPE
        cos, sin = self.rope(q, seq_len=seq_len)
        q, k = apply_rotary_pos_emb(q, k, cos, sin)

        # Scaled dot-product attention with causal mask
        attn = F.scaled_dot_product_attention(q, k, v, is_causal=True)
        return self.out_proj(attn.transpose(1, 2).contiguous().view(bsz, seq_len, -1))

class MooreTransformer(nn.Module):
    """Decoder-only transformer for state prediction."""

    def __init__(self, config: TransformerConfig):
        super().__init__()
        self.config = config
        self.token_embedding = nn.Embedding(config.vocab_size, config.d_model)

        self.rope = RotaryPositionEmbedding(
            dim=config.d_model // config.num_heads,
            max_seq_len=config.max_seq_len
        )

        self.blocks = nn.ModuleList([
            TransformerBlock(config, self.rope) for _ in range(config.num_layers)
        ])
        self.ln_f = nn.LayerNorm(config.d_model)
        self.head = nn.Linear(config.d_model, config.num_states)

    def forward(self, input_ids, targets=None, unknown_mask=None):
        x = self.token_embedding(input_ids)

        for block in self.blocks:
            x = block(x)

        x = self.ln_f(x)
        logits = self.head(x)

        loss = None
        if targets is not None:
            mask = unknown_mask if unknown_mask is not None else \
                   torch.ones_like(targets, dtype=torch.bool)

            # Compute cross-entropy only on masked positions
```

```
            if mask.any():
                flat_logits = logits.view(-1, self.config.num_states)
                flat_targets = targets.view(-1)
                flat_mask = mask.view(-1)
                loss = F.cross_entropy(flat_logits[flat_mask], flat_targets[flat_mask])

        return logits, loss
```

3.2 LSTM Implementation (src/models/moore_lstm.py)

```
"""
LSTM for predicting DFA states.
Serves as baseline with explicit sequential processing.
"""

@dataclass
class LSTMConfig:
    vocab_size: int
    num_states: int
    d_model: int = 256
    num_layers: int = 2
    dropout: float = 0.1
    bidirectional: bool = False

class MooreLSTM(nn.Module):
    """LSTM specialized for DFA state prediction."""

    def __init__(self, config: LSTMConfig):
        super().__init__()
        self.config = config
        self.num_directions = 2 if config.bidirectional else 1

        self.token_embedding = nn.Embedding(config.vocab_size, config.d_model)

        self.lstm = nn.LSTM(
            input_size=config.d_model,
            hidden_size=config.d_model,
            num_layers=config.num_layers,
            dropout=config.dropout if config.num_layers > 1 else 0.0,
            batch_first=True,
            bidirectional=config.bidirectional,
        )

        lstm_output_dim = config.d_model * self.num_directions
        self.head = nn.Linear(lstm_output_dim, config.num_states, bias=False)
```

```
        self._init_weights()

    def _init_weights(self):
        """Best practices for LSTM initialization."""
        for name, param in self.lstm.named_parameters():
            if "weight_ih" in name:
                nn.init.xavier_uniform_(param.data)
            elif "weight_hh" in name:
                nn.init.orthogonal_(param.data)  # Helps gradient flow
            elif "bias" in name:
                nn.init.zeros_(param.data)

    def forward(self, input_ids, targets=None, unknown_mask=None):
        x = self.token_embedding(input_ids)
        lstm_output, _ = self.lstm(x)
        logits = self.head(lstm_output)

        loss = None
        if targets is not None:
            mask = unknown_mask if unknown_mask is not None else \
                   torch.ones_like(targets, dtype=torch.bool)
            if mask.any():
                loss = F.cross_entropy(
                    logits.view(-1, self.config.num_states)[mask.view(-1)],
                    targets.view(-1)[mask.view(-1)]
                )

        return logits, loss
```

---

4. Training Framework

4.1 Curriculum Learning Strategy

Direct training on complex DFAs (5 states, 5 actions) failed for transformers. We implemented curriculum learning:

```
"""
Curriculum learning: Train on simple DFAs first, then complex ones.
"""

@dataclass
class CurriculumStage:
    """Configuration for one curriculum stage."""
    num_states: int
    min_actions_per_state: int
```

```
    max_actions_per_state: int
    num_samples: int
    epochs: int

# Training pipeline
stages = [
    # Stage 1: Simple DFAs (3 states, 3 actions)
    CurriculumStage(
        num_states=3,
        min_actions_per_state=3,
        max_actions_per_state=3,
        num_samples=10_000,
        epochs=10,
    ),
    # Stage 2: Complex DFAs (5 states, 4-5 actions)
    CurriculumStage(
        num_states=5,
        min_actions_per_state=4,
        max_actions_per_state=5,
        num_samples=10_000,
        epochs=3,
    ),
]

# Model is trained sequentially through stages
for stage in stages:
    model = train_stage(stage, model, args)
```

4.2 Trainer Implementation (`src/training/icl_trainer.py`)

```
"""
Training loop with evaluation utilities.
"""

@dataclass
class TrainingConfig:
    batch_size: int = 8
    learning_rate: float = 1e-3
    weight_decay: float = 0.0
    num_epochs: int = 3

class MooreICLTrainer:
    """Training wrapper supporting any architecture."""

