Replit Agent Prompt: Contrastive Representations Scale Reinforcement Learning to Massive Action Spaces

Project Overview

You are implementing experiments for an **ICML 2026 paper** that proves contrastive embeddings enable efficient reinforcement learning over massive action spaces through geometric uniformity properties.

Core Thesis: Reconstruction-based embeddings (BERT, LLaMA, Mistral) produce anisotropic representations with effective dimension ~50, causing linear regret in RL. Contrastive embeddings (SimCSE, Jina, LLM2Vec) produce uniform representations with effective dimension ~200, enabling sublinear regret and efficient exploration.

Theoretical Foundation:

- Section 3: Realizability depends on d_{eff} spanning reward function intrinsic dimensionality m
- Section 4: Contrastive learning optimizes $\mathcal{L}_{\text{InfoNCE}} = -\frac{1}{\tau}\mathcal{L}_{\text{align}} + \mathcal{L}_{\text{unif}}$, where uniformity term prevents dimensional collapse
- Section 5: When $m > d_{\text{eff}}$, RKHS norm explodes → linear regret
- Section 6: Context-conditioned policies (transformers) inherit embedding geometry

Deadlines:

- Abstract: January 22, 2026
- Full paper: January 28, 2026
- **Today: January 15, 2026 → 13 days remaining**

Embedding Comparison Strategy

Complete Embedding Lineup (6 embeddings × 3 use cases = 18 conditions)

Anisotropic (Reconstruction-based - predicted to fail):

1. **BERT-base-uncased** (110M params, MLM-trained)
   - Classic reconstruction baseline
   - Expected $d_{\text{eff}} \approx 40$
   - Extraction: Use [CLS] token or mean-pool last hidden layer
2. **RoBERTa-base** (125M params, MLM-trained)
   - Modern BERT successor, better training

- Expected $d_{\text{eff}} \approx 50$
- Extraction: Same as BERT (mean-pool last hidden layer)

3. **LLaMA-3-8B-base** (8B params, CLM-trained, **NO contrastive tuning**)
   - Modern LLM with standard causal language modeling
   - Expected $d_{\text{eff}} \approx 60$
   - **Key narrative:** "Even massive LLMs have anisotropic representations when trained only with CLM"
   - **Extraction method:** Load pretrained LLaMA-3, take last layer hidden states, mean-pool over tokens
   - **Important:** This is the BASE model, not instruction-tuned, not contrastively tuned

Contrastive (Uniform - predicted to succeed): 4. **SimCSE-base** (110M params, BERT + contrastive fine-tuning)

- Original contrastive method from theory
- Expected $d_{\text{eff}} \approx 200$
- Extraction: Use sentence-transformers library (handles pooling automatically)

5. **Jina-embeddings-v3** (570M params, contrastive-trained from scratch)
   - Modern SOTA, tops MTEB leaderboard (Dec 2024)
   - Expected $d_{\text{eff}} \approx 220$
   - Lightweight, efficient
   - Extraction: Use Jina API or HuggingFace model
6. **LLM2Vec-LLaMA-3-8B** (8B params, **same LLaMA-3 base + contrastive fine-tuning**)
   - Takes LLaMA-3-base (condition #3) and adds bidirectional attention + contrastive training
   - Expected $d_{\text{eff}} \approx 210$
   - **Key narrative:** "Contrastive fine-tuning fixes LLM anisotropy - same base model, different training"
   - **This is the critical A/B comparison:** LLaMA-3-base vs LLM2Vec-LLaMA-3 isolates the effect of contrastive tuning

Note on LLaMA/Mistral: These are standard decoder LLMs. We extract embeddings by:

1. Loading the pretrained model (no special setup)
2. Passing text through the model
3. Taking the last layer hidden states

4. Mean-pooling over the sequence dimension
5. Projecting to 768-dim via PCA

No instruction tuning, no chat templates, just base embeddings from the language model.

Dimension Standardization

Problem: Embeddings have different native dimensions (BERT: 768, LLaMA: 4096, Jina: 1024)

Solution: Project all to 768 dimensions for fair comparison

python

```python
from sklearn.decomposition import PCA

def standardize_embedding_dim(embedding, target_dim=768):
    """
    Project embedding to target dimension.
    - If embedding.shape[-1] > target_dim: PCA reduction
    - If embedding.shape[-1] < target_dim: zero-pad
    - If embedding.shape[-1] == target_dim: return as-is
    """
    current_dim = embedding.shape[-1]

    if current_dim == target_dim:
        return embedding
    elif current_dim > target_dim:
        # PCA reduction (fit on dataset, transform each embedding)
        pca = PCA(n_components=target_dim)
        # Fit on sample of embeddings, then transform
        return pca.fit_transform(embedding.reshape(1, -1)).squeeze()
    else:
        # Zero-pad
        padding = np.zeros(target_dim - current_dim)
        return np.concatenate([embedding, padding])
```

Implementation note: Fit PCA once on a sample of 10K embeddings from each dataset, save transformer, apply to all.

Embedding Extraction Implementation

python

```
import torch
import numpy as np
from transformers import AutoModel, AutoTokenizer, BitsAndBytesConfig
from sentence_transformers import SentenceTransformer
from sklearn.decomposition import PCA
import pickle

class EmbeddingExtractor:
    """
    Unified interface for extracting embeddings from all 6 models.
    Handles dimension standardization and caching.
    """

    def __init__(self, model_name, target_dim=768, use_quantization=False):
        self.model_name = model_name
        self.target_dim = target_dim
        self.use_quantization = use_quantization
        self.model = None
        self.tokenizer = None
        self.pca = None

        self._load_model()

    def _load_model(self):
        """Load the appropriate model based on model_name."""

        if self.model_name == 'bert':
            self.model = AutoModel.from_pretrained('bert-base-uncased')
            self.tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')

        elif self.model_name == 'roberta':
            self.model = AutoModel.from_pretrained('roberta-base')
            self.tokenizer = AutoTokenizer.from_pretrained('roberta-base')

        elif self.model_name == 'llama3':
            # LLaMA-3-8B base (anisotropic, no contrastive tuning)
            if self.use_quantization:
                # 4-bit quantization: 16GB → 4GB VRAM
                quantization_config = BitsAndBytesConfig(
                    load_in_4bit=True,
                    bnb_4bit_compute_dtype=torch.float16,
                    bnb_4bit_use_double_quant=True,
                    bnb_4bit_quant_type="nf4"
```

```
        )
        self.model = AutoModel.from_pretrained(
            "meta-llama/Meta-Llama-3-8B",
            quantization_config=quantization_config,
            device_map="auto",
            torch_dtype=torch.float16
        )
    else:
        self.model = AutoModel.from_pretrained(
            "meta-llama/Meta-Llama-3-8B",
            torch_dtype=torch.float16,
            device_map="auto"
        )
    self.tokenizer = AutoTokenizer.from_pretrained("meta-llama/Meta-Llama-3-8B")
    self.tokenizer.pad_token = self.tokenizer.eos_token  # LLaMA needs this

elif self.model_name == 'simcse':
    # SimCSE: contrastively fine-tuned BERT
    self.model = SentenceTransformer('princeton-nlp/sup-simcse-bert-base-uncased')

elif self.model_name == 'jina':
    # Jina-v3: modern SOTA contrastive embeddings
    self.model = AutoModel.from_pretrained(
        'jinaai/jina-embeddings-v3',
        trust_remote_code=True
    )
    self.tokenizer = AutoTokenizer.from_pretrained('jinaai/jina-embeddings-v3')

elif self.model_name == 'llm2vec':
    # LLM2Vec: LLaMA-3 + contrastive fine-tuning
    from llm2vec import LLM2Vec

    self.model = LLM2Vec.from_pretrained(
        "McGill-NLP/LLM2Vec-Meta-Llama-3-8B-Instruct-mntp",
        peft_model_name_or_path="McGill-NLP/LLM2Vec-Meta-Llama-3-8B-Instruct-mntp-supervised",
        device_map="auto",
        torch_dtype=torch.float16
    )

# Move to GPU if available
if self.model is not None and torch.cuda.is_available():
    if self.model_name not in ['llama3', 'llm2vec']:  # These use device_map
        self.model = self.model.cuda()
```

```python
def encode(self, text, batch_size=1):
    """
    Extract embedding for a single text or batch of texts.

    Args:
        text: str or List[str]
        batch_size: int (only used for batched encoding)

    Returns:
        embedding: numpy array of shape (target_dim,) or (len(text), target_dim)
    """
    is_single = isinstance(text, str)
    if is_single:
        text = [text]

    # Extract raw embeddings
    if self.model_name in ['simcse']:
        # sentence-transformers handles everything
        raw_embs = self.model.encode(text, show_progress_bar=False)

    elif self.model_name == 'llm2vec':
        # LLM2Vec has its own encode method
        raw_embs = self.model.encode(text)

    else:
        # Manual extraction for BERT, RoBERTa, LLaMA, Jina
        raw_embs = []
        for t in text:
            inputs = self.tokenizer(
                t,
                return_tensors='pt',
                truncation=True,
                max_length=512,
                padding=True
            )

            if torch.cuda.is_available() and self.model_name not in ['llama3']:
                inputs = {k: v.cuda() for k, v in inputs.items()}

            with torch.no_grad():
                outputs = self.model(**inputs, output_hidden_states=True)

                # Extract last layer hidden states
                last_hidden = outputs.hidden_states[-1]  # (1, seq_len, hidden_dim)
```

```
                # Mean pool over sequence dimension
                embedding = last_hidden.mean(dim=1).squeeze(0)  # (hidden_dim,)

                # Move to CPU
                embedding = embedding.cpu().numpy()
                raw_embs.append(embedding)

            raw_embs = np.stack(raw_embs)

        # Standardize dimension to target_dim
        standardized_embs = []
        for emb in raw_embs:
            std_emb = self._standardize_dim(emb)
            standardized_embs.append(std_emb)

        standardized_embs = np.stack(standardized_embs)

        if is_single:
            return standardized_embs[0]
        return standardized_embs

    def _standardize_dim(self, embedding):
        """Project embedding to target_dim using PCA or padding."""
        current_dim = embedding.shape[-1]

        if current_dim == self.target_dim:
            return embedding

        elif current_dim > self.target_dim:
            # Use PCA (fit once on dataset, then transform)
            if self.pca is None:
                # PCA not yet fitted - will be fitted on first batch
                print(f"Warning: PCA not fitted for {self.model_name}. "
                      f"Call fit_pca() on dataset first.")
                # Temporary: fit on this single embedding (not ideal)
                self.pca = PCA(n_components=self.target_dim)
                self.pca.fit(embedding.reshape(1, -1))

            return self.pca.transform(embedding.reshape(1, -1)).squeeze()

        else:
            # Zero-pad
            padding = np.zeros(self.target_dim - current_dim)
```

```python
        return np.concatenate([embedding, padding])

    def fit_pca(self, sample_texts, n_samples=1000):
        """
        Fit PCA on a sample of embeddings from the dataset.
        Call this once before encoding the full dataset.
        """
        print(f"Fitting PCA for {self.model_name} on {n_samples} samples...")

        # Sample texts
        if len(sample_texts) > n_samples:
            import random
            sample_texts = random.sample(sample_texts, n_samples)

        # Extract embeddings (without dimension standardization)
        raw_embs = []
        for text in sample_texts:
            if self.model_name in ['simcse']:
                emb = self.model.encode([text], show_progress_bar=False)[0]
            elif self.model_name == 'llm2vec':
                emb = self.model.encode([text])[0]
            else:
                inputs = self.tokenizer(
                    text,
                    return_tensors='pt',
                    truncation=True,
                    max_length=512
                )
                if torch.cuda.is_available() and self.model_name not in ['llama3']:
                    inputs = {k: v.cuda() for k, v in inputs.items()}

                with torch.no_grad():
                    outputs = self.model(**inputs, output_hidden_states=True)
                    last_hidden = outputs.hidden_states[-1]
                    emb = last_hidden.mean(dim=1).squeeze(0).cpu().numpy()

            raw_embs.append(emb)

        raw_embs = np.stack(raw_embs)

        # Fit PCA if needed
        current_dim = raw_embs.shape[-1]
        if current_dim > self.target_dim:
            self.pca = PCA(n_components=self.target_dim)
```

```
        self.pca.fit(raw_embs)
        print(f"PCA fitted: {current_dim} → {self.target_dim} dims")
    else:
        print(f"No PCA needed: {current_dim} ≤ {self.target_dim}")

def save_pca(self, path):
    """Save fitted PCA transformer."""
    if self.pca is not None:
        with open(path, 'wb') as f:
            pickle.dump(self.pca, f)

def load_pca(self, path):
    """Load fitted PCA transformer."""
    with open(path, 'rb') as f:
        self.pca = pickle.load(f)
```

