Connect 4: Neural Networks in Action

Project Overview

Connect 4 is a deceptively simple two-player strategy game that presents a rich combinatorial search space and complex board-state dynamics. In this project, we explore how deep learning models can be trained to predict optimal moves given a board configuration and compete effectively against human players.

Our work focuses on designing and training two distinct neural network architectures - a **Convolutional Neural Network (CNN)** and a **Transformer-based model**, to evaluate board states and recommend optimal moves. We analyze the training pipeline, architectural decisions, performance trade-offs, and classification challenges encountered during development.

Beyond model training, we extend this work into a fully functional system by integrating the trained models into a live gameplay environment using **Anvil (frontend), AWS (cloud infrastructure), and Docker (containerized backend services)**. This transforms the project from a standalone machine learning experiment into a production-style deployment of neural network agents.

The result is an end-to-end Connect 4 system that bridges neural network training, architectural experimentation, deployment engineering, and real-time inference demonstrating both machine learning rigor and applied systems integration.

Executive Summary

This project investigates the development and deployment of deep neural networks to play Connect 4, focusing on both model performance and real-world integration.

Key highlights include:

- **Data Generation Pipeline:** Creation of labeled board states representing optimal moves to train supervised learning models.
- **CNN Architecture Design:** Leveraging spatial locality to capture board patterns such as threats, blocks, and winning alignments.
- **Transformer Architecture Design:** Applying attention mechanisms to model long-range dependencies and board-wide strategic interactions.
- **Model Evaluation:** Comparative analysis of strengths, weaknesses, and misclassification scenarios across architectures.
- **Deployment Engineering:** Integration of trained models into a live Connect 4 web application using Anvil, AWS, and Docker containers.

- **System Architecture Design:** Separation of CNN and Transformer inference services into independent containerized backends for modularity and scalability.

Through this project, we demonstrate how neural networks can learn structured strategic reasoning from board representations and how machine learning systems can transition from research prototypes to deployable interactive applications.

Table of Contents

Neural Network Design and Training for Connect 4

In this section, we detail the end-to-end process of designing, training, and evaluating neural networks capable of playing Connect 4. Our objective was not only to achieve strong predictive performance, but to understand how different architectural choices influence strategic behavior on the board. We explore the data generation process, the design and training of a Convolutional Neural Network (CNN), and the development of a Transformer-based model, analyzing what worked well, what presented challenges, and how each model ultimately learned to interpret and act upon complex game states.

Data Generation

MCTS Configuration

We used Monte Carlo Tree Search with a skill level of 1500 simulations per move, balancing computational efficiency with move quality. At this skill level, MCTS provides strong strategic play while maintaining reasonable generation speed (~2-5 seconds per move).

Game Generation Process

Data was collected through MCTS self-play where both players used the same MCTS algorithm:

- **Random Opening Moves:** To increase dataset diversity, each game began with 0-3 random moves.. These opening moves were not recorded in the training data, serving only to create varied starting positions.
- **MCTS Self-Play:** After the random opening, MCTS selected moves for both players. Only these MCTS-recommended moves were recorded as training examples.
- **Game Termination:** Games ended when a player achieved four-in-a-row. No positions from completed games were added to the dataset after a win was detected.

Board Representation

We used 6×7×2 one-hot encoding rather than the simpler ±1/0 format:

- Channel 0: Positions occupied by +1 player (binary: 1 or 0)
- Channel 1: Positions occupied by -1 player (binary: 1 or 0)
- Empty positions: Both channels = 0

This representation provides clearer signals for neural networks to learn spatial patterns.

Perspective Normalization

A critical preprocessing step was normalizing all board positions to the +1 player's perspective. When the -1 player made a move, we:

- Flipped the board representation (swapped channels 0 and 1)
- Recorded the move from the +1 perspective

This ensures the neural network only needs to learn strategy for one player rather than maintaining separate representations for both perspectives.

Initial Generation

We generated data in batches to work within Google Colab's session limits. We played ~7,000 games generating 222,539 board-move pairs before deduplication.

Data Augmentation

To maximize dataset size and enforce symmetry, we applied horizontal mirroring:

- Each board position was flipped left-to-right
- Corresponding moves were mirrored (column 0↔6, 1↔5, 2↔4, 3→3)
- This doubled our effective dataset size

The mirroring is valid because Connect 4 is symmetric, so a strong move in column 2 is equally strong in column 4 when the board is mirrored.

Duplicate Handling

MCTS's inherent randomness means the same board position can appear multiple times with different recommended moves. We discovered that:

- 28.2% of positions were duplicates (same board, possibly different moves)
- 49.1% of these duplicates had conflicting move recommendations

Rather than just keeping the first occurrence, we used majority voting. This means that for each unique board position appearing multiple times, we count the frequency of each recommended move and keep the board with the most frequently recommended move. This approach is more reliable than random selection.

