Sub-100ms Latency Optimization in Production ML Systems: Lessons from Building Customer Intelligence Platforms

Ramya Boorugula

Corresponding Author: Ramya Boorugula (email: boorugular@gmail.com)

Abstract

Over the past three years, I've been working on optimizing production ML systems that need to make decisions fast enough to actually matter to customers. This paper documents what I learned building and optimizing customer intelligence platforms that process 50TB+ of data daily while keeping response times under 100ms.

The systems I worked on include ensemble churn prediction (XGBoost + LSTM + transformers hitting 87% accuracy), neural contextual bandits that drove 31% spend lift, and ML-powered finite state machines that improved account manager productivity by 2.4x. The main challenge was keeping all this complexity running fast enough to enable real-time interventions.

My key findings: (1) genetic algorithms can optimize feature pipelines better than manual engineering, cutting computation time by 70%, (2) smart caching strategies matter more than hardware upgrades, (3) async orchestration is critical but tricky to get right, and (4) you need to optimize for business metrics, not just technical ones.

The bottom line: complex ML systems can run fast enough for real-time use cases, but it requires rethinking how you approach optimization. This paper shares the practical techniques that actually worked in production.

1. Introduction

When I started working on customer intelligence systems three years ago, I thought the hard part would be building accurate models. Turns out, the real challenge is making those models fast enough to matter in the real world.

Here's the problem: customer intelligence isn't like batch processing or offline analytics. When someone's about to churn, you have maybe 30 seconds to detect it and trigger an intervention. When a user is browsing your site, you have 100ms to show them relevant recommendations

before they lose interest. When an account manager needs to prioritize their day, they can't wait 5 minutes for the system to compute optimal allocations.

I learned this the hard way. My first churn prediction system was technically impressive—87% accuracy with a beautiful ensemble of XGBoost, LSTM, and transformers. But it took 250ms to make a prediction. By the time we could intervene, customers had already mentally checked out. We were building a fire department that took 10 minutes to respond to emergency calls.

1.1 What This Paper Covers

This paper documents the engineering journey of optimizing three interconnected ML systems:

Churn Prediction: Ensemble model that had to predict customer churn fast enough to enable intervention workflows. Started at 250ms, needed to get under 50ms while maintaining 87% accuracy.

Recommendation Engine: Neural contextual bandits with Deep Q-Networks for long-term optimization. Had to adapt to user behavior in real-time while driving measurable business impact (we proved 31% spend lift).

Lifecycle Automation: ML-powered finite state machines managing customer journeys, plus linear programming for optimizing human resource allocation. Needed to update states and allocations fast enough to guide daily decision-making.

The engineering challenge: these systems are interconnected. Churn predictions feed into lifecycle states, which trigger account manager actions, which influence recommendations. Everything has to work together within tight latency budgets.

1.2 Why This Matters

Most ML optimization papers focus on making individual models faster. But production systems aren't just individual models—they're complex pipelines where one model's output becomes another model's input, and everything needs to happen fast enough to influence real customer moments.

This paper documents the practical techniques that actually worked in production. Some were obvious (caching, parallelization), others were surprising (genetic algorithms for feature engineering), and many were harder to implement than expected (async orchestration, resource isolation).

2. Starting Point: Why Everything Was Too Slow

When I inherited this system, it was technically sound but practically useless due to latency. Let me break down where the time was going.

2.1 The Latency Breakdown That Shocked Me

I expected model inference to be the bottleneck. I was wrong. After instrumenting everything, here's what I found:

Feature Engineering (40ms): This was the biggest surprise. We were pulling features from everywhere—user activity logs, transaction history, support tickets, behavioral analytics. Some features required aggregations over months of data. Others needed real-time calculations. Processing 200+ features was eating 40ms before we even got to the models.

Churn Ensemble (85ms):

- XGBoost: 20ms (feature interactions and non-linear patterns)
- LSTM: 30ms (sequential behavior modeling)
- Transformers: 35ms (contextual relationships)

Recommendation System (110ms):

- Context encoding: 35ms (user history + item features + real-time signals)
- Reward prediction: 25ms (neural network for click/purchase probabilities)
- Exploration strategy: 15ms (Thompson sampling with uncertainty estimates)
- DQN agent: 35ms (long-term optimization)

Lifecycle FSM (25ms per advertiser): This doesn't sound bad until you realize we had 50,000 advertisers. Updating all state machines would take 20+ minutes.

