

Adaptive RAG: enhancing small model generation using DQN and reinforcement learning

How we built a self-improving RAG system that learns individual user preferences in real-time without fine-tuning the LLM

“What is AI?”

Exposed the core issue: **small local models can’t stay consistent**, and no RAG pipeline can fix a shaky base model.

```
>>> what is AI
Okay, let's break down what AI is. It's a really broad topic, so I'll cover the core concepts and different types.

**At its most basic, AI (Artificial Intelligence) refers to the ability of a computer to perform tasks that typically require human intelligence.** This includes things like:

* **Learning:** Acquiring information and rules for using it.
* **Reasoning:** Using logic and information to draw conclusions.
* **Problem-solving:** Identifying and solving issues.
* **Understanding language:** Interpreting and responding to text and speech.
* **Visual perception:** Recognizing objects and patterns in images.

**Here's a breakdown of different types of AI:**

**1. Narrow AI (or Weak AI):** This is the type of AI we see most prominently today. It's designed and trained for a specific task. It excels at that single task.

    * **Examples:**
        * **Spam filters:** Identify and filter unwanted emails.
        * **Recommendation systems (Netflix, Amazon):** Suggest products based on your viewing or purchase history.
        * **Voice assistants (Siri, Alexa, Google Assistant):** Understand and respond to voice commands.
        * **Image recognition software:** Identify objects in pictures.
        * **Chess-playing AI (Deep Blue):** Can beat humans at chess.

**2. General AI (or Strong AI):** This is a more advanced and theoretical type of AI that would possess human-level intelligence. It could understand, learn, adapt, and implement knowledge in any domain. *We don't currently have a general AI.*

    * **Characteristics:** Could perform any intellectual task that a human being can.
    * **Status:** Currently a research goal, not widely available.

**3. Sub-Categories of AI:**

    * **Machine Learning (ML):** This is a subset of AI where systems learn from data without being explicitly programmed. They improve their performance over time as they're exposed to more data.
        * **Deep Learning:** A subset of Machine Learning that uses artificial neural networks with multiple layers - inspired by the structure of the human brain. It's particularly good at complex pattern recognition and analysis.

    * **Natural Language Processing (NLP):** Focuses on enabling computers to understand, interpret, and generate human language.
```

The breaking point: when user preferences matter

The real problem emerged when we deployed to actual users. Different users wanted different things:

- **User A** (busy executive): Always wants one-sentence answers
- **User B** (researcher): Always wants comprehensive explanations with examples
- **User C** (engineer): Wants step-by-step procedures when asking "how-to" questions

Small models couldn't adapt. They gave the same unpredictable output to everyone. Fine-tuning wasn't an option—too expensive, too slow, and you'd need to retrain for every user preference shift.

Critical Design Decision: This system is designed for **per-user adaptation**. Each user gets their own neural network that learns their individual preferences. This is not a one-size-fits-all model—it's a personal AI assistant that adapts to YOU.

The core insight: don't train the LLM, train a "strategy brain"

What if we don't touch the language model at all?

Instead of trying to make Gemma "smarter," what if we build a small neural network that learns **how to ask Gemma questions**? A meta-controller that figures out:

- Which prompting strategy works best for this query?
- How much context should we retrieve?
- What response format does this user prefer?

This is fundamentally a **decision-making problem under uncertainty**—the exact domain where reinforcement learning excels.

The architecture: a personal strategy-learning brain for each user

Imagine the system as having two brains:

Brain 1 (Gemma 3:1b): The language generator—frozen, never trained

Brain 2 (User-Specific DQN): The strategy selector—learns YOUR preferences

Brain 2 learns to solve a **contextual multi-armed bandit problem**:

```
# The 5 "arms" - each represents a different response format
STRATEGIES = ["concise", "detailed", "structured",
              "example_driven", "analytical"]

# Each strategy gets a different top_k for retrieval
STRATEGY_K_VALUES = {
    "concise": 3,      # Less context needed
    "detailed": 5,     # More context
    "structured": 4,
```

```
"example_driven": 4,  
"analytical": 6      # Maximum context  
}
```

Key Point: This neural network is **per-user**. When you interact with the system, it creates and trains a DQN specifically for you. Your colleague, using the same system, gets their own DQN learning their preferences. This enables true personalisation.

Component 1: understanding the query space

The first challenge: queries aren't one-dimensional. "What is X?" is different from "Explain X in detail," which is different from "How does X work?"

We classify every query across **three orthogonal dimensions** using GPT-4-mini (cheap at \$0.15 per million tokens):

```
# Multi-dimensional classification  
INTENT_TYPES = [  
    "definition",      # What is X?  
    "explanation",      # How/Why does X work?  
    "procedure",       # How to do X?  
    "comparison",      # X vs Y  
    "analysis",        # Analyse/Evaluate X  
    "factual"         # When/Where is X?  
]  
  
DEPTH_LEVELS = ["surface", "moderate", "comprehensive"]  
SCOPE_TYPES = ["specific", "broad"]
```

This creates **108 possible query clusters** ($6 \times 3 \times 2$). Each cluster represents a unique query type like "explanation_comprehensive_broad" or "definition_surface_specific".

The classification uses a carefully engineered prompt:

```
CLASSIFICATION_PROMPT = """"You are an expert at analysing user queries.  
Classify this query across THREE dimensions with precision.  
  
QUERY: "{query}"
```

```
DIMENSION 1 - INTENT (choose ONE):
- definition - User wants to know WHAT something IS
- explanation - User wants to understand HOW or WHY
- procedure - User wants STEP-BY-STEP instructions
- comparison - User wants DIFFERENCES or SIMILARITIES
- analysis - User wants EVALUATION or CRITICAL THINKING
- factual - User wants SPECIFIC FACTS or DATA

DIMENSION 2 - DEPTH: surface, moderate, comprehensive
DIMENSION 3 - SCOPE: specific, broad

Respond with ONLY valid JSON:
{"intent": "...", "depth": "...", "scope": "..."}"""
```

Why this matters: Users asking "What is X?" consistently prefer concise answers. Users asking "How does X work?" prefer detailed explanations with examples. By clustering queries, we can learn these patterns and generalise across similar query types.