Usage Example

```python
# Initialize extractors for all models
extractors = {}
for model_name in ['bert', 'roberta', 'llama3', 'simcse', 'jina', 'llm2vec']:
    # Use quantization for LLaMA to save memory
    use_quant = (model_name == 'llama3')
    extractors[model_name] = EmbeddingExtractor(model_name, target_dim=768, use_quantization=use_quant)

# Fit PCA on sample (do this once at the start)
sample_texts = [item['title'] + '. ' + item['description'] for item in sample_items[:1000]]
for name, extractor in extractors.items():
    extractor.fit_pca(sample_texts)
    extractor.save_pca(f'data/pca_{name}.pkl')

# Now extract embeddings for full dataset
for item in all_items:
    text = f"{item['title']}. {item['description']}"
    for name, extractor in extractors.items():
        embedding = extractor.encode(text)  # Returns (768,) array
        # ... store or use embedding
```

Memory & Cost Optimizations (CRITICAL FOR 13-DAY TIMELINE)

Overview

Challenge: Limited compute budget ($0-50), tight timeline (13 days), large models (LLaMA: 8B params)

Solution: Aggressive memory optimization + free tier maximization + smart caching

A. Streaming Data Downloads (Avoid Loading 100GB into RAM)

python

```python
import gzip
import json
import requests
from tqdm import tqdm

def download_amazon_streaming(category='Electronics', n_items=10000, save_path='data/amazon_10k.json'):
    """
    Stream-download dataset without loading full file into memory.
    Only downloads what we need, then stops.

    Memory: O(n_items) instead of O(full_dataset)
    Time: ~5 min instead of ~30 min
    """
    url = f"https://amazon-reviews-2023.github.io/data/{category}_metadata.jsonl.gz"

    items = []

    # Stream download (don't load entire file)
    print(f"Streaming {category} dataset...")
    response = requests.get(url, stream=True)

    with gzip.open(response.raw, 'rt', encoding='utf-8') as f:
        for line_num, line in enumerate(tqdm(f, total=n_items, desc="Downloading")):
            if len(items) >= n_items:
                # STOP EARLY - don't download the rest!
                break

            try:
                item = json.loads(line)
                # Only keep fields we need (save memory)
                filtered_item = {
                    'item_id': item.get('asin', f'item_{line_num}'),
                    'title': item.get('title', ''),
                    'description': item.get('description', [''])[0] if isinstance(item.get('description'), list) else item.get('description', ''),
                    'category': item.get('main_category', category),
                    'price': item.get('price', 0.0),
                    'avg_rating': item.get('average_rating', 0.0)
                }
                items.append(filtered_item)
            except:
                continue

    # Save to disk immediately
```

```
    print(f"Saving {len(items)} items to {save_path}...")
    with open(save_path, 'w') as f:
        json.dump(items, f, indent=2)

    return items

# Usage: Download multiple categories efficiently
def download_multi_category_amazon(n_items_per_category=3333):
    """
    Download from multiple categories to get diversity.
    Total: 3 categories × 3333 items = ~10K items
    """
    categories = ['Electronics', 'Clothing_Shoes_and_Jewelry', 'Home_and_Kitchen']
    all_items = []

    for cat in categories:
        items = download_amazon_streaming(cat, n_items=n_items_per_category)
        all_items.extend(items)
        print(f"{cat}: {len(items)} items")

        # Clear memory
        del items
        import gc
        gc.collect()

    print(f"Total items: {len(all_items)}")
    return all_items
```

B. Batched Embedding Computation (Avoid OOM)

```
python
```

```python
def embed_dataset_batched(items, extractor, batch_size=32, cache_path=None):
    """
    Embed dataset in batches to avoid OOM.
    Save incrementally to disk.

    Memory: O(batch_size) instead of O(dataset_size)
    Enables processing 10K items on 8GB GPU
    """
    embeddings = {}

    # Extract texts
    texts = [f"{item['title']}. {item['description']}" for item in items]
    item_ids = [item['item_id'] for item in items]

    # Process in batches
    print(f"Embedding {len(texts)} texts in batches of {batch_size}...")
    for i in tqdm(range(0, len(texts), batch_size), desc="Batches"):
        batch_texts = texts[i:i+batch_size]
        batch_ids = item_ids[i:i+batch_size]

        # Embed batch
        try:
            batch_embs = extractor.encode(batch_texts)

            # Handle single vs multiple outputs
            if len(batch_texts) == 1:
                batch_embs = [batch_embs]

            # Store
            for item_id, emb in zip(batch_ids, batch_embs):
                embeddings[item_id] = emb

            # Save checkpoint every 1000 items (crash recovery)
            if (i + batch_size) % 1000 == 0 and cache_path:
                with open(cache_path + '.tmp', 'wb') as f:
                    pickle.dump(embeddings, f)

            # Clear GPU memory
            if torch.cuda.is_available():
                torch.cuda.empty_cache()

        except Exception as e:
            print(f"Error at batch {i}: {e}")
```

```
            continue

    # Final save
    if cache_path:
        with open(cache_path, 'wb') as f:
            pickle.dump(embeddings, f)
        print(f"Saved {len(embeddings)} embeddings to {cache_path}")

        # Remove temp file
        import os
        if os.path.exists(cache_path + '.tmp'):
            os.remove(cache_path + '.tmp')

    return embeddings

# Usage with caching
cache = EmbeddingCache()
for model_name in ['bert', 'roberta', 'llama3', 'simcse', 'jina', 'llm2vec']:
    # Check cache first
    cached = cache.load(model_name, 'amazon_10k')
    if cached is not None:
        print(f"{model_name}: Loaded from cache")
        continue

    # Compute embeddings in batches
    print(f"{model_name}: Computing embeddings...")
    extractor = extractors[model_name]
    embeddings = embed_dataset_batched(
        items=amazon_items,
        extractor=extractor,
        batch_size=32 if model_name != 'llama3' else 8,  # Smaller batches for LLaMA
        cache_path=f'data/embeddings/{model_name}_amazon_10k.pkl'
    )

    # Cache for future runs
    cache.save(embeddings, model_name, 'amazon_10k')

    # Clear memory before next model
    del extractor, embeddings
    torch.cuda.empty_cache()
    gc.collect()
```

C. Mixed Precision Training (Reduce Memory by 50%)

python

```python
from torch.cuda.amp import autocast, GradScaler

class TrainerWithMixedPrecision:
    """
    Wrapper that adds mixed precision to any training loop.

    Benefits:
    - 40-50% memory reduction
    - 2-3× faster training
    - Works on T4, V100, A100 GPUs
    """

    def __init__(self, model, optimizer, use_amp=True):
        self.model = model
        self.optimizer = optimizer
        self.use_amp = use_amp and torch.cuda.is_available()
        self.scaler = GradScaler() if self.use_amp else None

    def training_step(self, batch):
        """Single training step with automatic mixed precision."""
        self.optimizer.zero_grad()

        if self.use_amp:
            # Forward pass in FP16
            with autocast():
                loss = self.compute_loss(batch)

            # Backward pass with scaled gradients
            self.scaler.scale(loss).backward()

            # Unscale before clipping
            self.scaler.unscale_(self.optimizer)
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=1.0)

            # Optimizer step
            self.scaler.step(self.optimizer)
            self.scaler.update()
        else:
            # Standard FP32 training
            loss = self.compute_loss(batch)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=1.0)
            self.optimizer.step()
```

```
        return loss.item()

    def compute_loss(self, batch):
        """Override this in your training code."""
        raise NotImplementedError
```

D. Gradient Checkpointing for Transformers (40% Memory Reduction)

Already integrated in RewardTransformer class - just set `use_checkpointing=True` in constructor.