Network overhead: 5-15ms per service call, and we were making dozens of them per request.

Total: 260ms+ for a single customer interaction. Way too slow for real-time intelligence.

2.2 Why Standard Optimization Didn't Work

My first attempts at optimization were pretty naive:

"Just add more hardware": Throwing GPUs at the problem helped with individual model inference but didn't address the fundamental architectural issues.

"Cache everything": Generic Redis caching helped a bit, but customer intelligence has unique freshness requirements that broke standard caching assumptions.

"Parallelize all the things": My first async implementation actually made things slower due to resource contention. Multiple models fighting for the same GPU memory isn't faster—it's chaos.

The real insight: customer intelligence systems have different constraints than typical ML applications. You can't optimize them the same way.

3. Optimization Strategy: What Actually Worked

After a lot of trial and error, I developed a systematic approach that worked across all three systems.

3.1 Genetic Algorithm Feature Optimization

This was probably my biggest surprise. I was spending weeks manually tuning feature combinations, and honestly, I was probably missing optimal combinations that no human would think to try.

The idea: treat feature pipelines like DNA. Each "chromosome" is a different combination of features and transformations. Let evolution find better combinations than manual engineering.

3.1.1 How I Built It

The genetic algorithm treated feature pipelines as chromosomes where each gene represented a feature selection or transformation decision:

```
class FeaturePipelineChromosome:
    def __init__(self):
        self.features = []  # Selected features
        self.transformations = {}  # Feature transformations
        self.aggregation_windows = {}  # Time windows for aggregations

    def fitness(self):
        accuracy = test_pipeline_accuracy(self)
        latency_ms = measure_pipeline_latency(self)

        # Heavy penalty for exceeding latency budget
        if latency_ms > 20:  # 20ms feature budget
            penalty = (latency_ms - 20) * 0.3
        else:
            penalty = 0

        return accuracy - penalty
```

Population initialization used domain knowledge to seed promising combinations. Mutation operators included feature addition/removal and transformation parameter adjustments. Crossover combined successful feature subsets from parent pipelines.

3.1.2 Results That Surprised Me

After running this overnight for three nights (I'm impatient), the algorithm evolved feature sets that:

- Reduced feature count from 200+ to 35 optimal features
- Cut computation time from 40ms to 12ms (70% reduction)
- Maintained 87% accuracy
- Found feature combinations I never would have tried manually

The algorithm discovered that rolling windows on behavioral features I thought were noise actually contained strong signals. It also learned to avoid expensive aggregations that provided minimal predictive value.

Key lesson: Sometimes the best optimization engineer is an algorithm you let loose on the problem.

3.2 Fixing the Ensemble Parallelization Nightmare

My first attempt at parallelizing the churn ensemble was embarrassingly naive:

```
# This actually made things slower
async def broken_ensemble(features):
    xgb_task = asyncio.create_task(xgboost_predict(features))
    lstm_task = asyncio.create_task(lstm_predict(features))
    transformer_task = asyncio.create_task(transformer_predict(features))

    results = await asyncio.gather(xgb_task, lstm_task, transformer_task)
    return combine_predictions(results)
```

Why was this slow? All three models were fighting for the same resources. It was like putting three drivers in the same car and wondering why you're not going faster.

3.2.1 Resource Isolation Strategy

The fix required resource isolation:

```
async def fast_ensemble(features):
    # XGBoost gets its own CPU cores with NUMA awareness
    xgb_task = run_on_cpu_pool(xgboost_predict, features)

    # LSTM gets primary GPU stream with pre-allocated memory
    lstm_task = run_on_gpu_stream_1(lstm_predict, features)

    # Transformer gets secondary GPU stream
    transformer_task = run_on_gpu_stream_2(transformer_predict, features)
```

```
    results = await asyncio.gather(xgb_task, lstm_task, transformer_task)

    # Learned weights from way too many experiments
    return 0.4 * results[0] + 0.35 * results[1] + 0.25 * results[2]
```

This required careful memory management and stream synchronization, but it worked. Results:

- Total ensemble time: 85ms → 32ms (62% reduction)
- Resource utilization: 40% → 85% average GPU utilization
- Much more consistent inference times

3.3 Model Quantization: What Works and What Doesn't

Everyone talks about quantization like it's magic. "Just use INT8 and everything gets faster!" Reality is more complicated.