Final Dataset Statistics

After deduplication with majority voting, we gathered 215,309 unique positions with no redundancy and each position has the consensus "best" move.

Move Distribution:

- Perfectly symmetric after mirroring (Col 0≈Col 6, Col 1≈Col 5, etc.)
- Center column (3): 10.5% - slightly less preferred than edges
- Edge columns (0,6): 17.1% each - MCTS favors edge control
- Balanced distribution (10-17% range) indicates diverse positions

Board Encoding Quality:

- Format: (215309, 6, 7, 2) numpy array
- Data type: float32 for memory efficiency
- Values: Only 0s and 1s (one-hot encoding)
- Moves: Integer range 0-6

Game Characteristics:

- Average ~30 recorded positions per game
- Games ranged from early victories with ~10 moves to near-full boards with ~40 moves

This dataset captures Connect 4 positions from opening moves through endgame tactics, providing the supervised learning foundation for both CNN and Transformer architectures.

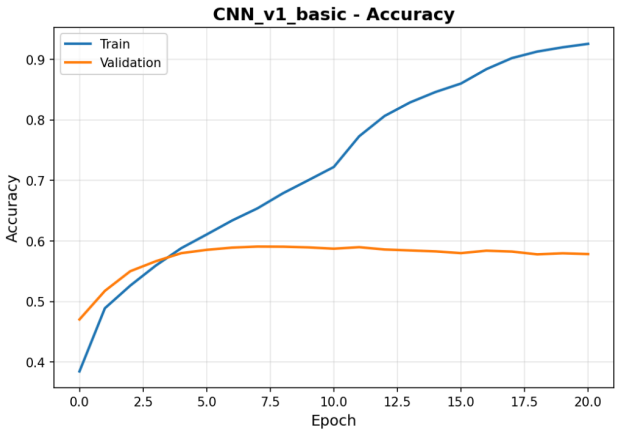
Convolutional Neural Network (CNN)

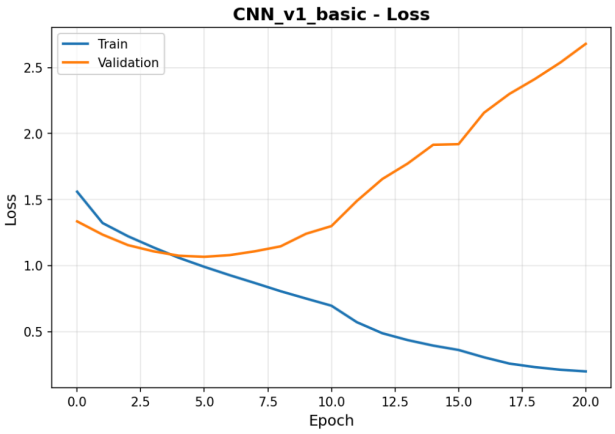
In building the convolutional neural network, we tested three architectures to find what works best:

CNN_v1_basic (Baseline)

- 4 convolutional layers (64→128→128→256 filters)
- 2 dense layers (256→128)
- 1.7M parameters

This architecture resulted in 58.4% test accuracy, but faced issues with overfitting, with 92% train vs 59% val accuracy.

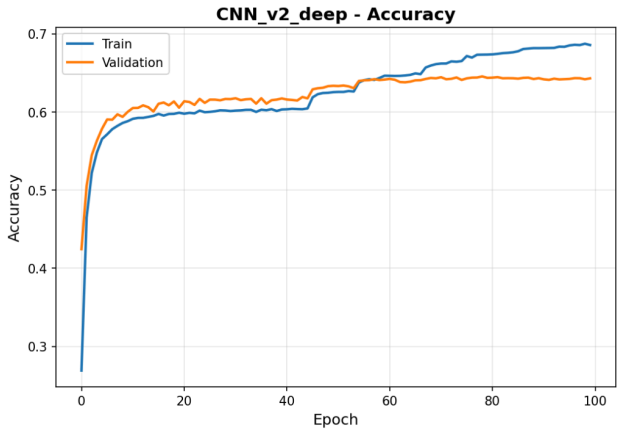


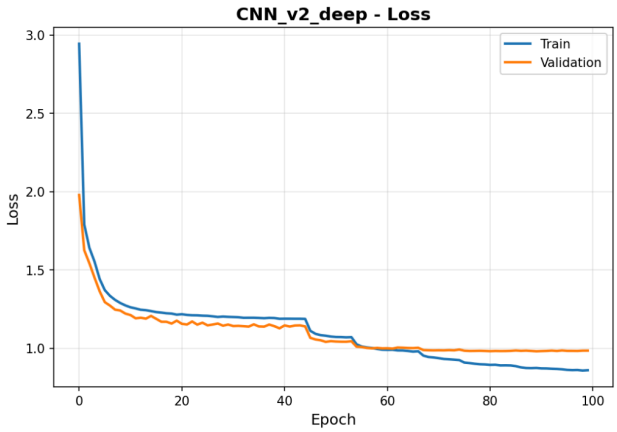


CNN_v2_deep (Final Model)

- 6 convolutional layers (128→128→256→256→512→512 filters)
- 2 dense layers (512→256)
- L2 regularization on all conv layers
- Dropout (20-50%) after dense layers
- 13.4M parameters

This architecture resulted in 64.1% test accuracy. It had much better generalization than the first model, with only a 4% gap between train and validation accuracy.