E. Free Tier Compute Strategy & Session Management

```python
```

```python
"""
Maximize free compute tiers to stay under $50 budget:

1. Google Colab Free (T4 GPU, 15GB VRAM, 12-hour sessions)
   - Good for: BERT, RoBERTa, SimCSE, Jina
   - Limit: 12 hours per session, disconnect if idle
   - Strategy: Save checkpoints to Google Drive every hour

2. Google Colab Pro ($10/month, A100 GPU, 40GB VRAM)
   - Only if LLaMA-3 needs more memory
   - Can avoid if using 4-bit quantization

3. Kaggle Notebooks (P100 GPU, 16GB VRAM, 30 hours/week free)
   - Backup if Colab quota exhausted
   - Supports same code as Colab

4. HuggingFace Spaces (CPU only, free)
   - For embedding computation only
   - Can precompute BERT/RoBERTa embeddings

Session planning for 13 days:
- Days 1-2: Colab (8 hours) - Precompute all embeddings with caching
- Days 3-6: Multiple Colab sessions - Train RecSys bandits
- Days 7-12: Multiple Colab sessions - Train Tools critics
- Always mount Google Drive, save checkpoints every hour
"""

class ColabSessionManager:
    """Auto-checkpoint manager for long Colab sessions."""

    def __init__(self, drive_mount='/content/drive'):
        self.drive_mount = drive_mount
        self.start_time = time.time()
        self.last_checkpoint = self.start_time

    def time_remaining(self):
        """Check remaining time in 12-hour session."""
        elapsed = (time.time() - self.start_time) / 3600
        remaining = 12 - elapsed
        if remaining < 1:
            print(f"⚠️  WARNING: Only {remaining*60:.0f} minutes remaining!")
        return remaining
```

```python
    def auto_checkpoint(self, model, path, interval_hours=1):
        """Auto-save every N hours."""
        current_time = time.time()

        if (current_time - self.last_checkpoint) / 3600 >= interval_hours:
            checkpoint_path = f"{self.drive_mount}/checkpoints/{path}"
            os.makedirs(os.path.dirname(checkpoint_path), exist_ok=True)
            torch.save(model.state_dict(), checkpoint_path)
            print(f"✓ Auto-checkpoint: {checkpoint_path}")
            self.last_checkpoint = current_time

# Usage
session = ColabSessionManager()
for epoch in range(n_epochs):
    train_one_epoch(model)
    session.auto_checkpoint(model, f"{model_name}_epoch{epoch}.pt")

    if session.time_remaining() < 0.5:
        print("⚠️ Session ending soon, stopping early")
        break
```

F. Early Stopping (Save 60% Training Time)

```python
```

```python
def train_with_early_stopping(model, train_data, val_data, max_epochs=50, patience=5):
    """
    Stop when validation loss plateaus.
    Typical savings: 50 epochs → 15 epochs = 70% time saved
    """
    best_val_loss = float('inf')
    patience_counter = 0
    best_model_state = None

    for epoch in range(max_epochs):
        train_loss = train_one_epoch(model, train_data)
        val_loss = evaluate(model, val_data)

        print(f"Epoch {epoch}: train={train_loss:.4f}, val={val_loss:.4f}")

        if val_loss < best_val_loss - 0.001:
            best_val_loss = val_loss
            patience_counter = 0
            best_model_state = model.state_dict().copy()
            print(f"  ✓ New best")
        else:
            patience_counter += 1
            print(f"  - No improvement ({patience_counter}/{patience})")

        if patience_counter >= patience:
            print(f"✓ Early stopping at epoch {epoch}")
            break

    model.load_state_dict(best_model_state)
    return model
```

Memory Budget Summary

```
Without optimizations:
- Full dataset in RAM: 2GB
- All embeddings in RAM: 8GB
- LLaMA-3-8B model: 16GB VRAM
- Training: 4GB VRAM
- Total: 30GB ❌ IMPOSSIBLE on free tier

With ALL optimizations:
- Streaming download: 100MB RAM
- Batched embeddings: 500MB RAM
```

```
- LLaMA-3 4-bit quant: 4GB VRAM
- Training with AMP + checkpointing: 2GB VRAM
- Total: 6.6GB ✅ FITS on free Colab T4 (15GB VRAM)

Target memory per use case:
- RecSys bandit: ~3GB VRAM ✅ Easy
- Tools A2C: ~5GB VRAM ✅ Manageable
- Math A2C: ~8GB VRAM ⚠️ Might need Colab Pro
```

Use Case 1: Neural Contextual Bandit (RecSys) - PRIORITY 1

Goal

Validate core theory (Section 5) by showing contrastive embeddings achieve lower regret on contextual bandit problem.

Dataset: Amazon Product Recommendation

Source:

- Amazon Reviews 2023 (McAuley lab): https://amazon-reviews-2023.github.io/
- Categories: Electronics, Clothing, Home & Kitchen, Books, Sports
- **Subsample to 10,000 items** for speed

Data structure:

```python
{
    'item_id': 'B07XYZ123',
    'title': 'Wireless Bluetooth Headphones',
    'description': 'High-quality over-ear headphones with 30hr battery...',
    'category': 'Electronics',
    'price': 79.99,
    'avg_rating': 4.3,
    'image_url': 'https://...'  # Optional for multimodal
}
```

Download script:

```python
```

```python
import requests
import json
from tqdm import tqdm

def download_amazon_subset(category='Electronics', n_items=10000):
    """
    Download Amazon product metadata.
    """
    # Use 2023 dataset API
    url = f"https://amazon-reviews-2023.github.io/data/{category}_metadata.jsonl.gz"

    # Download and decompress
    response = requests.get(url, stream=True)
    items = []

    with gzip.open(response.raw, 'rt') as f:
        for line in tqdm(f, total=n_items):
            if len(items) >= n_items:
                break
            item = json.loads(line)
            items.append(item)

    return items

# Download multiple categories for diversity
categories = ['Electronics', 'Clothing', 'Home_and_Kitchen']
all_items = []
for cat in categories:
    items = download_amazon_subset(cat, n_items=3333)
    all_items.extend(items)

# Save
with open('data/amazon_10k.json', 'w') as f:
    json.dump(all_items, f)
```

Neural Thompson Sampling Architecture

Use existing implementation: https://github.com/ZeroWeight/NeuralTS/blob/master/learner_diag.py

Key modifications needed:

```
python
```

```python
# Original NeuralTS expects feature vectors as input
# We provide pre-computed embeddings

class NeuralTSBandit:
    def __init__(self, embedding_dim=768, hidden_dim=100, lambda_reg=1.0, nu=1.0):
        """
        Neural Thompson Sampling with diagonal approximation.

        Args:
            embedding_dim: Dimension of input embeddings (768)
            hidden_dim: Hidden layer width (100)
            lambda_reg: Regularization strength
            nu: Variance scaling
        """
        self.network = nn.Sequential(
            nn.Linear(embedding_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1)
        )
        self.lambda_reg = lambda_reg
        self.nu = nu
        self.U = torch.eye(self.num_params())  # Diagonal approximation

    def forward(self, embedding, compute_variance=True):
        """
        Compute mean and variance for Thompson Sampling.

        Args:
            embedding: (embedding_dim,) Pre-computed item embedding
            compute_variance: Whether to compute posterior variance

        Returns:
            mu: Mean reward prediction
            sigma: Standard deviation (if compute_variance=True)
        """
        # Mean prediction
        mu = self.network(embedding).squeeze()

        if not compute_variance:
            return mu, None

        # Compute gradient (Jacobian)
        grads = torch.autograd.grad(mu, self.network.parameters(),
```

```
                                    retain_graph=True, create_graph=True)
        grad_vec = torch.cat([g.view(-1) for g in grads])

        # Variance via neural tangent kernel
        # σ² = g^T U^{-1} g / m
        sigma_sq = (grad_vec @ torch.inverse(self.U) @ grad_vec) / hidden_dim
        sigma = torch.sqrt(self.lambda_reg * self.nu * sigma_sq)

        return mu, sigma

    def select_action(self, candidate_embeddings):
        """
        Thompson Sampling action selection.

        Args:
            candidate_embeddings: (K, embedding_dim) K candidate items

        Returns:
            selected_idx: Index of selected item
            sampled_rewards: (K,) Sampled rewards for all candidates
        """
        sampled_rewards = []

        for emb in candidate_embeddings:
            mu, sigma = self.forward(emb, compute_variance=True)
            # Sample from posterior
            reward_sample = torch.normal(mu, sigma)
            sampled_rewards.append(reward_sample.item())

        selected_idx = np.argmax(sampled_rewards)
        return selected_idx, sampled_rewards

    def update(self, embedding, reward, t):
        """
        Update network parameters via SGD.

        Args:
            embedding: Selected item embedding
            reward: Observed reward (0 or 1)
            t: Current round number
        """
        # Prediction
        pred, _ = self.forward(embedding, compute_variance=False)
```

```python
    # Loss with time-decaying regularization
    loss = (reward - pred)**2 + (self.lambda_reg / t) * sum(
        p.norm()**2 for p in self.network.parameters()
    )

    # SGD update
    optimizer = torch.optim.SGD(self.network.parameters(), lr=0.01)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Update U matrix (diagonal approximation)
    grads = torch.autograd.grad(pred, self.network.parameters(),
                                retain_graph=True)
    grad_vec = torch.cat([g.view(-1) for g in grads])
    self.U += torch.outer(grad_vec, grad_vec)
```

Experiment Setup

Reward model: Click-through rate (CTR) prediction

- User context: Previous 5 items viewed
- Action space: 10,000 Amazon products
- Reward: 1 if user clicks item, 0 otherwise