3.3.1 XGBoost Quantization (Easy Win)

Tree models love quantization. The tree structure does the heavy lifting, so leaf value precision doesn't matter much. Converting from FP32 to INT8:

- 40% speedup
- Zero accuracy loss
- Trivial to implement

3.3.2 LSTM Quantization (Tricky)

LSTM cells are sensitive to quantization in recurrent connections. My solution:

- Keep LSTM cell states in FP16 (preserve sequential modeling)
- Quantize fully connected layers to INT8 (they're less sensitive)
- Result: 35% speedup with <0.5% accuracy hit

The key insight: not all parts of a model are equally sensitive to precision loss.

3.3.3 Transformer Mixed Precision (Complex but Worth It)

Attention mechanisms need careful precision management:

- Matrix operations: FP16 (computational efficiency)
- Attention weights: FP32 (numerical stability)
- Added key-value caching for users we'd seen recently

Result: 45ms → 18ms inference time (60% reduction). The caching was actually more important than the precision changes.

3.4 Making Neural Contextual Bandits Actually Fast

The recommendation system was probably the most complex optimization challenge. Neural contextual bandits need to balance exploration and exploitation while adapting to user behavior in real-time.

3.4.1 Context Encoding Hierarchy

My original context encoder treated all user data equally—6-month-old browsing history got the same processing intensity as real-time session data. This was expensive and unnecessary.

The fix was hierarchical encoding:

```
def fast_context_encoding(user_id, item_id, session_data):
    # Hot path for 80% of users (recently seen)
    if user_id in hot_user_cache:
        user_embedding = hot_user_cache[user_id]  # 2ms cache hit
    else:
        # Cold path for new/returning users
        user_embedding = full_user_encoder(user_id)  # 18ms
        hot_user_cache[user_id] = user_embedding

    # Items change less frequently
    item_embedding = item_cache.get(item_id)  # 1ms

    # Only session data needs fresh computation
    session_embedding = session_encoder(session_data)  # 6ms

    # Fast fusion network
    context = fusion_net([user_embedding, item_embedding, session_embedding])  # 4ms

    return context
```

Results: 35ms → 12ms for hot users (80% of traffic), 29ms for cold users.

3.4.2 DQN Student-Teacher Distillation

The full DQN was taking 40ms to evaluate action values. Too slow for real-time recommendations.

I used neural network distillation to create a fast approximation:

- Teacher: 7-layer DQN with full state representation
- Student: 3-layer network optimized for speed
- Training: Student learns to approximate teacher Q-values

Results: 12ms inference time, 94% performance retention. Good enough for real-time decisions.

3.4.3 Action Space Pruning

Even with a fast Q-network, evaluating thousands of possible recommendations was still slow. Solution: two-stage filtering.

```
def recommend_with_pruning(user_context):
    # Stage 1: Fast similarity-based pruning (8ms)
    candidates = similarity_index.query(user_context, top_k=100)

    # Stage 2: Quick quality filter (4ms)
    quality_scores = fast_quality_model.predict(candidates)
    top_candidates = candidates[quality_scores > threshold][:50]

    # Stage 3: Expensive DQN evaluation only on filtered set (12ms)
    q_values = fast_q_network.predict(user_context, top_candidates)

    # Final decision with exploration (2ms)
    if random.random() < exploration_rate:
        return random.choice(top_candidates)
    else:
        return top_candidates[np.argmax(q_values)]
```

Total: 26ms. Finally fast enough for real-time personalization.

3.5 Finite State Machine Optimization

The lifecycle automation system needed to update states for 50,000 advertisers in real-time. At 25ms per advertiser, this was clearly impossible.

3.5.1 Transition Prediction Caching

The insight: most state transitions are predictable. Advertisers don't randomly jump between states—they follow patterns.