    def __init__(self, model, train_dataset, val_dataset, collator, config):
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
        self.model = model.to(self.device)

        self.train_loader = DataLoader(
            train_dataset, batch_size=config.batch_size, shuffle=True, collate_fn=collator
        )
        self.val_loader = DataLoader(
            val_dataset, batch_size=config.batch_size, shuffle=False, collate_fn=collator
        )

        self.optimizer = torch.optim.AdamW(
            self.model.parameters(),
            lr=config.learning_rate,
            weight_decay=config.weight_decay,
        )

    def train(self):
        for epoch in range(1, self.config.num_epochs + 1):
            train_loss = self._run_epoch()
            val_loss = self.evaluate()
            print(f"[Epoch {epoch}] Train: {train_loss:.4f} | Val: {val_loss:.4f}")

    def _run_epoch(self) -> float:
        self.model.train()
        total_loss = 0.0

        for batch in self.train_loader:
            batch = {k: v.to(self.device) for k, v in batch.items()}

            _, loss = self.model(
                batch["input_ids"],
                targets=batch["target_ids"],
                unknown_mask=batch["loss_mask"],
            )

            self.optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)
            self.optimizer.step()

            total_loss += loss.item()

        return total_loss / len(self.train_loader)

@torch.no_grad()
def evaluate_model(model, dataloader, device) -> float:
    """Compute next-state prediction accuracy."""
```

```
    model.eval()
    correct, total = 0, 0

    for batch in dataloader:
        batch = {k: v.to(device) for k, v in batch.items()}
        logits, _ = model(batch["input_ids"])
        predictions = logits.argmax(dim=-1)

        mask = batch["loss_mask"]
        targets = batch["target_ids"]

        correct += ((predictions == targets) & mask).sum().item()
        total += mask.sum().item()

    return correct / total
```

---

5. Experimental Results Summary

5.1 Architecture Comparison

Model	Parameters	Val Accuracy (5S5A)	Key Finding
Transformer (2L)	~52K	**99.9%**	Requires curriculum learning
Large LSTM (256d, 1L)	~531K	98.9%	10× parameters for similar accuracy
Small LSTM (64d, 2L)	~68K	84.2%	Plateaus at lower accuracy
Vanilla RNN	~68K	~Random	Failed to learn

5.2 Key Observations

1. **Phase Transitions**: Transformers exhibit sharp accuracy jumps (epochs 3-4), suggesting sudden discovery of correct attention patterns
2. **Curriculum Necessity**: Transformers fail on 5S5A without pre-training on 3S3A; LSTMs succeed with direct training but require more parameters
3. **Length Extrapolation**: LSTMs maintain accuracy on longer queries (200-600 tokens); Transformers degrade slightly due to positional embedding limitations

---

6. Running the Code

Setup

```
# Install dependencies
pip install torch numpy

# Generate dataset (10,000 samples)
python -c "from src.datasets.moore_dataset import load_or_create_icl_samples, ICLDatasetConf
```

Training

```
# Train transformer with curriculum learning
python experiments/run_icl_transformer.py --batch-size 24 --num-layers 2

# Train LSTM directly on complex DFAs
python experiments/run_icl_lstm.py --epochs 20 --d-model 256 --num-layers 1

# Curriculum learning for LSTM
python experiments/run_icl_lstm_curriculum.py --dataset-type simple --checkpoint-name stage1
python experiments/run_icl_lstm_curriculum.py --dataset-type complex --checkpoint-name stage
```

Evaluation

```
# Test on held-out data
python -c "
from src.training.icl_trainer import evaluate_model
# ... load model and test_loader ...
acc = evaluate_model(model, test_loader, device)
print(f'Test Accuracy: {acc:.4f}')
"
```

---

7. Repository Structure

```
CS182_FinalProject/
  src/
      fsm/
          generator.py        # DFA generation
          trajectory_sampler.py  # Random walk sampling
      datasets/
          moore_dataset.py    # ICL dataset builder
      models/
          moore_transformer.py  # Transformer + RoPE
```

```
        moore_lstm.py       # LSTM baseline
        moore_vanilla_rnn.py  # RNN baseline
    training/
        icl_trainer.py      # Transformer trainer
        lstm_trainer.py     # LSTM trainer
experiments/
    run_icl_transformer.py  # Curriculum training script
    run_icl_lstm.py         # Direct LSTM training
    run_icl_lstm_curriculum.py  # Curriculum LSTM training
data/                       # Generated datasets
checkpoints/                # Saved models
results/                    # Training metrics & plots
```

---

8. Reproducibility Notes

- **Random Seeds**: All generators accept seed parameters for reproducibility
- **Checkpointing**: Models save optimizer state for training resumption
- **Metrics Logging**: Training history saved to JSON for analysis
- **Configuration Dataclasses**: All hyperparameters documented and serializable

---

9. Acknowledgments

This work was completed as part of CS 182 (Deep Learning) at UC Berkeley. Co-authors: Keshab Agarwal, Evan Davis, Yuri Lee. The research was inspired by Oprea (2020)'s work on cognitive complexity and Akyürek et al. (2024)'s study of transformers learning formal languages.

This code sample demonstrates research-grade implementation: modular design, comprehensive documentation, reproducible experiments, and systematic comparison of neural architectures on a well-defined learning task.