Simulation:

python

```python
def simulate_user_clicks(items, embedding_model, n_rounds=10000, K_candidates=50):
    """
    Simulate user click behavior for bandit evaluation.

    Args:
        items: List of 10,000 Amazon products
        embedding_model: One of {BERT, RoBERTa, LLaMA, SimCSE, Jina, LLM2Vec}
        n_rounds: Number of interaction rounds
        K_candidates: Number of items to score per round

    Returns:
        cumulative_regret: (n_rounds,) Cumulative regret over time
        total_clicks: Number of successful recommendations
    """
    # Pre-compute all embeddings (CRITICAL BOTTLENECK - see below)
    print("Precomputing embeddings...")
    item_embeddings = {}
    for item in tqdm(items):
        text = f"{item['title']}. {item['description']}"
        emb = embedding_model.encode(text)
        emb = standardize_embedding_dim(emb, target_dim=768)
        item_embeddings[item['item_id']] = emb

    # Initialize bandit
    bandit = NeuralTSBandit(embedding_dim=768, hidden_dim=100)

    # Simulate user sessions
    cumulative_regret = []
    total_regret = 0

    for t in tqdm(range(n_rounds)):
        # Sample user context (random past items)
        context_items = random.sample(items, k=5)
        context_embs = [item_embeddings[item['item_id']] for item in context_items]
        context_emb = np.mean(context_embs, axis=0)  # Average context

        # Sample K candidates to consider
        candidates = random.sample(items, k=K_candidates)
        candidate_embs = torch.tensor([
            item_embeddings[item['item_id']] for item in candidates
        ])

        # Bandit selects action
```

```
        selected_idx, _ = bandit.select_action(candidate_embs)
        selected_item = candidates[selected_idx]

        # Simulate user click (reward)
        # True reward: based on category match + rating
        true_reward = compute_true_reward(selected_item, context_items)
        optimal_reward = max(compute_true_reward(c, context_items) for c in candidates)

        # Observe reward
        reward = true_reward
        regret = optimal_reward - true_reward
        total_regret += regret
        cumulative_regret.append(total_regret)

        # Update bandit
        selected_emb = candidate_embs[selected_idx]
        bandit.update(selected_emb, reward, t+1)

    return np.array(cumulative_regret), sum(rewards)

def compute_true_reward(item, context_items):
    """
    Ground truth reward function (unknown to bandit).

    Reward based on:
    - Category match with context (50% weight)
    - Item rating (30% weight)
    - Price appropriateness (20% weight)
    """
    # Category match
    context_categories = [c['category'] for c in context_items]
    category_match = 1.0 if item['category'] in context_categories else 0.0

    # Rating
    rating_score = item['avg_rating'] / 5.0

    # Price (prefer mid-range)
    price_score = 1.0 - abs(item['price'] - 50.0) / 100.0
    price_score = max(0, min(1, price_score))

    # Weighted combination
    reward = 0.5 * category_match + 0.3 * rating_score + 0.2 * price_score
```

```python
    # Binarize with probability = reward
    return 1 if random.random() < reward else 0
```

CRITICAL BOTTLENECK: Embedding Computation

Problem: 10,000 items × 6 models = 60,000 embedding computations

Solution: Aggressive caching

```python
```

```python
import os
import pickle
from pathlib import Path

class EmbeddingCache:
    """
    Cache embeddings to disk to avoid recomputation.
    """
    def __init__(self, cache_dir='data/embeddings'):
        self.cache_dir = Path(cache_dir)
        self.cache_dir.mkdir(exist_ok=True, parents=True)

    def get_cache_path(self, model_name, dataset_name):
        return self.cache_dir / f"{model_name}_{dataset_name}.pkl"

    def load(self, model_name, dataset_name):
        cache_path = self.get_cache_path(model_name, dataset_name)
        if cache_path.exists():
            print(f"Loading cached embeddings: {cache_path}")
            with open(cache_path, 'rb') as f:
                return pickle.load(f)
        return None

    def save(self, embeddings, model_name, dataset_name):
        cache_path = self.get_cache_path(model_name, dataset_name)
        print(f"Saving embeddings to cache: {cache_path}")
        with open(cache_path, 'wb') as f:
            pickle.dump(embeddings, f)

# Usage
cache = EmbeddingCache()

def get_embeddings(items, model_name, dataset_name='amazon_10k'):
    """
    Get embeddings with caching.
    """
    # Try cache first
    cached = cache.load(model_name, dataset_name)
    if cached is not None:
        return cached

    # Compute embeddings
    model = load_embedding_model(model_name)
```

```
    embeddings = {}

    for item in tqdm(items, desc=f"Embedding with {model_name}"):
        text = f"{item['title']}. {item['description']}"
        emb = model.encode(text)
        emb = standardize_embedding_dim(emb, target_dim=768)
        embeddings[item['item_id']] = emb

    # Cache for future runs
    cache.save(embeddings, model_name, dataset_name)

    return embeddings

# Precompute all embeddings once
models = ['bert', 'roberta', 'llama3', 'simcse', 'jina', 'llm2vec']
for model in models:
    get_embeddings(items, model)
```

Time estimate:

- 10K items × 6 models × ~0.1 sec/item = **~100 minutes total** (one-time cost)
- Cached retrieval: **~1 second per model**

Metrics

Primary:

1. **Cumulative regret** $\mathcal{R}_T = \sum_{t=1}^{T}(r_t^* - r_t)$
2. **Regret plots** over time (show convergence rate)

Secondary: 3. **Click-through rate** (CTR) - percentage of successful recommendations 4. **Regret gap** between models at T=10,000

Expected Results

```
python
```

```
# Predicted cumulative regret at T=10,000
results = {
    'BERT': 3500,        # High regret (anisotropic)
    'RoBERTa': 3200,     # High regret (anisotropic)
    'LLaMA-3': 3400,     # High regret (LLM still anisotropic!)
    'SimCSE': 1800,      # Low regret (contrastive)
    'Jina-v3': 1600,     # Lowest regret (SOTA contrastive)
    'LLM2Vec': 1750      # Low regret (contrastive-tuned LLM)
}
```

Key comparisons:

- SimCSE vs RoBERTa: ~44% regret reduction
- Jina vs BERT: ~54% regret reduction
- LLM2Vec vs LLaMA-3: ~48% regret reduction (validates that LLMs need contrastive tuning!)

Timeline: Days 1-6

- **Day 1:** Download Amazon data, set up embedding cache, precompute all embeddings
- **Days 2-3:** Implement NeuralTS bandit, test on single model
- **Days 4-5:** Run all 6 models × 5 random seeds = 30 experiments
- **Day 6:** Generate regret plots, compute statistics

Use Case 2: A2C Transformer for Tool Selection - PRIORITY 2

Goal

Show contrastive embeddings enable better sequential decision-making in multi-step tasks (validates Section 6).

Dataset: ToolBench

Source: ToolLLM GitHub: https://github.com/OpenBMB/ToolBench

Focus on I3-Instruction (cross-category tool chaining):

- Example task: "Book a flight to NYC and reserve a hotel near Times Square"
- Requires chaining tools across categories: Travel (flights) → Hospitality (hotels)
- ToolLLM's greedy DFS struggles here (~45-50% success)

Data structure:

```python
{
    'task_id': 'I3-1523',
    'instruction': 'Book a flight to NYC and reserve a hotel...',
    'required_tools': ['SearchFlights', 'BookFlight', 'HotelSearch', 'ReserveHotel'],
    'tool_categories': ['Travel', 'Travel', 'Hospitality', 'Hospitality'],
    'optimal_sequence': ['SearchFlights', 'BookFlight', 'HotelSearch', 'ReserveHotel'],
    'success_criteria': 'Flight booked AND hotel reserved'
}
```

Download script:

```bash
# Clone ToolBench repo
git clone https://github.com/OpenBMB/ToolBench
cd ToolBench

# Extract I3 tasks
python scripts/extract_i3_tasks.py --output data/toolbench_i3.json
```

Tool Database

16,464 APIs from ToolBench:

```python
{
    'tool_id': 'SearchFlights_v2',
    'name': 'Search Flights',
    'category': 'Travel',
    'description': 'Search for available flights between two cities on a given date',
    'parameters': ['origin', 'destination', 'date'],
    'example_usage': 'SearchFlights(origin="LAX", destination="JFK", date="2024-02-15")'
}
```

Precompute tool embeddings:

```python
```

```
def embed_tools(tools, model_name):
    """
    Embed tool descriptions for critic network.
    """
    cache = EmbeddingCache()
    cached = cache.load(model_name, 'toolbench_16k')
    if cached:
        return cached

    model = load_embedding_model(model_name)
    tool_embeddings = {}

    for tool in tqdm(tools, desc=f"Embedding tools with {model_name}"):
        # Concatenate all tool info
        text = f"{tool['name']}. {tool['description']}. Category: {tool['category']}"
        emb = model.encode(text)
        emb = standardize_embedding_dim(emb, target_dim=768)
        tool_embeddings[tool['tool_id']] = emb

    cache.save(tool_embeddings, model_name, 'toolbench_16k')
    return tool_embeddings
```

Time estimate: 16K tools × 6 models × 0.1 sec = **~160 minutes** (one-time cost)

A2C Architecture

Actor (Policy): Rule-based or small LLM that proposes next tool **Critic (Value):** Your 36M transformer over contrastive embeddings

```python
```

```python
import torch
import torch.nn as nn

class RewardTransformer(nn.Module):
    """
    Lightweight transformer critic for scoring tool sequences.

    Architecture: 36M parameters
    - Input: Query embedding + tool sequence embeddings
    - Output: Scalar value (quality of this sequence)
    """
    def __init__(self, d_model=768, nhead=8, num_layers=2, hidden_dim=768):
        super().__init__()

        # Query projection
        self.query_proj = nn.Linear(d_model, d_model)

        # Sequence encoder (self-attention over tool history)
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            dim_feedforward=hidden_dim,
            dropout=0.1,
            batch_first=True
        )
        self.sequence_encoder = nn.TransformerEncoder(
            encoder_layer,
            num_layers=num_layers
        )

        # Value head (predicts cumulative reward)
        self.value_head = nn.Sequential(
            nn.Linear(d_model * 2, d_model),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(d_model, 1)
        )

    def forward(self, query_emb, tool_sequence_embs):
        """
        Args:
            query_emb: (batch_size, 768) Task instruction embedding
            tool_sequence_embs: (batch_size, seq_len, 768) Tool sequence
```

```
        Returns:
            value: (batch_size, 1) Predicted cumulative reward
        """
        batch_size = query_emb.shape[0]

        # Project query context
        query_ctx = self.query_proj(query_emb)  # (batch, 768)

        # Encode tool sequence (self-attention)
        seq_encoding = self.sequence_encoder(tool_sequence_embs)  # (batch, seq_len, 768)

        # Pool to final state (last tool encoding)
        final_state = seq_encoding[:, -1, :]  # (batch, 768)

        # Combine query + final state
        combined = torch.cat([query_ctx, final_state], dim=-1)  # (batch, 1536)

        # Predict value
        value = self.value_head(combined)  # (batch, 1)

        return value

# Parameter count check
model = RewardTransformer()
total_params = sum(p.numel() for p in model.parameters())
print(f"Total parameters: {total_params / 1e6:.1f}M")  # Should be ~36M
```