I built a caching system that exploited this predictability:

```
class SmartTransitionPredictor:
    def __init__(self):
        self.hot_cache = {}  # Recent predictions
        self.pattern_cache = {}  # Common patterns

    def predict_transition(self, advertiser_id, features):
        # Hash features for cache lookup
```

```
    feature_hash = self.bucket_features(features)
    cache_key = (advertiser_id, feature_hash)

    # Try hot cache first
    if cache_key in self.hot_cache:
        return self.hot_cache[cache_key]  # 0.3ms

    # Try pattern matching for similar advertisers
    pattern = self.find_similar_pattern(features)
    if pattern in self.pattern_cache:
        result = self.adapt_pattern(pattern, features)  # 2ms
        self.hot_cache[cache_key] = result
        return result

    # Last resort: full ML computation
    result = self.full_model.predict(advertiser_id, features)  # 18ms
    self.update_caches(cache_key, pattern, result)
    return result
```

Results: 25ms → 3ms average (88% reduction). This made real-time FSM updates feasible.

3.5.2 Linear Programming with Warm Starts

The account manager allocation optimization required solving LP problems with thousands of variables. Standard solvers were taking 50ms+.

Key optimizations:

- **Warm starts**: Reuse previous solutions as starting points
- **Problem decomposition**: Break large problems into smaller chunks
- **Smart constraints**: Add business logic to guide solver toward practical solutions

```
def optimize_with_warm_start(current_allocation, changes):
    # Start from previous solution
    problem = create_lp_problem()
    problem.set_warm_start(current_allocation)

    # Only re-optimize changed constraints
    for change in changes:
        problem.update_constraint(change)

    # Solve incrementally
    solution = problem.solve(time_limit=15)  # ms
    return solution
```

Results: 50ms → 15ms while maintaining solution quality.

4. System Integration: Making Everything Work Together

The individual optimizations were great, but the real challenge was orchestrating everything together. Customer interactions trigger multiple ML systems that have complex dependencies.

4.1 Dependency-Aware Async Pipeline

Some computations can run in parallel, others have dependencies. For example:

- Churn prediction + base recommendations (parallel)
- Churn score → lifecycle update (dependency)
- Lifecycle state → account manager notification (dependency)

My orchestration framework handles this automatically:

```
async def process_customer_interaction(user_id, interaction_data):
    # Start parallel computations
    churn_task = predict_churn_ensemble(user_id, interaction_data)
    base_reco_task = generate_base_recommendations(user_id, interaction_data)

    # Wait for churn prediction (needed for lifecycle)
    churn_score = await churn_task
    lifecycle_task = update_lifecycle_state(user_id, churn_score, interaction_data)

    # Continue parallel execution
    base_recs = await base_reco_task
    personalized_recs = personalize_with_churn_context(base_recs, churn_score)

    # Final coordination
    new_state = await lifecycle_task
    final_recs = await personalized_recs

    # Conditional execution
    if new_state.requires_intervention:
        await notify_account_manager(user_id, new_state, final_recs)

    return {
        'churn_score': churn_score,
        'recommendations': final_recs,
        'lifecycle_state': new_state
```

```
}
```

This reduced total pipeline time from 100ms+ to 62ms while handling all dependencies correctly.

4.2 Hierarchical Caching That Actually Works

Generic caching advice doesn't work for customer intelligence systems. You need domain-specific strategies that understand business logic.

4.2.1 Cache Design

My three-level cache hierarchy:

L1 (Application Memory):

- Hot user embeddings, 15-min TTL
- 78% hit rate, 12ms average savings
- Invalidated on user behavior changes

L2 (Redis):

- Item embeddings, 1-hour TTL
- 85% hit rate, 8ms average savings
- Shared across application instances

L3 (Precomputed):

- State transition probabilities, 5-min TTL
- 90% hit rate when applicable
- Updated async in background

4.2.2 Business-Aware Invalidation

The tricky part: knowing when to invalidate caches. Customer intelligence data has complex freshness requirements.

Rules I learned the hard way:

- User behavior changes → immediate L1 invalidation
- Model deployments → coordinated cache clearing
- A/B test boundaries → cache isolation
- Time-sensitive features → aggressive TTLs

Total impact: 35% average latency reduction with proper cache management.