Component 2: the personal strategy-learning neural network

At the heart of the system is a **Dueling Deep Q-Network (DQN)**—a reinforcement learning architecture that learns to predict how good each strategy will be for a given query **for this specific user**.

The dueling architecture: separating value from advantage

Standard Q-networks output one value per action: "How good is taking action A in state S?"

Dueling DQNs split this into two separate streams:

```
class DuelingDQN(nn.Module):
    def __init__(self, input_dim: int, output_dim: int, hidden_dims:
List[int]):
        super(DuelingDQN, self).__init__()

        # Shared feature extractor
        self.feature_extractor = nn.Sequential(...)
```

```

# Value stream: "How good is this query overall?"
self.value_stream = nn.Sequential(
    nn.Linear(prev_dim, 64),
    nn.ReLU(),
    nn.Linear(64, 1) # Single value
)

# Advantage stream: "How much better is each strategy?"
self.advantage_stream = nn.Sequential(
    nn.Linear(prev_dim, 64),
    nn.ReLU(),
    nn.Linear(64, output_dim) # 5 advantages
)

def forward(self, x):
    features = self.feature_extractor(x)
    value = self.value_stream(features)
    advantages = self.advantage_stream(features)

    # Dueling formula
    q_values = value + (advantages - advantages.mean(dim=1,
keepdim=True))
    return q_values

```

Why dueling? Imagine a query cluster where most strategies work well (high base value) but one strategy is particularly excellent. The value stream learns "this is generally a good query cluster" while the advantage stream learns "strategy X is especially good here." This decomposition helps the network learn faster and generalise better.

The 424-dimensional feature vector

The network processes a rich feature vector for each query:

```

NEURAL_INPUT_DIM = 424 # Total feature dimensions

# Feature composition:
# 1. Semantic embedding: 384-dim (Sentence Transformer)
# 2. Classification features: 11-dim (6+3+2 one-hot)
# 3. Intent hash embedding: 8-dim

```

```
# 4. Query statistics: 21-dim (length, complexity, etc.)
```

This high-dimensional representation allows the network to learn complex patterns about which strategies work for which query characteristics—for **each individual user**.

Network architecture configuration

```
# Neural Network Configuration
NEURAL_INPUT_DIM = 424
NEURAL_HIDDEN_DIMS = [256, 128, 64] # Deep network
NEURAL_OUTPUT_DIM = 5 # One Q-value per strategy
NEURAL_DROPOUT = 0.3 # High dropout for generalisation
NEURAL_LEARNING_RATE = 0.0001 # Conservative learning
```

Component 3: learning from your feedback

Here's where reinforcement learning happens. Every interaction follows this cycle:

Step 1: User submits query

Step 2: The system classifies the query into a cluster

Step 3: Your personal neural network predicts Q-values for all 5 strategies

Step 4: System selects strategy (exploration vs exploitation)

Step 5: RAG retrieves context with strategy-specific top_k

Step 6: Gemma generates a response with a strategy-specific prompt

Step 7: You give thumbs up 👍 or thumbs down 👎

Step 8: Your neural network learns from YOUR feedback

The learning mechanism: prioritised experience replay

Traditional reinforcement learning stores experiences uniformly. We use **prioritised experience replay**, which focuses on "surprising" experiences—cases where the network's prediction was very wrong.

```
class PrioritizedReplayBuffer:
    def __init__(self, capacity: int, alpha: float = 0.6):
        self.capacity = capacity
        self.alpha = alpha # Prioritisation strength
        self.buffer = []
        self.priorities = []
```

```
def add(self, experience: Tuple, priority: float = 1.0):
    max_priority = max(self.priorities) if self.priorities else 1.0
    # New experiences get max priority
    self.priorities.append(max_priority)
```

Each experience gets a priority based on **TD-error** (temporal difference error):

```
def add_experience(self, features, action, reward, next_features):
    current_q = self.predict(features)[action]
    # TD-error: how "wrong" was our prediction?
    priority = abs(reward - current_q) + 1e-5

    self.replay_buffer.add(experience, priority)
```

During training, high-priority experiences are sampled more frequently:

```
def sample(self, batch_size: int, beta: float = 0.4):
    priorities = np.array(self.priorities)
    probabilities = priorities ** self.alpha
    probabilities /= probabilities.sum()

    # Sample based on priorities
    indices = np.random.choice(len(self.buffer), size=batch_size,
                               p=probabilities, replace=False)
```

Real-world impact: Without prioritisation, your network might need 200 interactions to learn your preferences. With prioritisation, it converges in ~50-75 interactions.

Component 4: exploration vs exploitation - adapting to you

This is the classic RL dilemma: Should the system try new strategies (explore) or use what it knows works for you (exploit)?