Policy (Actor):

```
python
```

```python
class ToolPolicy:
    """
    Simple policy that proposes candidate next tools.

    Options:
    1. Rule-based: BM25 keyword matching (like ToolLLM)
    2. LLM-based: Small model (Mistral-7B) generates suggestions

    For simplicity, use rule-based.
    """
    def __init__(self, tools):
        self.tools = tools
        # Build BM25 index
        from rank_bm25 import BM25Okapi
        corpus = [f"{t['name']} {t['description']}" for t in tools]
        tokenized_corpus = [doc.split() for doc in corpus]
        self.bm25 = BM25Okapi(tokenized_corpus)

    def propose_tools(self, task_instruction, tool_history, n_candidates=10):
        """
        Propose n candidate next tools.

        Args:
            task_instruction: "Book a flight to NYC and reserve a hotel..."
            tool_history: ['SearchFlights', 'BookFlight']
            n_candidates: Number of tools to propose

        Returns:
            candidate_tools: List of n tool objects
        """
        # BM25 score based on instruction + what's left to do
        query = task_instruction + " " + " ".join(tool_history)
        tokenized_query = query.split()
        scores = self.bm25.get_scores(tokenized_query)

        # Remove already-used tools
        used_ids = [t['tool_id'] for t in tool_history]
        available_tools = [t for t in self.tools if t['tool_id'] not in used_ids]
        available_scores = [scores[i] for i, t in enumerate(self.tools)
                            if t['tool_id'] not in used_ids]

        # Top-n candidates
        top_indices = np.argsort(available_scores)[-n_candidates:]
```

```
    candidates = [available_tools[i] for i in top_indices]

    return candidates
```

A2C Training Loop

python

```
def train_a2c_toolbench(tasks, tools, tool_embeddings, model_name, n_epochs=20):
    """
    Train A2C agent on ToolBench tasks.

    Args:
        tasks: I3 instruction tasks (training split)
        tools: Full tool database (16K tools)
        tool_embeddings: Pre-computed embeddings for all tools
        model_name: Which embedding model to use
        n_epochs: Number of training epochs

    Returns:
        trained_critic: Trained RewardTransformer
        training_stats: Dict of metrics
    """
    # Initialize
    critic = RewardTransformer()
    policy = ToolPolicy(tools)
    optimizer = torch.optim.Adam(critic.parameters(), lr=1e-4)

    # Training loop
    for epoch in range(n_epochs):
        epoch_rewards = []
        epoch_values = []

        for task in tqdm(tasks, desc=f"Epoch {epoch+1}/{n_epochs}"):
            # Embed task instruction
            task_emb = tool_embeddings['task'][task['task_id']]  # Pre-cached

            # Initialize episode
            tool_history = []
            tool_sequence_embs = []
            rewards = []
            values = []
            log_probs = []

            max_steps = 10
            for step in range(max_steps):
                # Policy proposes candidates
                candidates = policy.propose_tools(
                    task['instruction'],
                    tool_history,
                    n_candidates=10
```

```
    )

    # Embed candidates
    candidate_embs = torch.stack([
        torch.tensor(tool_embeddings[c['tool_id']])
        for c in candidates
    ])  # (10, 768)

    # Critic scores each candidate
    candidate_scores = []
    for c_emb in candidate_embs:
        seq = tool_sequence_embs + [c_emb]
        seq_tensor = torch.stack(seq).unsqueeze(0)  # (1, seq_len, 768)
        task_tensor = torch.tensor(task_emb).unsqueeze(0)  # (1, 768)

        value = critic(task_tensor, seq_tensor)
        candidate_scores.append(value.item())

    # Select action (softmax over critic scores)
    scores_tensor = torch.tensor(candidate_scores)
    probs = torch.softmax(scores_tensor / 0.1, dim=0)  # Temperature = 0.1
    action_idx = torch.multinomial(probs, 1).item()

    selected_tool = candidates[action_idx]
    log_prob = torch.log(probs[action_idx])

    # Execute tool (simulated)
    reward = execute_tool_simulation(selected_tool, task, tool_history)

    # Store trajectory
    tool_history.append(selected_tool)
    tool_sequence_embs.append(candidate_embs[action_idx])
    rewards.append(reward)
    log_probs.append(log_prob)

    # Check if task completed
    if check_task_complete(task, tool_history):
        # Success bonus
        rewards[-1] += 10.0
        break

# Compute returns (discounted cumulative rewards)
returns = []
G = 0
```

```
        for r in reversed(rewards):
            G = r + 0.99 * G
            returns.insert(0, G)
        returns = torch.tensor(returns)

        # Compute advantages
        # Re-compute values for trajectory
        trajectory_values = []
        for i in range(len(tool_sequence_embs)):
            seq = torch.stack(tool_sequence_embs[:i+1]).unsqueeze(0)
            task_tensor = torch.tensor(task_emb).unsqueeze(0)
            value = critic(task_tensor, seq)
            trajectory_values.append(value)

        trajectory_values = torch.cat(trajectory_values)
        advantages = returns - trajectory_values.detach()

        # Actor loss (policy gradient with advantage)
        log_probs = torch.stack(log_probs)
        actor_loss = -(log_probs * advantages).mean()

        # Critic loss (MSE between predicted value and return)
        critic_loss = ((trajectory_values - returns) ** 2).mean()

        # Total loss
        loss = actor_loss + 0.5 * critic_loss

        # Update
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(critic.parameters(), 1.0)
        optimizer.step()

        # Stats
        epoch_rewards.append(sum(rewards))
        epoch_values.append(trajectory_values.mean().item())

    # Epoch summary
    print(f"Epoch {epoch+1}: Avg Reward = {np.mean(epoch_rewards):.2f}, "
          f"Avg Value = {np.mean(epoch_values):.2f}")

    return critic, {'rewards': epoch_rewards, 'values': epoch_values}

def execute_tool_simulation(tool, task, tool_history):
```

```
    """
    Simulate tool execution and return reward.

    Reward structure:
    - +1 if tool is in task's required_tools
    - +2 if tool is in correct position in sequence
    - -0.5 if tool is redundant (already used)
    - -1 if tool is irrelevant (wrong category)
    """
    if tool['tool_id'] in [t['tool_id'] for t in tool_history]:
        return -0.5  # Redundant

    if tool['tool_id'] in task['required_tools']:
        # Check if in correct position
        current_position = len(tool_history)
        optimal_position = task['optimal_sequence'].index(tool['tool_id'])

        if current_position == optimal_position:
            return 2.0  # Perfect placement
        else:
            return 1.0  # Right tool, wrong timing

    # Check category match
    task_categories = set(task['tool_categories'])
    if tool['category'] in task_categories:
        return 0.5  # Relevant category
    else:
        return -1.0  # Irrelevant

def check_task_complete(task, tool_history):
    """
    Check if all required tools have been used.
    """
    used_ids = [t['tool_id'] for t in tool_history]
    required_ids = task['required_tools']
    return all(req in used_ids for req in required_ids)
```

CRITICAL BOTTLENECK: Tool Execution

Problem: Real API calls are slow and rate-limited

Solution: Use cached execution results from ToolLLM paper

```
python
```

```python
class ToolExecutionCache:
    """
    Load pre-cached tool execution results from ToolLLM.

    ToolLLM authors ran all tools and cached results.
    We reuse their cache for fast simulation.
    """
    def __init__(self, cache_path='data/toolbench_cache.json'):
        with open(cache_path, 'r') as f:
            self.cache = json.load(f)

    def execute(self, tool_id, parameters, task_id):
        """
        Look up cached execution result.
        """
        key = f"{tool_id}_{task_id}"
        if key in self.cache:
            return self.cache[key]
        else:
            # Default: return empty result
            return {'status': 'not_cached', 'result': None}
```

Evaluation on I3 Tasks

python

```python
def evaluate_toolbench(critic, tasks, tools, tool_embeddings, model_name):
    """
    Evaluate trained critic on held-out I3 tasks.

    Metrics:
    - Success rate (% tasks completed)
    - Avg tools used (efficiency)
    - Tool diversity (unique categories)
    """
    policy = ToolPolicy(tools)

    successes = 0
    total_tools_used = []
    categories_explored = []

    for task in tqdm(tasks, desc=f"Evaluating {model_name}"):
        # Run episode (greedy evaluation, no exploration)
        task_emb = tool_embeddings['task'][task['task_id']]
        tool_history = []
        tool_sequence_embs = []

        max_steps = 10
        for step in range(max_steps):
            # Propose candidates
            candidates = policy.propose_tools(task['instruction'], tool_history)

            # Critic scores
            candidate_embs = torch.stack([
                torch.tensor(tool_embeddings[c['tool_id']])
                for c in candidates
            ])

            # Select greedily (no sampling)
            best_score = -float('inf')
            best_tool = None

            for c, c_emb in zip(candidates, candidate_embs):
                seq = tool_sequence_embs + [c_emb]
                seq_tensor = torch.stack(seq).unsqueeze(0)
                task_tensor = torch.tensor(task_emb).unsqueeze(0)

                value = critic(task_tensor, seq_tensor)
```

```
            if value.item() > best_score:
                best_score = value.item()
                best_tool = c
                best_emb = c_emb

        # Execute best tool
        tool_history.append(best_tool)
        tool_sequence_embs.append(best_emb)

        # Check completion
        if check_task_complete(task, tool_history):
            successes += 1
            break

    # Stats
    total_tools_used.append(len(tool_history))
    categories = set(t['category'] for t in tool_history)
    categories_explored.append(len(categories))

# Summary metrics
success_rate = successes / len(tasks)
avg_tools = np.mean(total_tools_used)
avg_categories = np.mean(categories_explored)

print(f"\n{model_name} Results:")
print(f"  I3 Success Rate: {success_rate:.1%}")
print(f"  Avg Tools Used: {avg_tools:.2f}")
print(f"  Avg Categories: {avg_categories:.2f}")

return {
    'success_rate': success_rate,
    'avg_tools': avg_tools,
    'avg_categories': avg_categories
}
```

Expected Results

```
python
```

```
# Predicted I3 success rates
results = {
    'ToolLLM (original)': 0.47,    # From their paper
    'BERT': 0.52,                  # Slight improvement (learned critic)
    'RoBERTa': 0.56,               # Better anisotropic baseline
    'LLaMA-3': 0.54,               # LLM still anisotropic
    'SimCSE': 0.68,                # Major jump (contrastive)
    'Jina-v3': 0.72,               # Best (SOTA contrastive)
    'LLM2Vec': 0.70                # Validates contrastive tuning
}
```

Key narrative:

- Contrastive embeddings (SimCSE, Jina, LLM2Vec) enable cross-category reasoning
- Anisotropic embeddings (BERT, RoBERTa, LLaMA) struggle to discover complementary tool chains
- LLM2Vec shows that even LLMs need contrastive tuning for RL tasks

Timeline: Days 7-12

- **Day 7:** Download ToolBench, precompute tool embeddings (all 6 models)
- **Days 8-9:** Implement A2C with transformer critic, test on single model
- **Days 10-11:** Train all 6 models (20 epochs each × 6 models = ~24 hours compute)
- **Day 12:** Evaluate on I3 test set, generate comparison table

---

Use Case 3: A2C Transformer for Math Reasoning - OPTIONAL (if time permits)

Goal

Show method generalizes to mathematical reasoning tasks (validates Section 6 in different domain).