5. Results: What We Actually Achieved

After 18 months of optimization work, here's what changed:

5.1 Latency Improvements

System Component	Before	After	Improvement
Feature Engineering	40ms	12ms	70%
Churn Ensemble	85ms	32ms	62%
Recommendation System	110ms	26ms	76%
Lifecycle FSM	25ms	3ms	88%
Total Pipeline	**260ms**	**62ms**	**76%**

5.2 Business Impact That Actually Mattered

The latency improvements translated directly to business results:

5.2.1 Churn Prevention

- **Before**: 45% intervention success rate (customers we could save)
- **After**: 68% intervention success rate
- **Why**: Faster predictions meant catching customers earlier in the churn process

5.2.2 Recommendation Performance

- **Measured spend lift**: 31% increase in customer spending
- **Statistical confidence**: $p < 0.001$ using propensity score matching
- **Key insight**: Recommendations that arrive too late don't drive conversions

5.2.3 Account Manager Productivity

- **Baseline**: Manual customer prioritization
- **Optimized**: 2.4x improvement in customer outcomes
- **Mechanism**: Real-time state updates + optimized allocation

5.3 System Reliability Under Load

Production deployment showed the optimizations were robust:

- **Peak traffic**: 10,000+ concurrent customer interactions
- **Failover performance**: <5% degradation during node failures
- **Scaling**: Linear latency with customer growth

6. What I Learned: Engineering Insights That Actually Matter

6.1 Optimization Priority Hierarchy

Three years of optimization taught me that not all optimizations are equal:

1. **Algorithmic optimization** (highest ROI): Feature engineering, model architecture
2. **System optimization** (medium ROI): Parallelization, caching, resource management
3. **Infrastructure optimization** (lowest ROI): Hardware upgrades, deployment tuning

This surprised me. I expected infrastructure to provide the biggest gains, but algorithmic changes often gave order-of-magnitude improvements while hardware changes were incremental.

6.2 Business Metrics Drive Technical Decisions

Critical insight: technical metrics can be misleading if they don't correlate with business outcomes.

Example: I could improve churn model accuracy from 87% to 89% by adding more features, but this increased latency from 45ms to 80ms. The slower, more accurate model performed worse in practice because it couldn't enable fast enough interventions.

Lesson: optimize for business impact, not technical perfection.

6.3 Complexity vs. Performance Trade-offs

Customer intelligence systems need sophisticated models to achieve business impact. The key is strategic complexity allocation:

High complexity where it matters:

- Ensemble models for churn (each model type catches different patterns)
- Neural bandits for recommendations (drive measurable business lift)
- ML-powered FSMs for lifecycle automation (enable intelligent automation)

Aggressive simplification elsewhere:

- Fast heuristics for candidate generation

- Cached computations for stable features
- Approximate algorithms for non-critical paths

6.4 Common Engineering Mistakes I Made

6.4.1 Async Programming Pitfalls

Implementing async orchestration revealed several gotchas:

Resource contention: Multiple models competing for shared GPU memory

- *Solution*: Resource pools and isolation

Error propagation: Single component failures breaking entire pipelines

- *Solution*: Timeout strategies and graceful degradation

Debugging nightmares: Async stack traces are confusing

- *Solution*: Correlation IDs and extensive logging

6.4.2 Cache Invalidation Complexity

ML systems have unique caching requirements that break standard patterns:

Model updates: When you deploy a new model, related caches become stale **Data distribution shifts**: Cached predictions can become incorrect over time **A/B test isolation**: Different experimental groups need separate caches

The solution: domain-specific invalidation logic that understands ML semantics.

6.4.3 Monitoring Production ML Systems

Standard application monitoring isn't enough for ML systems. You need:

End-to-end business metrics: Customer outcomes, not just technical performance **Model-specific monitoring**: Accuracy, bias, prediction distribution tracking **Real-time business correlation**: Alert when technical degradation impacts business metrics

7. Related Work and Industry Context

7.1 How This Differs from Academic Research

Most ML optimization research focuses on individual model performance or theoretical improvements. This work addresses practical challenges of integrated production systems:

- **Multi-model orchestration**: Academic work rarely addresses dependency management across multiple models
- **Business constraint optimization**: Research typically optimizes accuracy; production systems need to balance accuracy with latency and business impact
- **Real-world resource constraints**: Academic environments don't face the same scaling and reliability requirements

7.2 Industry Practices

My approaches align with and extend industry practices from major tech companies:

Google's recommendation systems use similar caching and parallelization but focus more on content recommendation than customer intelligence.

Amazon's personalization platforms employ comparable feature optimization but rely more heavily on infrastructure scaling than algorithmic optimization.