We use **adaptive epsilon-greedy** exploration that changes based on training progress:

```
def get_adaptive_epsilon_params(num_queries: int) -> dict:
    """Adaptive epsilon based on training length"""
```

```

if num_queries <= 50:
    return {
        'decay': 0.97,
        'start': 0.90,
        'min': 0.15,
        'description': 'Aggressive exploration for early learning'
    }
elif num_queries <= 100:
    return {
        'decay': 0.98,
        'start': 0.85,
        'min': 0.12
    }
else:
    return {
        'decay': 0.99,
        'start': 0.75,
        'min': 0.10
    }

```

The epsilon decays gradually after each interaction:

```

# Epsilon decay after each feedback
self.epsilon = max(
    self.epsilon * EPSILON_DECAY, # 0.985
    EPSILON_MIN                   # 0.10
)

```

Concept drift detection: when your preferences change

User preferences change over time. Maybe you initially wanted concise answers, but now want detailed explanations. We detect this using **statistical z-scores**:

```

DRIFT_WINDOW_SIZE = 15      # Recent performance window
DRIFT_REFERENCE_SIZE = 30   # Historical baseline
DRIFT_THRESHOLD = 2.5      # Statistical significance

def detect_concept_drift(self, recent_rewards, reference_rewards):
    recent_mean = np.mean(recent_rewards)
    reference_mean = np.mean(reference_rewards)

```



```

reference_std = np.std(reference_rewards)

z_score = (recent_mean - reference_mean) / (reference_std + 1e-8)

if abs(z_score) > DRIFT_THRESHOLD:
    # Drift detected! Boost exploration
    self.epsilon = min(self.epsilon * 1.3, 0.95)
    return True

```

Why this matters: Without drift detection, your system would get stuck using outdated strategies. With drift detection, it adapts when your preferences shift—typically within 10-15 queries.

Component 5: reward shaping with strategy families

Instead of binary rewards (+1 for good, -1 for bad), we use **graduated rewards** based on strategy families:

```

INTENT_STRATEGY_RULES = {
    "definition": {
        "primary": ["concise"],
        "acceptable": ["structured"],
        "family": ["concise", "structured"]
    },
    "explanation": {
        "primary": ["detailed", "example_driven"],
        "acceptable": ["analytical"],
        "family": ["detailed", "example-driven", "analytical"]
    }
}

REWARD_SHAPING = {
    'correct_primary': 1.0,          # Perfect match
    'correct_acceptable': 0.5,      # Good alternative
    'wrong_family': 0.0,            # Neutral
    'wrong_strategy': -0.3          # Soft penalty
}

```

This creates a **smoother learning signal**. If you ask "What is machine learning?" and we provide a structured definition instead of a concise one, we get +0.5 instead of -0.3. Both formats work reasonably well for definitions—one is just more optimal.

Impact on learning: Reward shaping reduces variance in learning and helps your network converge faster.

Component 6: the hybrid memory system - fast and slow learning

We maintain **two types of memory** working in concert:

Memory type 1: neural network weights (slow learning)

- Generalizable patterns across all your queries
- Slow to update but robust
- Handles novel query types through generalisation

Memory type 2: cluster-specific strategy performance (fast learning)

- Fast, explicit tracking per query cluster
- Quick lookup: "What worked before for this exact cluster?"
- Provides a warm-start signal for the neural network

```
def record_strategy_performance(self, cluster_name, strategy, reward):
    "Track performance per cluster"
    perf = self.clusters[cluster_name]['strategy_performance']

    if strategy not in perf:
        perf[strategy] = {'total': 0, 'wins': 0}

    perf[strategy]['total'] += 1
    if reward > 0:
        perf[strategy]['wins'] += 1
```

When selecting a strategy, we combine both memories:

```
def select_strategy(self, query, features, cluster_name, best_strategy):
    # Get Q-values from YOUR neural network
    q_values = self.neural_bandit.predict(features)
```

```

# Boost strategy from cluster memory (warm-start)
if best_strategy and self.neural_bandit.needs_warm_start():
    best_idx = STRATEGIES.index(best_strategy)
    q_values[best_idx] += WARM_START_EXPLORATION_BONUS

# Epsilon-greedy selection
if random.random() < self.epsilon:
    strategy_idx = random.randint(0, len(STRATEGIES)-1)
else:
    strategy_idx = np.argmax(q_values)

```

Component 7: strategy-specific prompting and retrieval

Each strategy doesn't just change the prompt—it changes the **entire retrieval and generation pipeline**:

```

# Strategy-specific prompts
STRATEGY_PROMPTS = {
    "concise": """You MUST answer in EXACTLY ONE sentence.
No more than 20 words.

Context: {context}
Question: {question}

ONE SENTENCE ONLY: """,

    "detailed": """ Provide a comprehensive, thorough explanation.
Use 4-6 sentences minimum. Include background, explanation, and
implications.

Context: {context}
Question: {question}

Detailed comprehensive answer: """,

    "structured": """ "Answer using EXACTLY this format:

1. MAIN POINT: [one sentence]
2. KEY DETAILS: [bullet points]

```

```
3. SUMMARY: [one sentence]
```

```
Context: {context}
```

```
Question: {question}
```

```
Structured answer:""
```

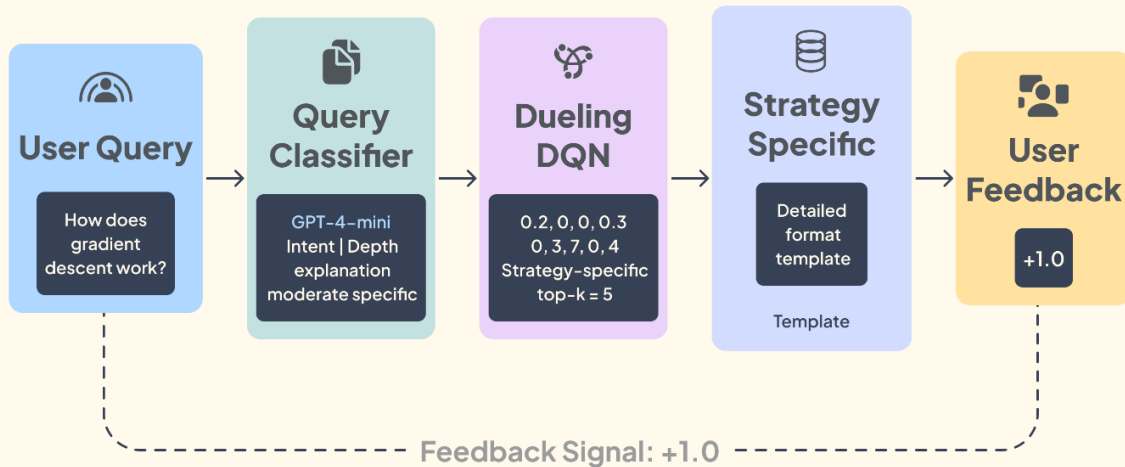
```
}
```

The prompts are **deliberately extreme** to force Gemma into specific formats. Small models need strong constraints—subtle hints don't work.