Dataset: GSM8K & MATH

GSM8K (training): 8.5K grade-school math problems **MATH-500 (testing):** 500 hardest competition problems

Example:

```python
```

```
{
    'problem': 'Janet's ducks lay 16 eggs per day. She eats three for breakfast '
               'every morning and bakes muffins for her friends every day with four. '
               'She sells the remainder at the farmers\' market daily for $2 per fresh '
               'duck egg. How much in dollars does she make every day at the farmers\' market?',
    'solution': 'Step 1: Eggs consumed = 3 (breakfast) + 4 (muffins) = 7...',
    'answer': '18'
}
```

Download:

```python
from datasets import load_dataset

# GSM8K
gsm8k = load_dataset('gsm8k', 'main')
train_problems = gsm8k['train']  # 7,473 problems
test_problems = gsm8k['test']    # 1,319 problems

# MATH
math_dataset = load_dataset('hendrycks/math')
# Filter to hardest 500
math_500 = [p for p in math_dataset['test'] if p['level'] >= 4][:500]
```

Architecture (Same as Tools)

Critic: RewardTransformer (36M params, reuse from tools) **Policy:** Small LLM generates next reasoning steps

- Use Mistral-7B or LLaMA-3-8B
- Prompt: "Given problem and current steps, propose next step"

Key Differences from Tools:

Reasoning steps instead of tools:

```python
```

```
# Tool selection
sequence = [tool1_emb, tool2_emb, tool3_emb]

# Math reasoning
sequence = [step1_emb, step2_emb, step3_emb]
# Where each step is text like "Calculate 3 + 4 = 7"
```

Reward signal:

- Intermediate: +1 if step moves toward solution (hard to judge automatically)
- Terminal: +10 if final answer is correct (easy to verify)

Training: Monte Carlo Returns

```python
```

```python
def train_a2c_math(problems, step_embeddings, model_name, n_epochs=30):
    """
    Train A2C on math problems.

    Similar to tool training, but:
    - Policy is LLM generating reasoning steps
    - Reward is answer correctness
    """
    critic = RewardTransformer()
    policy_llm = load_llm('mistral-7b')  # For step generation

    for epoch in range(n_epochs):
        for problem in tqdm(problems):
            # Generate reasoning trajectory
            trajectory = generate_solution(problem, policy_llm, critic, step_embeddings)

            # Check final answer
            if trajectory['final_answer'] == problem['answer']:
                # Success! Reward = 10
                trajectory['rewards'][-1] += 10.0

            # Compute returns and advantages
            returns = compute_returns(trajectory['rewards'])
            values = [critic(problem_emb, seq_emb) for seq_emb in trajectory['sequences']]
            advantages = returns - values

            # Update critic
            loss = ((values - returns) ** 2).mean()
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

CRITICAL BOTTLENECK: LLM Policy

Problem: Generating reasoning steps requires running LLM many times

Solutions:

1. **Use smaller LLM:** Mistral-7B instead of 70B (10× faster)
2. **Batch generation:** Generate 5 candidate steps at once
3. **Cached common steps:** Pre-generate steps for common problem types

Time estimate:

- 1 problem = 10 steps × 0.5 sec/step = 5 sec
- 7K problems × 5 sec = **~10 hours per epoch**
- 30 epochs = **300 hours** (too slow!)

Realistic approach:

- Train on **2K problem subset** instead of full 7K
- Reduce to **15 epochs**
- Total: 2K × 5 sec × 15 = **~42 hours** (more feasible)

Expected Results

```python
# Predicted MATH-500 solve rates
results = {
    'rStar-Math (reference)': 0.90,  # From their paper (1.5B critic, 100 rollouts)
    'BERT': 0.58,                   # Anisotropic baseline
    'RoBERTa': 0.62,                # Better anisotropic
    'LLaMA-3': 0.60,                # LLM anisotropic
    'SimCSE': 0.68,                 # Contrastive
    'Jina-v3': 0.71,                # Best contrastive
    'LLM2Vec': 0.69                 # Contrastive-tuned LLM
}

# Avg rollouts before solve
rollouts = {
    'rStar-Math': 100,
    'RoBERTa': 95,
    'SimCSE': 68,     # 30% reduction (your key claim!)
    'Jina-v3': 65
}
```

Timeline: Days 13-15 (ONLY if on schedule)

- **Day 13:** Download datasets, implement math-specific components
- **Days 14-15:** Train subset (2K problems), evaluate on MATH-500 subset

SKIP THIS if behind schedule - RecSys + Tools is sufficient for paper.

Embedding Analysis (Theory Validation) - PRIORITY 0 (DO FIRST!)

Goal

Empirically validate theoretical claims from Sections 3-4.

Measurements

1. Eigenvalue Spectra

```python
```

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eigh

def compute_eigenvalue_spectrum(embeddings):
    """
    Compute eigenvalues of embedding covariance matrix.

    Args:
        embeddings: (N, d) array where N = number of items, d = 768

    Returns:
        eigenvalues: (d,) sorted in descending order
        eigenvectors: (d, d)
    """
    # Center embeddings
    embeddings_centered = embeddings - embeddings.mean(axis=0)

    # Compute covariance
    N = embeddings.shape[0]
    Sigma = (embeddings_centered.T @ embeddings_centered) / N

    # Eigendecomposition
    eigenvalues, eigenvectors = eigh(Sigma)

    # Sort descending
    idx = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]

    return eigenvalues, eigenvectors

# Compute for all models
spectra = {}
for model_name in ['bert', 'roberta', 'llama3', 'simcse', 'jina', 'llm2vec']:
    # Load cached embeddings
    embeddings = load_embeddings(model_name, 'amazon_10k')
    emb_matrix = np.stack(list(embeddings.values()))  # (10000, 768)

    eigenvalues, _ = compute_eigenvalue_spectrum(emb_matrix)
    spectra[model_name] = eigenvalues

# Plot
```

```
plt.figure(figsize=(10, 6))
for model_name, eigs in spectra.items():
    plt.plot(np.log(eigs), label=model_name, linewidth=2)

plt.xlabel('Dimension Index')
plt.ylabel('Log Eigenvalue')
plt.title('Eigenvalue Spectra: Anisotropic vs Contrastive Embeddings')
plt.legend()
plt.grid(alpha=0.3)
plt.savefig('results/eigenvalue_spectra.pdf', dpi=300, bbox_inches='tight')
```

Expected plot:

- BERT, RoBERTa, LLaMA: Sharp drop-off after dimension ~50-60
- SimCSE, Jina, LLM2Vec: Gentle decay through dimension ~200-220

Key for paper: This is Figure 1 - directly validates Section 3.2 assumptions.

2. Effective Dimension (Participation Ratio)

```
python
```

```python
def compute_participation_ratio(eigenvalues):
    """
    d_eff = (sum λ_i)² / (sum λ_i²)

    Measures "how many dimensions are effectively used"
    """
    return (eigenvalues.sum() ** 2) / (eigenvalues ** 2).sum()

# Compute for all models
deff_results = {}
for model_name, eigs in spectra.items():
    deff = compute_participation_ratio(eigs)
    deff_results[model_name] = deff
    print(f"{model_name:12s}: d_eff = {deff:.1f}")

# Expected output:
# bert        : d_eff = 42.3
# roberta     : d_eff = 51.7
# llama3      : d_eff = 58.2
# simcse      : d_eff = 203.5
# jina        : d_eff = 218.9
# llm2vec     : d_eff = 209.4
```

Table for paper:

Model	**Type**	d_{eff}
BERT	Anisotropic	42
RoBERTa	Anisotropic	52
LLaMA-3	Anisotropic	58
SimCSE	Contrastive	204
Jina-v3	Contrastive	219
LLM2Vec	Contrastive	209

Validates: Lemma 4.2 (uniformity maximizes d_{eff})

3. RKHS Norms of Learned Reward Functions

python

```python
def compute_rkhs_norm(reward_weights, eigenvalues):
    """
    ||R||² = Σ (w_i² / λ_i)

    Args:
        reward_weights: (768,) learned weights from reward network
        eigenvalues: (768,) from embedding covariance

    Returns:
        rkhs_norm: scalar
    """
    # Project weights onto eigenbasis first (if needed)
    # For simplicity, assume weights are already in eigenbasis

    rkhs_norm_squared = np.sum(reward_weights**2 / (eigenvalues + 1e-10))
    return np.sqrt(rkhs_norm_squared)

# Extract reward weights from trained bandits/critics
def extract_reward_weights(trained_model):
    """
    Extract final layer weights from neural network.

    For NeuralTS: final layer maps (hidden_dim,) -> (1,)
    For A2C critic: value head final layer
    """
    if isinstance(trained_model, NeuralTSBandit):
        # Get final linear layer
        final_layer = trained_model.network[-1]
        weights = final_layer.weight.data.cpu().numpy()  # (1, hidden_dim)
        return weights.squeeze()
    elif isinstance(trained_model, RewardTransformer):
        # Get value head final layer
        final_layer = trained_model.value_head[-1]
        weights = final_layer.weight.data.cpu().numpy()
        return weights.squeeze()

# Compute RKHS norms after training
rkhs_norms = {}
for model_name in models:
    # Load trained bandit/critic
    trained_model = torch.load(f'checkpoints/{model_name}_bandit.pt')

    # Extract weights
```

```
    weights = extract_reward_weights(trained_model)

    # Get eigenvalues for this embedding
    eigs = spectra[model_name]

    # Compute RKHS norm
    norm = compute_rkhs_norm(weights, eigs)
    rkhs_norms[model_name] = norm
    print(f"{model_name:12s}: ||R|| = {norm:.2f}")

# Expected output:
# bert        : ||R|| = 324.51
# roberta     : ||R|| = 287.34
# llama3      : ||R|| = 298.67
# simcse      : ||R|| = 8.42
# jina        : ||R|| = 6.78
# llm2vec     : ||R|| = 7.91
```

Validates: Lemma 3.2 (when $m > d_{\text{eff}}$, RKHS norm explodes)

Key insight: Anisotropic embeddings force reward functions into high-RKHS-norm regime.