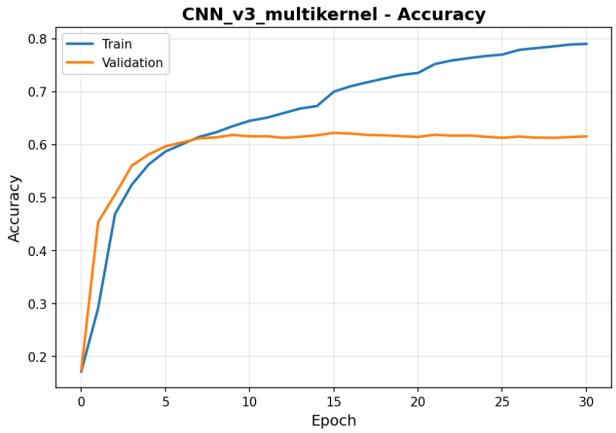


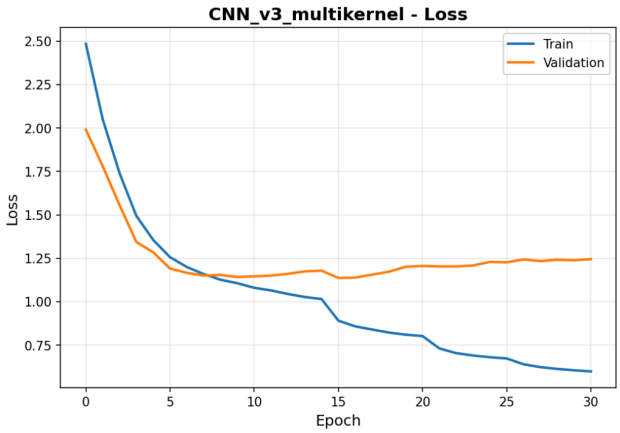


CNN_v3_multikernel (Experimental)

- Multiple kernel sizes (3×3, 5×5, 7×7) to capture patterns at different scales
- Concatenates outputs from different kernel sizes
- Similar depth to CNN_v2

This architecture resulted in 62.1% test accuracy, slightly lower than the second model. It faced issues with overfitting, though not as much as the first model. This showed us that multiple scales don't help much on small 6×7 boards; 3×3 filters are sufficient.





Given these results, we decided to go with CNN_v2_deep as our CNN of choice. The deeper architecture was able to capture more complex patterns, with the progressive filter growth learning hierarchical features well. The heavy regularization prevented overfitting, with the much closer train/val performance showing true learning.