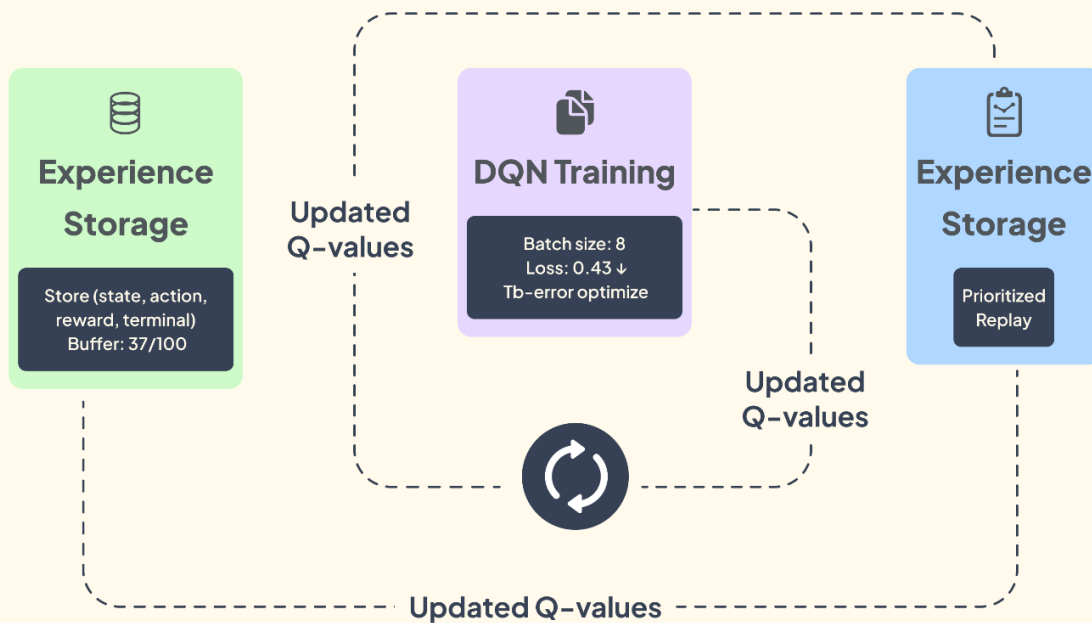
The complete system flow

Neural-RL Adaptive RAG

Inference Pipeline



Learning Loop



Step 1: Multi-dimensional classification

The system first needs to understand what type of query this is. It sends the query to GPT-4-mini, which analyses it across three dimensions:

- **Intent Classification:** Determines this is an "explanation" query (user wants to understand HOW something works, not just WHAT it is)
- **Depth Assessment:** Classifies as "moderate" depth (not a surface-level quick answer, but not a comprehensive deep-dive either)
- **Scope Determination:** Identifies as "specific" scope (focused on one concept, not broad coverage)

Result: Query cluster = "explanation_moderate_specific"

This clustering is crucial because your neural network will learn that queries in this cluster typically benefit from detailed explanations with examples.

Step 2: Feature extraction (building the 424-dimensional vector)

The system now builds a rich representation of your query by combining multiple feature types:

Semantic understanding (384 dimensions):

- The Sentence Transformer model encodes the query's meaning into a 384-dimensional vector
- Captures semantic similarity: "How does gradient descent work?" is similar to "Explain the gradient descent algorithm"

Classification features (11 dimensions):

- One-hot encoding of intent (6 dimensions): [0, 1, 0, 0, 0, 0] for "explanation"
- One-hot encoding of depth (3 dimensions): [0, 1, 0] for "moderate"
- One-hot encoding of scope (2 dimensions): [1, 0] for "specific"

Additional context (29 dimensions):

- Intent hash embedding (8 dimensions): Learned representation of intent patterns
- Query statistics (21 dimensions): Length, word count, complexity metrics, cluster metadata

Total: 424-dimensional feature vector that completely characterises your query

This rich representation allows your neural network to understand not just what you asked, but HOW you asked it and what context surrounds it.

Step 3: Strategy selection (your personal neural network decides)

Your personal Dueling DQN now predicts how good each of the 5 strategies would be for this query:

Neural network processing:

- Takes the 424-dimensional feature vector as input
- Passes through deep layers: $424 \rightarrow 256 \rightarrow 128 \rightarrow 64$ dimensions
- Value stream outputs: "This query situation has a base value of 0.6" (generally good query)
- Advantage stream outputs: How much better/worse each strategy is than average

Q-value predictions (one per strategy):

- Concise: 0.2 (low confidence - explanations rarely work well concise)
- Detailed: 0.8 (highest - system learned YOU prefer detailed for explanations)
- Structured: 0.3 (medium-low - structured format not ideal for this)
- Example-driven: 0.7 (high - also good for explanations)
- Analytical: 0.4 (medium - not the best fit)

Exploration vs exploitation decision:

- Current epsilon (exploration rate): 0.15 (85% exploitation, 15% exploration)
- System generates random number: 0.87 (above 0.15)
- Decision: EXPLOIT - use learned knowledge
- **Selected strategy:** "detailed" (highest Q-value of 0.8)

If the random number had been below 0.15, the system would have explored by trying a random strategy to continue learning.