4. Intrinsic Dimensionality of Rewards

```
python
```

```
def estimate_reward_dimensionality(rewards, embeddings, epsilon=0.01):
    """
    Find minimum m such that rewards can be approximated
    in m-dimensional subspace with error < epsilon.

    Validates Assumption 3.1.

    Args:
        rewards: (N,) observed rewards for N items
        embeddings: (N, 768) embeddings for those items
        epsilon: Approximation tolerance

    Returns:
        m: Minimum intrinsic dimensionality
    """
    # PCA on embeddings weighted by rewards
    weighted_embs = embeddings * rewards[:, np.newaxis]

    # SVD
    U, S, Vt = np.linalg.svd(weighted_embs, full_matrices=False)

    # Find m where cumulative variance > (1 - epsilon)
    total_var = np.sum(S**2)
    cumulative_var = np.cumsum(S**2) / total_var

    m = np.argmax(cumulative_var > (1 - epsilon)) + 1

    print(f"Reward intrinsic dimensionality: m = {m}")
    print(f"Explains {cumulative_var[m-1]:.1%} of variance")

    return m

# Test on learned rewards
for model_name in models:
    print(f"\n{model_name}:")

    # Get embeddings
    embeddings = load_embeddings(model_name, 'amazon_10k')
    emb_matrix = np.stack(list(embeddings.values()))

    # Get learned rewards (from trained bandit)
    trained_model = torch.load(f'checkpoints/{model_name}_bandit.pt')
    rewards = []
```

```
    for emb in emb_matrix:
        with torch.no_grad():
            pred, _ = trained_model.forward(torch.tensor(emb))
        rewards.append(pred.item())
    rewards = np.array(rewards)

    # Estimate m
    m = estimate_reward_dimensionality(rewards, emb_matrix, epsilon=0.01)

# Expected output:
# All models should find m ≈ 150-200
# This validates that realistic reward functions need ~200 dimensions
```

Validates: Assumption 3.1 (rewards have intrinsic dimensionality $m \in [100, 200]$)

5. Coverage Metric ϱ(k,K)

```
python
```

```python
def compute_coverage_metric(embeddings, k_values=[10, 50, 100, 500, 1000], n_trials=100):
    """
    ϱ(k,K) = Expected minimum distance from unsampled items to k-sample.

    Lower ϱ = better coverage = uniformity.

    Args:
        embeddings: (K, 768) All item embeddings
        k_values: List of sample sizes to test
        n_trials: Number of Monte Carlo trials per k

    Returns:
        rho_curves: Dict mapping k -> ϱ(k,K)
    """
    from scipy.spatial.distance import cosine

    K = len(embeddings)
    rho_curves = {k: [] for k in k_values}

    for k in k_values:
        distances_all_trials = []

        for trial in range(n_trials):
            # Sample k items
            sample_indices = np.random.choice(K, k, replace=False)
            sample_embs = embeddings[sample_indices]

            # For remaining items, compute distance to nearest sampled item
            remaining_indices = np.setdiff1d(np.arange(K), sample_indices)

            for idx in remaining_indices:
                emb = embeddings[idx]
                # Min distance to sample
                min_dist = min(cosine(emb, s) for s in sample_embs)
                distances_all_trials.append(min_dist)

        # Average over trials
        rho_curves[k] = np.mean(distances_all_trials)

    return rho_curves

# Compute for all models
coverage_results = {}
```

```
for model_name in models:
    embeddings = load_embeddings(model_name, 'amazon_10k')
    emb_matrix = np.stack(list(embeddings.values()))

    rho = compute_coverage_metric(emb_matrix)
    coverage_results[model_name] = rho

    print(f"{model_name}: ϱ(100, 10K) = {rho[100]:.4f}")

# Plot
plt.figure(figsize=(10, 6))
k_values = [10, 50, 100, 500, 1000]
for model_name in models:
    rho_vals = [coverage_results[model_name][k] for k in k_values]
    plt.plot(k_values, rho_vals, marker='o', label=model_name, linewidth=2)

plt.xlabel('Sample Size (k)')
plt.ylabel('Coverage ϱ(k,K)')
plt.title('Exploration Coverage: Lower is Better')
plt.legend()
plt.grid(alpha=0.3)
plt.xscale('log')
plt.savefig('results/coverage_metric.pdf', dpi=300, bbox_inches='tight')
```

Expected plot:

- Anisotropic (BERT, RoBERTa, LLaMA): Higher ϱ (worse coverage)
- Contrastive (SimCSE, Jina, LLM2Vec): Lower ϱ (better coverage)

Validates: That uniformity enables better exploration (fewer samples needed to cover space)

Timeline: Days 1-2

- **Day 1:** Compute eigenvalue spectra, d_{eff}, generate plots
- **Day 2:** Compute RKHS norms (requires trained models - do after RecSys), coverage metric

---

Code Structure

```
contrastive_rl_icml2026/
├── README.md
├── requirements.txt
```

```
├── setup.py
├── config/
│   ├── embeddings.yaml       # Model paths, dimensions
│   ├── recsys_config.yaml
│   ├── tools_config.yaml
│   └── math_config.yaml
├── data/
│   ├── amazon/
│   │   ├── raw/             # Downloaded data
│   │   └── processed/       # Cleaned, subsampled
│   ├── toolbench/
│   │   ├── tools.json       # 16K tool database
│   │   ├── i3_train.json
│   │   └── i3_test.json
│   ├── math/
│   │   ├── gsm8k/
│   │   └── math500/
│   └── embeddings/          # CACHED EMBEDDINGS (critical!)
│       ├── bert_amazon_10k.pkl
│       ├── roberta_amazon_10k.pkl
│       ├── llama3_amazon_10k.pkl
│       ├── simcse_amazon_10k.pkl
│       ├── jina_amazon_10k.pkl
│       ├── llm2vec_amazon_10k.pkl
│       ├── bert_toolbench_16k.pkl
│       └── ... (all combinations)
├── src/
│   ├── __init__.py
│   ├── embeddings/
│   │   ├── __init__.py
│   │   ├── base.py          # Abstract embedding interface
│   │   ├── bert.py
│   │   ├── roberta.py
│   │   ├── llama.py
│   │   ├── simcse.py
│   │   ├── jina.py
│   │   ├── llm2vec.py
│   │   ├── cache.py         # EmbeddingCache class
│   │   └── utils.py         # standardize_embedding_dim()
│   ├── models/
│   │   ├── __init__.py
│   │   ├── neural_ts.py      # NeuralTSBandit
│   │   ├── reward_transformer.py  # RewardTransformer (A2C critic)
│   │   ├── tool_policy.py    # ToolPolicy (actor)
```

```
│   │   └── math_policy.py    # LLM-based reasoning policy
│   ├── training/
│   │   ├── __init__.py
│   │   ├── bandit_trainer.py  # Train NeuralTS
│   │   ├── a2c_trainer.py     # Train A2C
│   │   └── utils.py           # Compute returns, advantages
│   ├── evaluation/
│   │   ├── __init__.py
│   │   ├── bandit_eval.py    # Evaluate regret
│   │   ├── tools_eval.py     # Evaluate I3 success rate
│   │   └── math_eval.py      # Evaluate solve rate
│   ├── analysis/
│   │   ├── __init__.py
│   │   ├── eigenvalues.py    # Compute spectra, d_eff
│   │   ├── rkhs.py           # Compute RKHS norms
│   │   ├── coverage.py       # Compute ϱ(k,K)
│   │   └── intrinsic_dim.py  # Estimate reward dimensionality
│   └── utils/
│       ├── __init__.py
│       ├── data_loader.py    # Download, preprocess datasets
│       ├── metrics.py        # Regret, success rate, diversity
│       └── visualization.py  # Plot generation
├── experiments/
│   ├── 01_embedding_analysis.py    # Priority 0
│   ├── 02_recsys_bandit.py         # Priority 1
│   ├── 03_tools_a2c.py             # Priority 2
│   └── 04_math_a2c.py              # Priority 3 (optional)
├── scripts/
│   ├── download_data.sh           # Automated dataset download
│   ├── precompute_embeddings.py    # Compute all embeddings once
│   └── run_all_experiments.sh      # End-to-end pipeline
├── results/
│   ├── plots/
│   │   ├── eigenvalue_spectra.pdf
│   │   ├── regret_curves.pdf
│   │   ├── coverage_metric.pdf
│   │   └── ... (all figures for paper)
│   ├── metrics/
│   │   ├── recsys_results.json
│   │   ├── tools_results.json
│   │   └── math_results.json
│   └── checkpoints/
│       ├── bert_bandit.pt
│       ├── simcse_critic_tools.pt
```

```
│       └── ...
└── paper/
    ├── figures/          # Final figures for ICML submission
    ├── tables/           # LaTeX tables
    └── draft.tex         # Paper draft
```

Dependencies & Environment Setup

Requirements

```
txt
```

```
# requirements.txt

# Core ML
torch>=2.0.0
transformers>=4.35.0
sentence-transformers>=2.2.0
accelerate>=0.25.0

# Embeddings
openai-clip>=1.0.1        # For CLIP (if using multimodal)

# Data processing
datasets>=2.14.0
numpy>=1.24.0
pandas>=2.0.0
scipy>=1.10.0
scikit-learn>=1.3.0

# RL utilities
gym>=0.26.0               # For environment interface

# Search & ranking
rank-bm25>=0.2.2          # For BM25 tool search

# Utilities
tqdm>=4.65.0
pyyaml>=6.0
requests>=2.31.0

# Visualization
matplotlib>=3.7.0
seaborn>=0.12.0

# Optional (for math)
sympy>=1.12               # Math verification
```

Installation

```
bash
```

```
# Create environment
conda create -n contrastive_rl python=3.10
conda activate contrastive_rl

# Install PyTorch (CUDA 11.8 or CPU)
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118

# Install requirements
pip install -r requirements.txt

# Download model checkpoints
python scripts/download_models.py
```