Netflix's recommendation engine uses similar A/B testing frameworks but operates in different business contexts with different latency requirements.

The main difference: customer intelligence systems have tighter latency constraints because they need to enable real-time interventions, not just content serving.

8. Future Challenges and Open Problems

8.1 Emerging Optimization Opportunities

Several areas look promising for further optimization:

Edge deployment: Moving inference closer to customers to eliminate network latency entirely. This is particularly relevant for mobile applications and global deployments.

Hardware acceleration: Custom ASICs designed specifically for customer intelligence workloads could provide significant performance improvements.

Federated learning: Distributed training approaches could improve model freshness while maintaining customer privacy, but introduce new latency challenges.

8.2 Scaling Challenges Ahead

As these systems grow, new engineering problems emerge:

LLM integration: The industry is moving toward integrating large language models into customer intelligence systems. These models are powerful but computationally expensive. Finding ways to get LLM benefits within latency constraints is an active research area.

Real-time feature updates: Current systems batch feature updates. True real-time intelligence would require streaming feature computation at scale.

Multi-tenancy optimization: Efficiently sharing resources across different customer segments while maintaining isolation and performance guarantees.

8.3 Evaluation Methodology Limitations

Current approaches to measuring optimization success have gaps:

Long-term impact: Most metrics focus on immediate outcomes. Better evaluation of long-term business impact would improve optimization decisions.

Causal attribution: It's often difficult to attribute business outcomes to specific technical changes. Better causal inference methods would help.

Cross-domain validation: Optimization strategies that work for one business context may not generalize. More systematic evaluation across domains would be valuable.

9. Conclusion

Building production customer intelligence systems taught me that latency optimization isn't just about making individual models faster—it's about orchestrating complex systems to work together within tight time constraints while maintaining the sophistication needed for business impact.

The key insights from this work:

1. **Algorithmic optimization often beats infrastructure optimization**: Feature engineering and model architecture changes provided bigger gains than hardware upgrades.

2. **Business metrics should drive technical decisions**: Optimizing for business outcomes rather than pure technical metrics leads to better real-world performance.

3. **System-level thinking is essential**: Production ML systems are pipelines, not individual models. Optimization strategies must account for dependencies and interactions.

4. **Domain-specific solutions matter**: Customer intelligence systems have unique constraints that require specialized approaches to caching, monitoring, and optimization.

The results speak for themselves: 76% latency reduction while maintaining model performance that drives measurable business impact (87% churn prediction accuracy, 31% recommendation spend lift, 2.4x account manager productivity improvement).

More importantly, this work demonstrates that you don't have to choose between model sophistication and real-time performance. With the right engineering approaches, complex ML systems can operate within stringent latency constraints while delivering significant business value.

The techniques documented here should be applicable to other customer intelligence systems, though the specific optimizations will need to be adapted to different business contexts and technical constraints.

For engineers working on similar problems: start with comprehensive profiling, optimize the whole pipeline (not just individual components), and always validate that technical improvements translate to business impact. The goal isn't to build the fastest system—it's to build the system that has the biggest positive impact on customers and business outcomes.

Acknowledgments

Thanks to the engineering and data science teams who worked on these systems with me. Special recognition to the account managers and customer success teams who helped me understand the real-world requirements that drove these optimizations. Their feedback was crucial for understanding which technical improvements actually mattered in practice.

Also thanks to the customers who unknowingly participated in our A/B tests. Their behavior patterns taught us what works and what doesn't in customer intelligence systems.

References

[1] Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785-794.

[2] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735-1780.

[3] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

[4] Chapelle, O., & Li, L. (2011). An empirical evaluation of thompson sampling. *Advances in neural information processing systems*, 24.

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540), 529-533.

[6] Li, L., Chu, W., Langford, J., & Schapire, R. E. (2010). A contextual-bandit approach to personalized news article recommendation. *Proceedings of the 19th international conference on World wide web*, 661-670.

[7] Hinton, G., Vinyals, O., & Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

[8] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., ... & Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2704-2713.

[9] Rosenbaum, P. R., & Rubin, D. B. (1983). The central role of the propensity score in observational studies for causal effects. *Biometrika*, 70(1), 41-55.

[10] Malkov, Y. A., & Yashunin, D. A. (2018). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4), 824-837.