Step 4: Strategy-specific retrieval

Now that the strategy is selected, the retrieval pipeline adapts:

Retrieval configuration:

- Strategy "detailed" is configured to retrieve **5 documents** (more context needed for comprehensive explanations)
- If it had selected "concise", it would only retrieve 3 documents (less context for brief answers)
- If "analytical" was chosen, it would retrieve 6 documents (maximum context for deep analysis)

Vector search process:

- Your query embedding is compared against all document chunks in ChromaDB
- Cosine similarity calculated for each chunk
- Top 5 most relevant chunks retrieved based on semantic similarity

Retrieved context:

- Document 1: "Gradient descent is an optimisation algorithm..."
- Document 2: "The learning rate parameter controls..."
- Document 3: "Common variations include SGD, Adam..."
- Document 4: "Convergence depends on the loss landscape..."
- Document 5: "Practical implementation considerations..."

Step 5: Strategy-specific prompt construction

The system now constructs a highly specific prompt designed to force Gemma into the "detailed" format:

Prompt engineering:

- **Not used:** Generic prompt like "Answer this question"
- **Used:** Extremely prescriptive prompt: "Provide a comprehensive, thorough explanation with full details. Use 4-6 sentences minimum. Include background, explanation, and implications."

Why this works: Small models like Gemma need mechanical constraints. The prompt literally instructs: "Use 4-6 sentences minimum" which gives Gemma clear boundaries it can follow, unlike vague instructions like "be detailed."

Complete prompt structure:

- Instruction layer: What format to use
- Context layer: The 5 retrieved document chunks
- Question layer: Your original query
- Format reminder: Reinforces the expected output format





Step 6: Generation (Gemma produces a response)

Gemma 3:1b now generates a response following the detailed prompt template:

Generated response (5-6 sentences): "Gradient descent is an iterative optimisation algorithm used to minimise a loss function by adjusting model parameters. The algorithm calculates the gradient (derivative) of the loss function with respect to each parameter, indicating the direction of steepest increase. Parameters are then updated in the opposite direction of the gradient, scaled by a learning rate hyperparameter. This process repeats until convergence, when the loss reaches a minimum. The learning rate is crucial—too large causes overshooting, too small causes slow convergence. Common variants include Stochastic Gradient Descent (SGD), which

uses mini-batches for efficiency, and adaptive methods like Adam that adjust learning rates per parameter."

Format Adherence:


-  6 sentences (meets 4-6 requirement)
-  Includes background (what it is)
-  Includes explanation (how it works)
-  Includes implications (learning rate importance, variants)

The mechanical prompt constraints worked—Gemma produced exactly the format specified.

Step 7: User provides feedback

You read the response and evaluate it:

Your assessment:

- The explanation was comprehensive and helpful
- Appropriate level of detail for understanding the concept
- Good balance of theory and practical considerations
- **Decision:** Click  (thumbs up)

System receives:

- Feedback signal: +1 (positive)
- This is translated to reward: +1.0 (primary strategy reward)

Why this reward: The system checks: Query was "explanation" intent, strategy used was "detailed", which is a PRIMARY strategy for explanations according to reward shaping rules. Therefore, a full positive reward of +1.0.

If the strategy had been "analytical" (also works for explanations but not optimal), reward would be +0.5 (acceptable alternative).

Step 8: Experience storage (learning from this interaction)

The system now stores this valuable learning experience:

Experience tuple created:

- **State (features):** The 424-dimensional vector representing your query
- **Action (strategy index):** 1 (index of "detailed" in the strategy list)
- **Reward:** +1.0 (positive feedback)
- **Next state:** None (terminal state - interaction complete)
- **Terminal flag:** True (no future states to consider)

Priority calculation:

- The system predicted Q-value was 0.8 for this action
- Actual reward received was 1.0
- TD-error: $|1.0 - 0.8| = 0.2$
- Priority score: $0.2 + \text{small constant} = \sim 0.2$

This experience gets priority 0.2, meaning it will be sampled more frequently than experiences where the network was already accurate (small TD-error).

Prioritised replay buffer update:

- Experience added to buffer (current size: 37 experiences)
- Sorted by priority for future sampling
- Buffer capacity: 100 experiences (will keep the most recent/important ones)

Step 9: Neural network training (your model updates)

Since your buffer now has 37 experiences (well above the minimum of 5), training is triggered:

Batch sampling (prioritised):

- System samples 8 experiences from the buffer (batch size = 8)
- Higher-priority experiences (bigger mistakes) sampled more frequently
- This batch might include: 3 high-priority (surprising) + 5 medium-priority experiences

Double DQN training process:**Forward pass (current Q-values):**

- Your policy network processes the 8 sampled states (8×424 features)
- Outputs Q-values for all strategies for each state
- Extracts Q-values for the actions that were actually taken

Target calculation (What Q-values should be):

- For terminal states (like this one): Target = reward only (no future value)
- For non-terminal states: Target = reward + $0.97 \times \max(\text{future Q-values})$
- Uses target network (slowly updated copy) for stability
- Prevents Q-value overestimation through the Double DQN architecture

Loss calculation:

- Compares predicted Q-values to target Q-values
- Uses Smooth L1 Loss (Huber loss) for robustness
- Weighs each sample by importance sampling weight (from prioritised replay)

- Higher-priority experiences have more influence on the gradient

Backpropagation:

- Loss: 0.43 (decreasing over time as network learns)
- Gradients calculated through all network layers
- Gradients clipped to maximum norm of 1.0 (prevents exploding gradients)
- Optimiser (Adam) updates network weights
- Learning rate: 0.0001 (conservative for stable learning)

Target network soft update:

- Target network slowly tracks policy network
- Update formula: $\text{target} = 0.005 \times \text{policy} + 0.995 \times \text{target}$ (TAU = 0.005)
- This provides stable targets for Q-learning

Priority updates:

- New TD-errors calculated for the 8 sampled experiences
- Buffer priorities updated to reflect new prediction accuracy
- Experiences where the network is now accurate get lower priority

Training metrics updated:

- Training step: 23 → 24
- Average loss: 0.43 (down from 0.51 in previous training)
- Average Q-value: 0.67 (stabilising around true expected rewards)
- Buffer size: 37 experiences

Step 10: Exploration rate decay (shifting toward exploitation)

After successful training, the system updates its exploration strategy:

Epsilon decay:

- Current epsilon before: 0.150
- Decay multiplier: 0.985
- New epsilon: $0.150 \times 0.985 = 0.148$
- Minimum epsilon: 0.10 (won't go below this)

What this means:

- Originally (query 1): 95% exploration ($\epsilon = 0.95$) - trying everything randomly
- Early learning (query 20): 75% exploration ($\epsilon = 0.75$) - mostly exploring
- Current (query 85): 15% exploration ($\epsilon = 0.15$) - mostly exploiting learned knowledge

- Eventually (query 150+): 10% exploration ($\epsilon = 0.10$) - highly confident, occasional exploration

Practical impact:

- The next similar query has a 14.8% chance of random exploration
- 85.2% chance of using learned best strategy ("detailed" for explanations)
- System is increasingly confident in its learned preferences for YOU

Step 11: Cluster memory update (fast learning mechanism)

In parallel with neural network training, the cluster-specific memory is updated:

Cluster performance tracking:

- Cluster: "explanation_moderate_specific"
- Strategy used: "detailed"
- Outcome: WIN (positive feedback)

Updated statistics:

- "detailed" strategy in this cluster: 12 wins out of 15 attempts (80% win rate)
- "example_driven" strategy: 8 wins out of 10 attempts (80% win rate)
- "analytical" strategy: 5 wins out of 8 attempts (62.5% win rate)
- Other strategies: Lower performance

Best strategy determination:

- Both "detailed" and "example-driven" show 80% win rates
- "detailed" has more attempts (15 vs 10) - more reliable
- Cluster memory marks "detailed" as the best strategy for this cluster

Future impact:

- Next time you ask a similar explanation question (moderate depth, specific scope)
- Neural network gets a small boost (+0.2) for "detailed" strategy from cluster memory
- Provides a warm-start signal, especially valuable in early learning phases

The cycle completes: system is now smarter

What changed after this single interaction:

1. **Neural network weights updated:**

- Network is slightly more confident that "detailed" works for explanation queries
 - Q-value for detailed strategy in similar contexts: 0.8 → 0.82
2. **Cluster memory enriched:**
 - "explanation_moderate_specific" cluster has one more data point
 - Win rate for "detailed" strategy updated: helps future warm-start
 3. **Exploration reduced:**
 - Epsilon decayed: 0.150 → 0.148
 - The system is more likely to exploit learned knowledge next time
 4. **Experience buffer grown:**
 - 37 experiences are now available for training
 - Richer training signal for future batches
 5. **System confidence increased:**
 - Loss decreasing over time: learning is working
 - Q-values stabilising: predictions becoming more accurate

Next time you ask a similar question: The system will likely select "detailed" strategy again (unless exploring), retrieve appropriate context, generate a well-formatted response, and continue learning from your feedback—getting better with every interaction.

This is how your personal neural network learns your preferences through continuous feedback and adaptation.

The training process: how your personal model learns

Phase 1: Cold start (Your First 1-15 Queries)

The challenge:

- Your replay buffer is empty → no training possible yet
- No cluster history → no hints
- Pure random exploration ($\epsilon=0.95$)
- Inconsistent experience (but expected!)

The solution - warm-start strategy:

```
WARM_START_ENABLED = True
WARM_START_EXPLORATION_BONUS = 0.2
WARM_START_MINIMUM_PER_STRATEGY = 2

def needs_warm_start(self) -> bool:
```

```
return any(
    count < WARM_START_MINIMUM_PER_STRATEGY
    for count in self.strategy_attempts.values()
)
```

Your experience:

- Queries 1-5: Random, experimental (system tells you it's learning)
- Queries 6-15: Starting to see patterns
- Buffer at 5+ experiences: Your neural network starts training!

Phase 2: Fast learning (Queries 16-50)

What's happening:

- Your neural network is trained every 5-8 queries
- Prioritised replay, focusing on the biggest mistakes
- Exploration still high ($\epsilon=0.90 \rightarrow 0.80$)
- Your cluster memory is building up

Observable changes:

- Win rate climbs from 50% (random) to 65-70%
- Some query types converge quickly (definitions \rightarrow concise)
- Other query types are still being explored
- Epsilon decaying: $0.90 \rightarrow 0.80 \rightarrow 0.70$

Network internals:

```
# Training metrics you can monitor
{
    'training_steps': 15,
    'buffer_size': 35,
    'recent_loss': 0.42,  # Decreasing
    'recent_q_value': 0.65,  # Stabilizing
    'epsilon': 0.75  # Decaying
}
```

Phase 3: Convergence (Queries 51-100)

What's happening:

- Your network is increasingly confident

- Exploration dropping ($\epsilon=0.70 \rightarrow 0.40$)
- Most clusters have clear best strategies
- Training focusing on edge cases