Model Downloads

```
python
```

```python
# scripts/download_models.py

from transformers import AutoModel, AutoTokenizer
from sentence_transformers import SentenceTransformer

def download_all_models():
    """
    Pre-download all embedding models to avoid runtime delays.
    """
    print("Downloading embedding models...")

    # BERT
    print("  - BERT...")
    AutoModel.from_pretrained('bert-base-uncased')
    AutoTokenizer.from_pretrained('bert-base-uncased')

    # RoBERTa
    print("  - RoBERTa...")
    AutoModel.from_pretrained('roberta-base')
    AutoTokenizer.from_pretrained('roberta-base')

    # LLaMA-3 (requires HuggingFace token)
    print("  - LLaMA-3 (requires auth)...")
    # AutoModel.from_pretrained('meta-llama/Meta-Llama-3-8B', use_auth_token=True)

    # SimCSE
    print("  - SimCSE...")
    SentenceTransformer('princeton-nlp/sup-simcse-bert-base-uncased')

    # Jina
    print("  - Jina-v3...")
    AutoModel.from_pretrained('jinaai/jina-embeddings-v3')

    # LLM2Vec (requires special handling)
    print("  - LLM2Vec...")
    # Download from their repo

    print("All models downloaded!")

if __name__ == '__main__':
    download_all_models()
```

Timeline & Compute Budget

Timeline (13 days remaining)

Phase 0: Embedding Analysis (Days 1-2)

- Day 1: Compute eigenvalue spectra, d_{eff}, make Figure 1
- Day 2: Compute coverage metric $\varrho(k,K)$, finalize theory validation plots

Phase 1: RecSys Neural Bandit (Days 3-6)

- Day 3: Download Amazon data, precompute all 6 embeddings (10K items × 6 models)
- Day 4: Implement NeuralTS bandit, test on 1 model
- Day 5: Run all 6 models × 5 seeds = 30 experiments (~12 hours compute)
- Day 6: Generate regret plots, RKHS norm analysis, finalize RecSys results

Phase 2: Tools A2C (Days 7-12)

- Day 7: Download ToolBench, precompute tool embeddings (16K tools × 6 models)
- Day 8: Implement A2C + transformer critic, test on 1 model
- Day 9-10: Train all 6 models (20 epochs each, ~24 hours total compute)
- Day 11: Evaluate on I3 test set, compute success rates
- Day 12: Generate comparison tables, finalize Tools results

Phase 3: Paper Writing (Days 13-14)

- Day 13 (Jan 27): Write experimental section, generate all plots/tables
- Day 14 (Jan 28): Final polish, submit by deadline

Phase 4: Math (OPTIONAL - skip if behind)

- Only attempt if Days 1-12 finish ahead of schedule

Compute Requirements

Embedding computation (one-time):

- Amazon 10K: 10K items × 6 models × 0.1 sec = 100 min
- ToolBench 16K: 16K tools × 6 models × 0.1 sec = 160 min
- Total: ~4.5 hours

Training:

- RecSys NeuralTS: 6 models × 5 seeds × 2 hours = 60 hours
- Tools A2C: 6 models × 20 epochs × 2 hours = 240 hours (parallelize!)
- Math A2C: Skip unless ahead

Total compute: ~300 GPU-hours

Hardware:

- Free Google Colab T4 (15 GB VRAM): Sufficient for everything except LLaMA-3
- M4 Mac (16 GB RAM): Can run BERT, RoBERTa, SimCSE, Jina
- Recommended: Rent A100 for 1-2 days (~$50) to parallelize training

Parallelization Strategy

Critical: Run experiments in parallel to meet deadline

```bash
# Run all 6 embedding models simultaneously (if enough GPUs)
python experiments/02_recsys_bandit.py --model bert --seed 42 &
python experiments/02_recsys_bandit.py --model roberta --seed 42 &
python experiments/02_recsys_bandit.py --model simcse --seed 42 &
python experiments/02_recsys_bandit.py --model jina --seed 42 &
python experiments/02_recsys_bandit.py --model llama3 --seed 42 &
python experiments/02_recsys_bandit.py --model llm2vec --seed 42 &

# Or use job scheduler
for model in bert roberta llama3 simcse jina llm2vec; do
  for seed in 42 43 44 45 46; do
    sbatch run_bandit.sh $model $seed
  done
done
```

Success Criteria

Minimum Viable Results (Must Achieve)

Embedding Analysis:

1. ✅ Eigenvalue spectra plot showing anisotropic vs contrastive separation

2. ✅ d_{eff} table: anisotropic ~40-60, contrastive ~200-220
3. ✅ RKHS norms: anisotropic >100, contrastive <10

RecSys Bandit: 4. ✅ Regret curves: contrastive methods 40-50% lower regret than anisotropic 5. ✅ Statistical significance: $p < 0.05$ (t-test across 5 seeds)

Tools A2C: 6. ✅ I3 success rate: SimCSE/Jina >65%, BERT/RoBERTa <55% 7. ✅ Outperform ToolLLM baseline (47%) with all methods

Stretch Goals

8. ✅ Math reasoning: SimCSE solve rate >65%, rollouts reduced by 25%
9. ✅ LLM2Vec validates that contrastive tuning fixes LLM anisotropy
10. ✅ Multimodal embeddings (CLIP/Jina-CLIP) on image+text RecSys

---

Key Bottlenecks & Solutions

Bottleneck 1: Embedding Computation Time

Problem: 60K+ embeddings (10K Amazon × 6 models + 16K tools × 6 models)

Solution:

- Aggressive disk caching (EmbeddingCache class)
- Compute once, reuse forever
- Parallelize across models (6 processes)
- Time: ~5 hours total (one-time cost)

Bottleneck 2: LLaMA-3 Access

Problem: Requires HuggingFace auth token, 8GB model

Solution:

- Request access: https://huggingface.co/meta-llama/Meta-Llama-3-8B
- Use 4-bit quantization: `load_in_4bit=True` (reduces to 4 GB)
- Fallback: Use Mistral-7B instead (open, no auth)

Bottleneck 3: RL Training Time

Problem: 6 models × 30 experiments = 180 training runs

Solution:

- Parallelize across GPUs/machines
- Use smaller networks (36M params is fast!)
- Reduce epochs if behind schedule (20 → 10)
- Cached embeddings make inference fast

Bottleneck 4: Tool Execution

Problem: Real API calls are slow

Solution:

- Use ToolLLM's cached results (they already ran all APIs)
- Download from their repo: `toolbench_cache.json`
- Simulation based on cached results

Bottleneck 5: Math Verification

Problem: Checking if answer is correct requires parsing

Solution:

- Use regex to extract final numerical answer
- SymPy for equation verification
- Skip this if time runs out (tools + recsys is enough)

Experiment Scripts

Master Script

```
python
```

```python
# experiments/run_all.py

import subprocess
import time

def run_experiment(script, model, seed):
    """
    Run a single experiment.
    """
    cmd = f"python {script} --model {model} --seed {seed}"
    print(f"Starting: {cmd}")

    start = time.time()
    result = subprocess.run(cmd, shell=True, capture_output=True)
    elapsed = time.time() - start

    print(f"Finished: {cmd} ({elapsed/60:.1f} min)")

    return result.returncode == 0

def main():
    models = ['bert', 'roberta', 'llama3', 'simcse', 'jina', 'llm2vec']
    seeds = [42, 43, 44, 45, 46]

    # Phase 0: Embedding analysis
    print("\n=== Phase 0: Embedding Analysis ===")
    run_experiment('experiments/01_embedding_analysis.py', model=None, seed=None)

    # Phase 1: RecSys bandit
    print("\n=== Phase 1: RecSys Neural Bandit ===")
    for model in models:
        for seed in seeds:
            success = run_experiment('experiments/02_recsys_bandit.py', model, seed)
            if not success:
                print(f"WARNING: {model} seed {seed} failed!")

    # Phase 2: Tools A2C
    print("\n=== Phase 2: Tools A2C ===")
    for model in models:
        success = run_experiment('experiments/03_tools_a2c.py', model, seed=42)
        if not success:
            print(f"WARNING: {model} failed!")
```

```
    # Phase 3: Math (optional)
    # Uncomment if ahead of schedule
    # print("\n=== Phase 3: Math A2C (optional) ===")
    # for model in models:
    #     run_experiment('experiments/04_math_a2c.py', model, seed=42)

    print("\n=== All experiments complete! ===")
    print("Results in: results/")
    print("Generate paper plots: python scripts/make_paper_figures.py")

if __name__ == '__main__':
    main()
```

Expected Paper Contributions

Main Claims

1. **Theoretical:** Contrastive embeddings ensure bounded RKHS norms by maximizing effective dimension d_{eff}, enabling sublinear regret
2. **Empirical:** Contrastive embeddings (SimCSE, Jina) achieve 40-50% lower regret than reconstruction-based (BERT, RoBERTa, LLaMA) across three domains
3. **Architectural:** Lightweight transformer critics (36M params) over frozen contrastive embeddings match or exceed heavy LLM-based critics (1.5B params)
4. **LLM Insight:** Base LLMs have anisotropic representations; LLM2Vec's success validates that contrastive fine-tuning is necessary

Figures for Paper

1. **Figure 1:** Eigenvalue spectra (validates Section 3.2)
2. **Figure 2:** Regret curves - RecSys (validates Section 5)
3. **Figure 3:** I3 success rates - Tools (validates Section 6)
4. **Figure 4:** Coverage metric ϱ(k,K) (shows exploration efficiency)
5. **Figure 5:** RKHS norms vs d_{eff} (validates Lemma 3.2)

Tables for Paper

1. **Table 1:** Embedding comparison (d_{eff}, RKHS norms)
2. **Table 2:** RecSys results (regret, CTR)

3. **Table 3:** Tools results (I3 success, diversity)
4. **Table 4:** Math results (solve rate, rollouts) - if time permits

---

Final Notes

Core Insight

Contrastive uniformity → high d_{eff} → bounded RKHS norm → efficient exploration → sublinear regret

Keep this thread through all experiments!

Flexibility

- If rStar-Math replication is hard, cite their results and focus on Tools
- If ToolBench download fails, use subset or cached data
- If LLaMA-3 requires too much compute, drop it (have 5 other embeddings)
- RecSys + Tools + Embedding Analysis is sufficient for strong paper

Documentation

- Comment code thoroughly
- Save all hyperparameters in config files
- Log all experiments (use wandb or tensorboard)
- Random seeds for reproducibility

Communication

If anything blocks you:

1. Check the bottleneck solutions above
2. Simplify the experiment (reduce data size, epochs, models)
3. Skip optional components (Math, multimodal, LLM2Vec)

Priority order: Embedding Analysis > RecSys > Tools > Math

Good luck! You have a strong story, solid theory, and a tractable implementation plan. The key is aggressive caching and parallelization to hit the deadline. 🚀