CNN Results

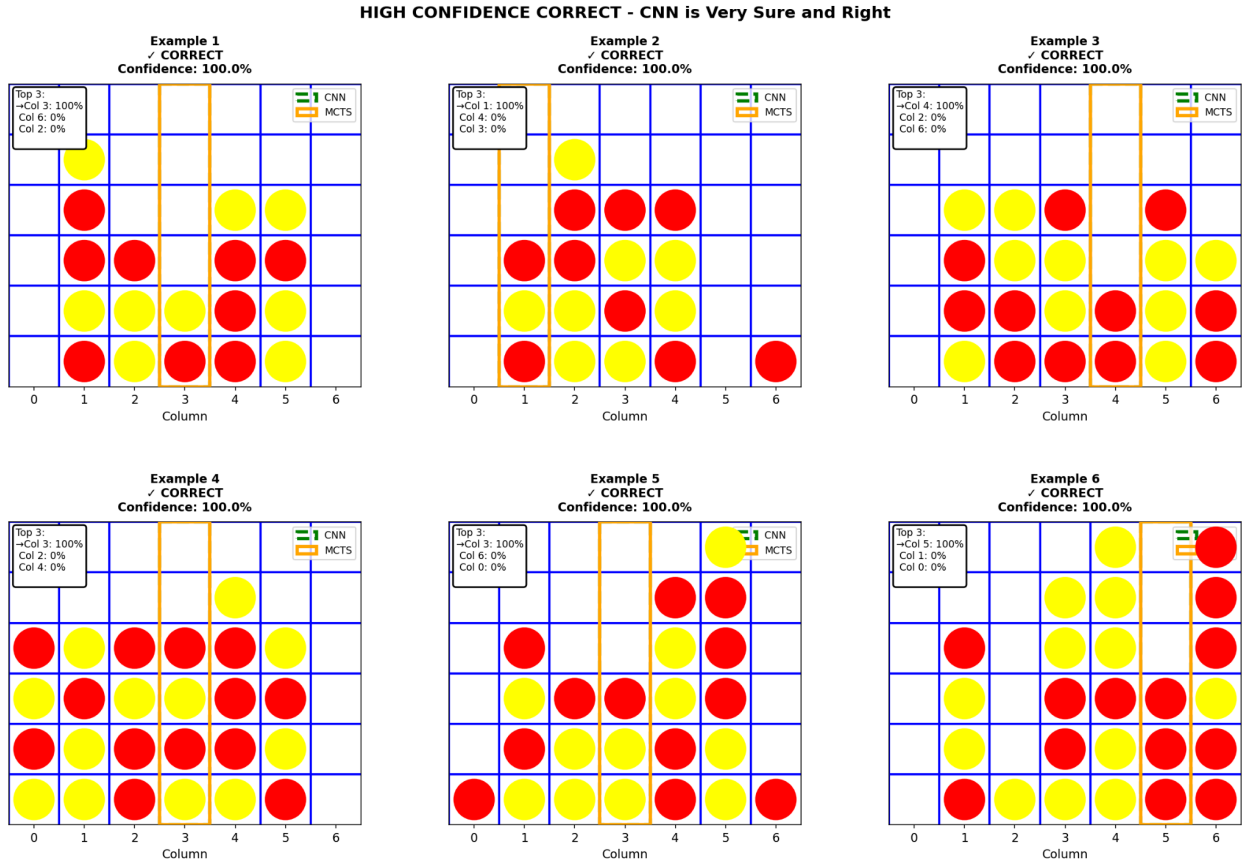
Test Performance

- Accuracy: 64.1% (CNN agrees with MCTS 64% of the time)

- Confidence on correct predictions: 76%
- Confidence on incorrect predictions: 50%

What the CNN Learned

Easy Boards (High Confidence, Correct)