Observable changes:

- Win rate 75-85%
- Consistent performance within query clusters
- You notice the system "gets" your preferences
- Occasional exploration keeps it flexible

Phase 4: Maintenance (queries 100+)

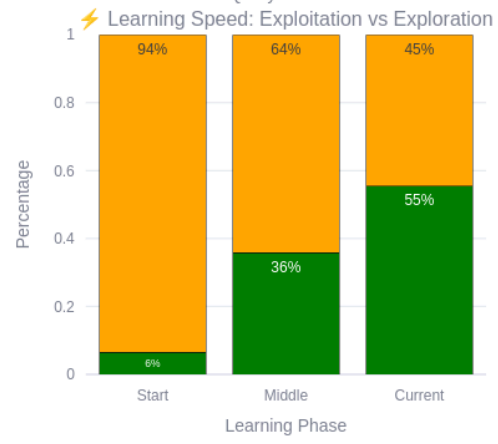
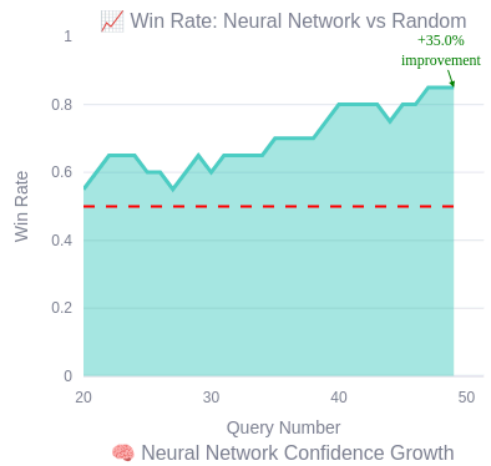
What's happening:

- Low exploration ($\epsilon=0.30 \rightarrow 0.10$)
- Concept drift monitoring is active
- Your network maintains learned policy
- Adapts if your preferences change

Observable changes:

- Win rate stabilizes at 85-90%
- When drift detected: temporary exploration boost
- System adapts to preference changes within 10-15 queries

🎓 **LEARNING PROOF: Why Neural Network + RL is Better**





Comprehensive Analytics



Performance Overview

Success Rate	Total Queries	Learning Health	Exploration Rate
90.0%	50	84/100	19.6%
↑ +40.0% vs random			



Comprehensive Learning Proof



Learning Speed Analysis

Learning Speed	Convergence Point	Current Success Rate
Very Fast	10	90.0%

The results: after training and complete implementation

After training the model on my preference of how a definition answer should be, and asking the same question and as you can see, it responded with a concise one-line answer, unlike before.



Neural-RL Adaptive RAG System

Training **Testing** Analytics Chat Advanced Metrics



Testing Mode

Enter Test Query

What is AI

Test Query



Response

AI encompasses artificial intelligence systems designed to mimic human cognitive functions.



Current System State

Total Queries

50

Success Rate

90.0%

Learning Health

84/100



Strategy Analysis

Selected Strategy

Concise

Intent Detected

definition

Depth Level

surface

> Strategy Performance

The real-world challenges we faced

Challenge 1: Cold start performance

The problem: Your first 10-15 queries are essentially random. You might get frustrated.

What we tried:

1. Pre-training on synthetic data → Didn't generalise to real preferences
2. Starting with uniform distribution → Still inconsistent
3. **Clear user expectations:** Show "Learning Your Preferences" indicator
4. **Warm-start bonuses:** Boost strategies from cluster memory
5. **Optional transfer learning:** Load pre-trained base weights (not preferences)

Final Solution:

```
# Warm-start configuration
```

```

WARM_START_ENABLED = True
WARM_START_MINIMUM_PER_STRATEGY = 2

# User sees: "Learning Phase: 5/15 queries"
# System forces diversity in first 10 queries

```

Challenge 2: Feedback sparsity

The problem: Users don't always give feedback. Some queries get no thumbs up/down.

What we tried:

1. ❌ Assume positive if no feedback → Biased learning
2. ❌ Request feedback on every query → Annoying
3. ✅ **Smart feedback requests:** Only ask when uncertain

Final solution:

```

# Only request feedback on high-uncertainty queries
if max(q_values) - min(q_values) < 0.3: # Network unsure
    # Show prominent feedback request
    show_feedback_request(importance="high")

```

Challenge 3: Q-value explosion

The problem: During early training, Q-values would explode ($\pm 100+$), causing unstable learning.

Root cause: Improper terminal state handling + bootstrapping errors

Final solution:

```

def train_batch(self):
    # CRITICAL: Proper terminal state handling
    target = torch.where(
        terminals,
        Rewards,                                # Terminal: no future value
        rewards + GAMMA * next_q                 # Non-terminal: bootstrap
    )

    # Clip Q-values to prevent explosion
    target = torch.clamp(target, -10.0, 10.0)

```

```
# Gradient clipping
torch.nn.utils.clip_grad_norm_(self.policy_net.parameters(), 1.0)
```

Challenge 4: Concept drift sensitivity

The problem: Your preferences change over time. The system needs to detect and adapt.

Solution:

```
def detect_concept_drift(self):
    recent_rewards = self.reward_history[-15:]
    reference_rewards = self.reward_history[-45:-15]

    z_score = (np.mean(recent) - np.mean(reference)) / np.std(reference)

    if abs(z_score) > 2.5: # Statistical significance
        # Your preferences changed!
        self.epsilon = min(self.epsilon * 1.3, 0.95) # Re-explore
    return True
```

Performance metrics: what we measure

Learning efficiency (per user)

- **Win Rate vs queries:** Your improvement over time
- **Cold start duration:** Queries until 70% win rate (target: <15)
- **Convergence speed:** Queries until 85% win rate (target: <75)

Model health (your neural network)

```
def get_training_metrics(self):
    return {
        'training_steps': self.training_step,
        'buffer_size': len(self.replay_buffer),
        'recent_loss': np.mean(self.loss_history),
        'recent_q_value': np.mean(self.q_value_history),
        'recent_reward': np.mean(self.reward_history),
```

```
'epsilon': self.epsilon,  
'training_ready': self.should_train()  
}
```

User experience

- **Response consistency:** Same cluster → same strategy ($\pm 5\%$)
- **Format adherence:** Does the response match the selected strategy? ($> 90\%$)
- **Adaptation speed:** Queries to adapt to your preference change (< 20)

System performance

- **Classification time:** $< 50\text{ms}$ (GPT-4-mini + caching)
- **Strategy selection time:** $< 5\text{ms}$ (neural network forward pass)
- **Training time:** $< 100\text{ms}$ per batch (CPU)
- **End-to-end latency:** $< 500\text{ms}$ (including Gemma generation)

Why this approach works for individual users

Personalisation at scale

- **Each user gets their own DQN:** Your network, your preferences
- **No interference between users:** Your colleague's preferences don't affect yours
- **True personalisation:** System learns YOUR specific needs

Data efficiency

Traditional fine-tuning: 10,000+ examples

RLHF: 1,000+ preference pairs

Our approach: 50-100 YOUR interactions

Because we're learning a simpler function:

"Given query features, which of 5 strategies does THIS USER prefer?"

Continuous adaptation

```
# After every feedback from YOU:  
if len(self.replay_buffer) >= MIN_BUFFER_FOR_TRAINING:  
    training_result = self.neural_bandit.train_batch()  
    # Your network updates immediately
```

Explainability

Every decision is traceable:

```
selection_info = {
    'method': 'neural_network',
    'q_values': q_values,
    'epsilon': self.epsilon,
    'cluster_best': best_strategy,
    'confidence': 'high' if max_q - min_q > 0.5 else 'low'
}
# You can see WHY the system chose each strategy
```

System architecture summary

```
class LocalAdaptiveRAG:
    """Personal RAG system that learns YOUR preferences"""

    def __init__(self, user_id: str):
        self.user_id = user_id # YOUR unique ID





        # Shared components (efficient)
        self.vector_store = LocalVectorStore(user_id)
        self.classifier = LocalQueryClassifier(user_id)

        # YOUR personal learning agent
        self.rl_agent = EnhancedRLAgent(user_id)

        # YOUR personal neural network
        self.neural_bandit = NeuralBandit(user_id)
```




When this approach makes sense

Ideal use cases:

-  **Individual users with consistent preferences** (executives, researchers, engineers)
-  Cost-sensitive production systems
-  Need for continuous adaptation
-  Latency-critical applications (local inference)

-  Privacy-sensitive data (everything stays local)

Not ideal for:

-  One-off queries (not enough feedback to learn)
-  Extremely diverse query types (need more feedback data)
-  Users who never give feedback (the system can't learn)

Technical implementation: the complete stack

```
# Requirements
"""
ollama                # Local LLM
sentence-transformers  # Embeddings
chromadb              # Vector store
torch                 # Neural network
numpy, pandas         # Data processing
plotly, streamlit     # Visualisation
psutil                # Monitoring
"""

# Key Configuration
NEURAL_INPUT_DIM = 424
NEURAL_HIDDEN_DIMS = [256, 128, 64]
NEURAL_LEARNING_RATE = 0.0001
BATCH_SIZE = 8
EPSILON_START = 0.95
EPSILON_DECAY = 0.985
GAMMA = 0.97
```

Future directions

1. Cross-user base models

- Pre-train base network on aggregated patterns (not preferences)
- Fine-tune per-user with 10-20 queries instead of 50-100
- Transfer learning for faster cold start

2. Hierarchical strategies

```
# Current: 5 flat strategies
```

```
# Future: Hierarchical
STRATEGIES = {
    "concise": ["concise_technical", "concise_simple"],
    "detailed": ["detailed_theoretical", "detailed_practical"]
}
```

3. Active learning

```
# Intelligently request feedback on uncertain queries
if network_uncertainty > threshold:
    request_feedback(priority="high",
                     reason="Unsure which format you prefer")
```

4. Multi-objective optimisation

```
# Optimize for: satisfaction + latency + cost
# Pareto-optimal strategy selection
q_values = alpha*satisfaction + beta*speed + gamma*cost
```

Conclusion: personal AI that learns from you

This work introduces a fundamentally different path for adaptive AI: rather than training the language model itself, the system trains a personal controller that learns how you prefer it to respond. It is built on several core insights. Personalisation becomes truly powerful when every user has their own neural network. A compact, one-million-parameter DQN is sufficient to guide and shape the behaviour of a much larger three-billion-parameter language model. Your own feedback becomes the most valuable training signal, far more meaningful than any synthetic dataset. The system keeps learning continuously, adapting to your evolving preferences in real time and adjusting within just a handful of interactions.

Another strength lies in transparency—its decision-making process remains fully explainable. With only fifty to a hundred interactions, it can understand your style, refine itself as your needs shift within ten to fifteen queries, operate entirely on CPU, and scale at essentially no cost. For individuals who want a RAG setup that genuinely aligns with their personal expectations, this architecture offers a practical, efficient, and highly effective way forward.

Complete implementation available at: https://github.com/pratikbhande/small_adaptive_rag

Presented at OASYS 2025 AI Conference

This research demonstrates that sophisticated reinforcement learning techniques can be

practically applied to create truly personal AI systems. The future isn't one-size-fits-all AI—it's AI that learns from you, adapts to you, and works the way you want it to.