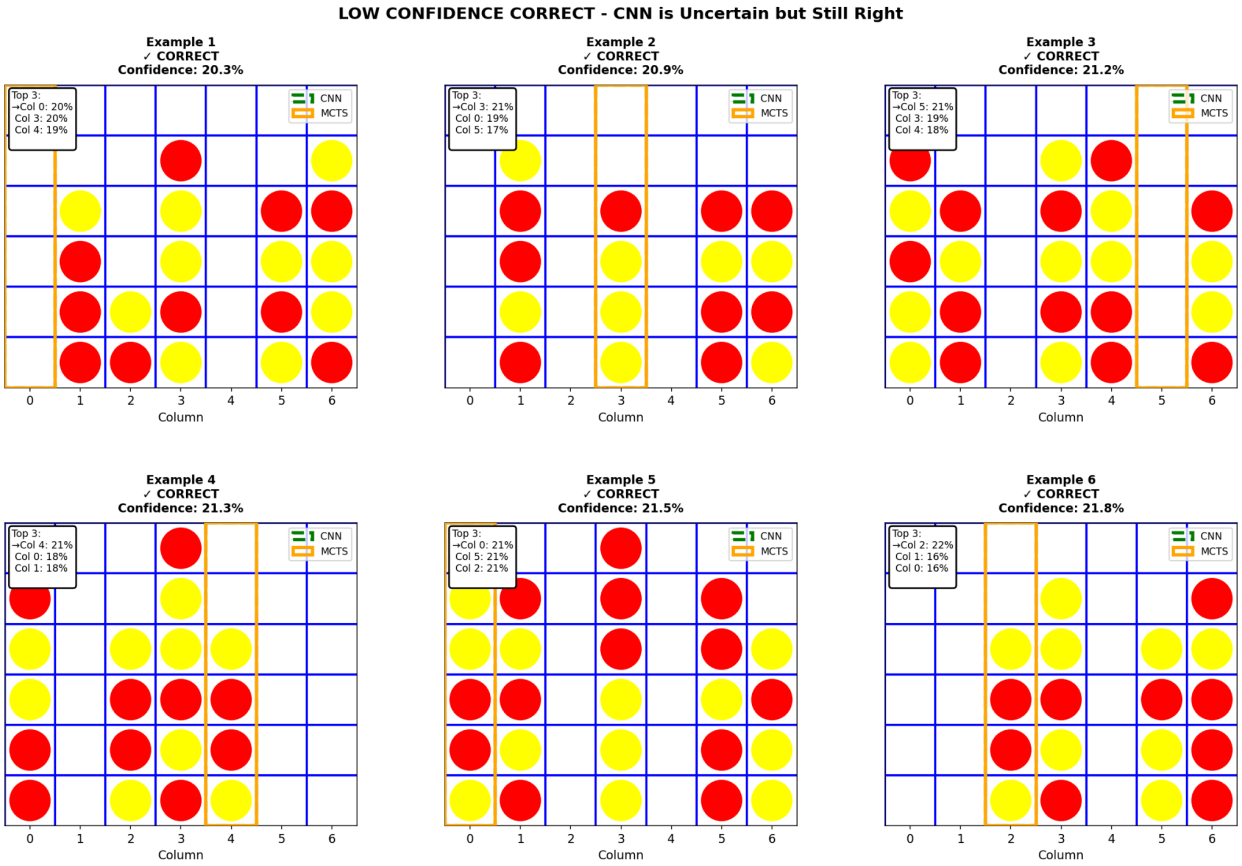


The CNN confidently (>80%) handles:

- Immediate threats: Three in a row that must be blocked
- Winning moves: Three in a row that need completion
- Opening moves: Center control in early game

These are pattern-matching tasks where CNNs excel.

Difficult Boards (Low Confidence, Correct)

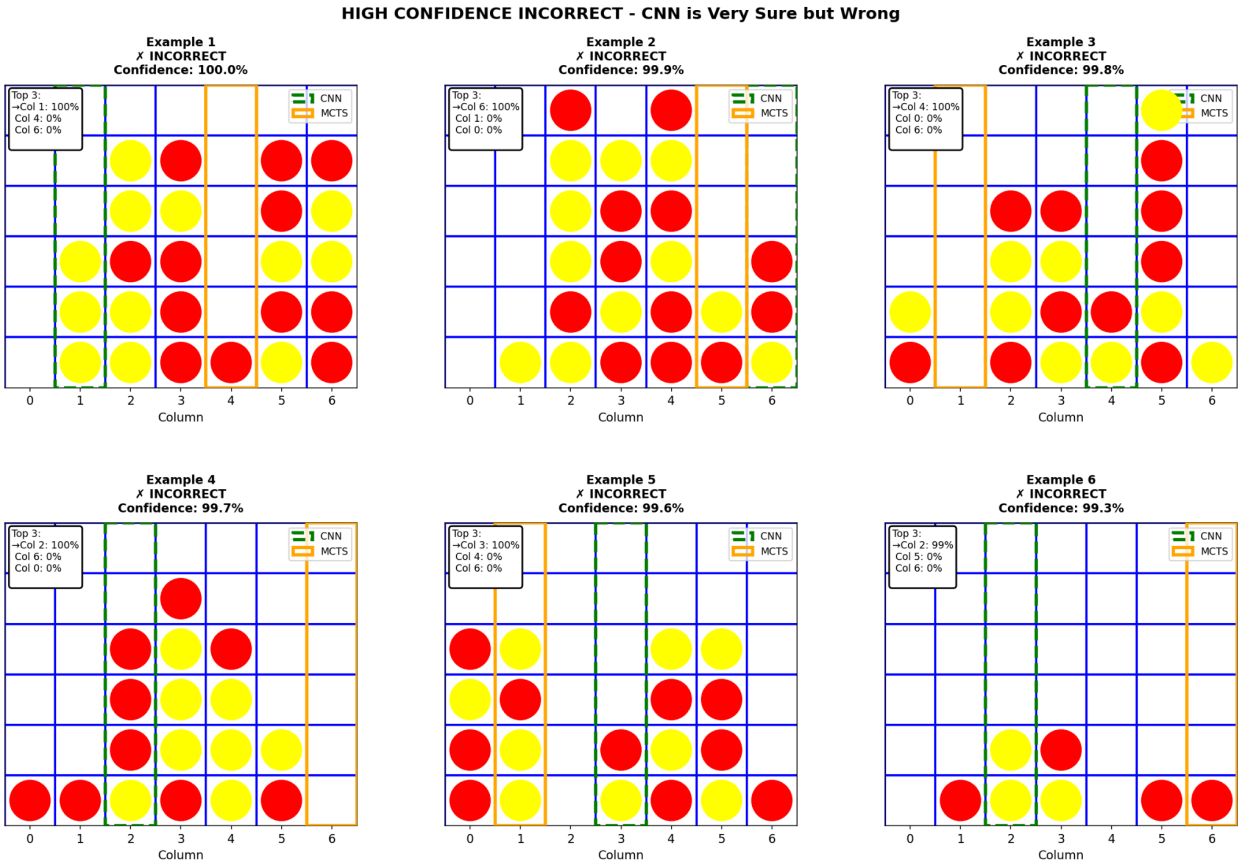


The CNN struggles with (<35% confidence):

- Complex mid-game positions: 12-20 pieces, no immediate threats
- Multiple good options: Where several moves are similarly valid
- Strategic setup moves: Requires multi-move planning, not just pattern matching

Low confidence often means the position genuinely has multiple reasonable moves.

Mistakes (High Confidence, Incorrect)

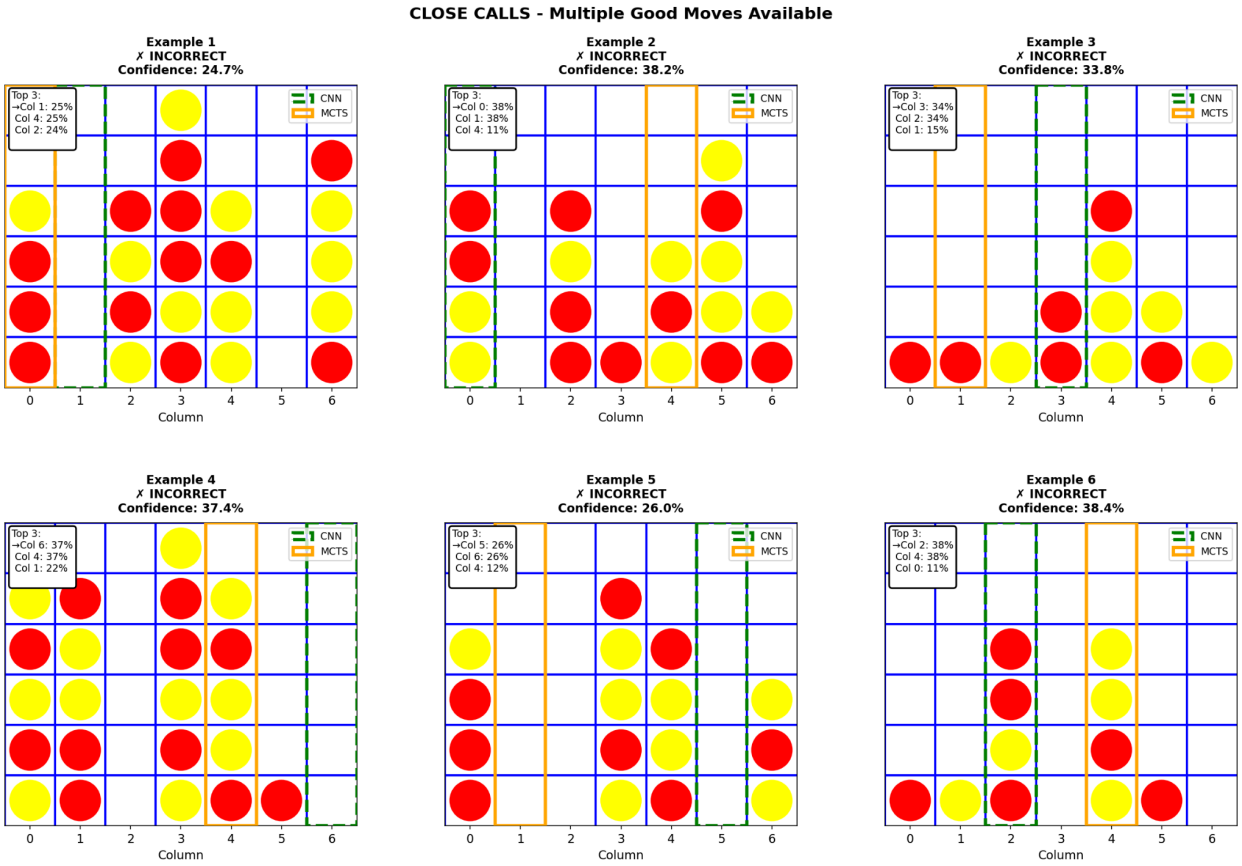


The CNN makes confident mistakes on:

- Fork opportunities: Misses subtle setups that create two ways to win
- Multi-front threats: Focuses on one threat while missing a bigger one elsewhere
- Late-game positions: Requires calculation over pattern recognition

These errors reveal CNN's limitation: it's great at "what's on the board now" but struggles with "what could happen in 3-4 moves."

Close Calls



Positions where the CNN's top two choices are within 15% probability. Often, both moves are actually good - this highlights that our accuracy metric (agreement with MCTS) isn't perfect. MCTS isn't always right, and sometimes multiple moves lead to similar outcomes.

Why Some Boards Are Hard for CNN

1. Multi-move planning: CNN recognizes current patterns but can't search ahead 3-4 moves like MCTS
2. Multiple valid options: When several moves are equally good, disagreeing with MCTS doesn't mean the CNN is "wrong"
3. Limited training examples: 215K samples can't cover billions of possible positions; novel boards are harder
4. Spread-out patterns: Very dispersed threats across the board are harder to detect than compact local patterns

At 64% accuracy, the CNN learned genuine Connect 4 strategy such as threat recognition, blocking, center control, and winning patterns. Its mistakes are instructive, showing the boundary between pattern matching (where it excels) and multi-move planning (where search algorithms dominate).

The architectural exploration validated that depth and regularization matter. CNN_v2_deep's superior performance over the baseline (64% vs 58%) came from going deeper while preventing overfitting

through L2 regularization and dropout. The multi-kernel experiment (CNN_v3) showed that on small grids like Connect 4, simple 3×3 filters are sufficient and larger receptive fields don't add value.

Transformer

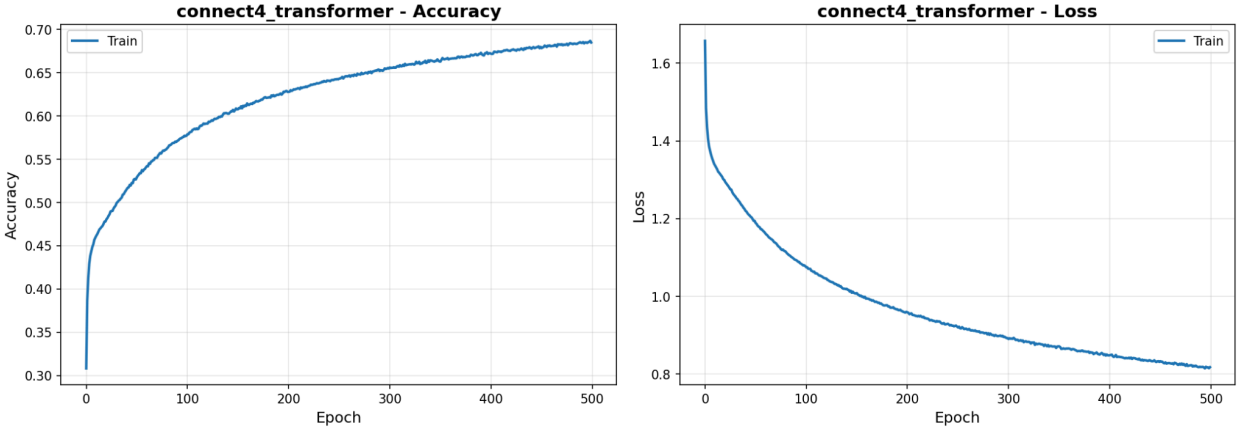
For the transformer, we decided to divide the board into overlapping six 3x3x2 subimages and use additive positional encoding. Next, to find the optimal hyperparameters, we ran a sweep using WandB to obtain the best architecture for training:

- *Number of Layers: 4*
- *Embedded dimensions: 128*
- *Heads: 4*
- *MLP dimensions: 128*
- *Dropout Rate: 0.15*
- *Learning Rate: 0.00045*
- *Batch Size: 256*

Transformer Results

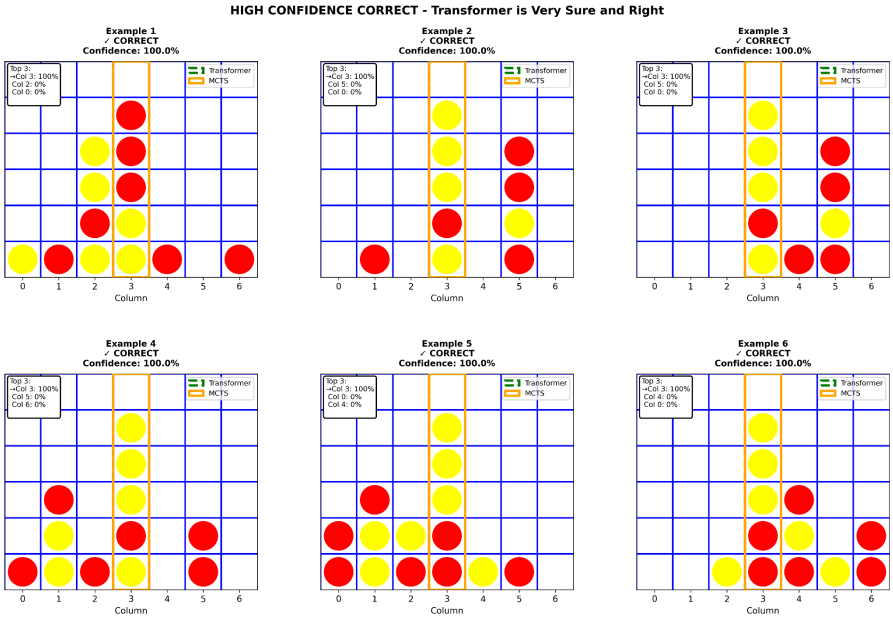
Performance

- Accuracy: 70% after training of the full dataset for 500 epochs
- Model agrees with MCTS 80% of the time
- Model has a win rate of 31% against MCTS with a step count of 500



What the Transformer Learned

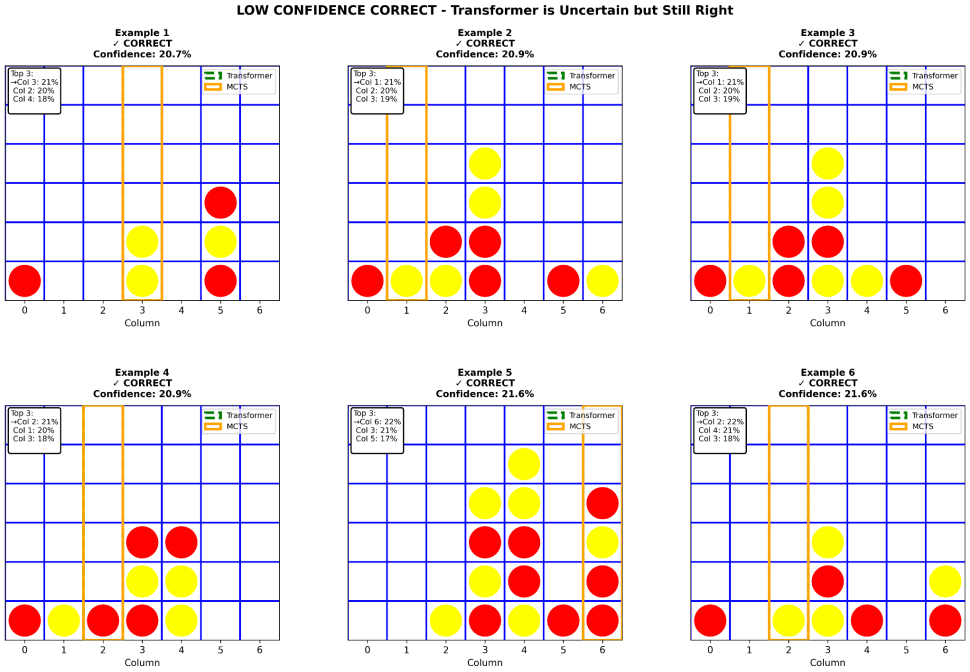
Easy Boards (High Confidence, Correct)



The Transformer handles several types of boards with certainty:

- Immediate threats: Three in a row that must be blocked
- Winning moves: Three in a row that need completion
- Opening moves: Center control in early game

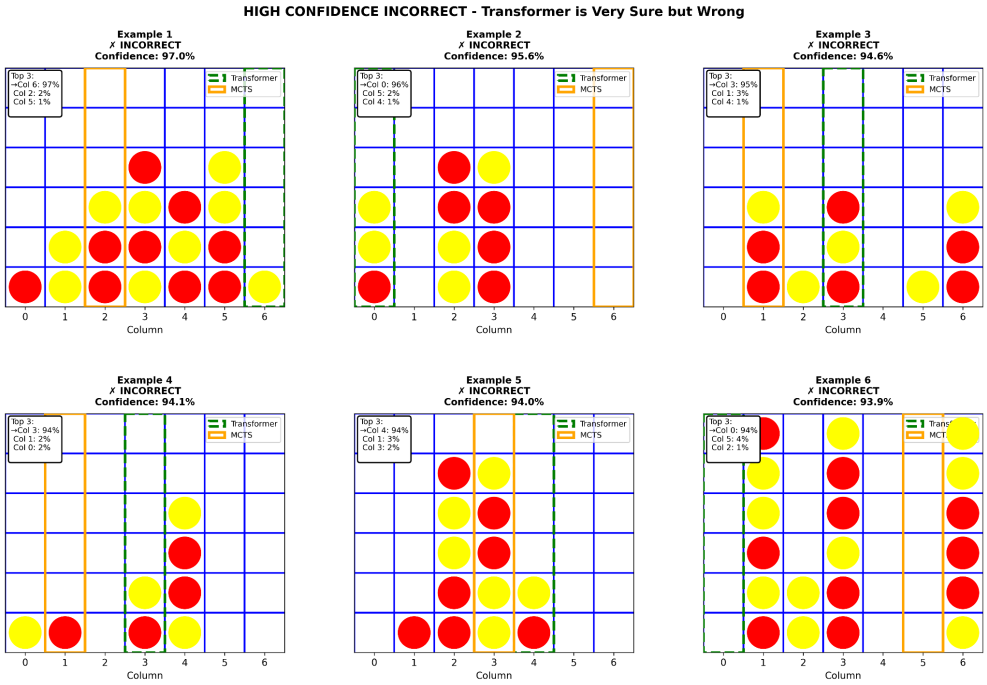
Difficult Boards (Low Confidence, Correct)



The Transformer struggles with and is less certain when dealing with:

- Post opening moves: Transformer could take multiple possible good moves
- Midgame strategies: Transformer lacks direction and forward planning
- Strategic setup moves: Requires multi-move planning, not just pattern matching

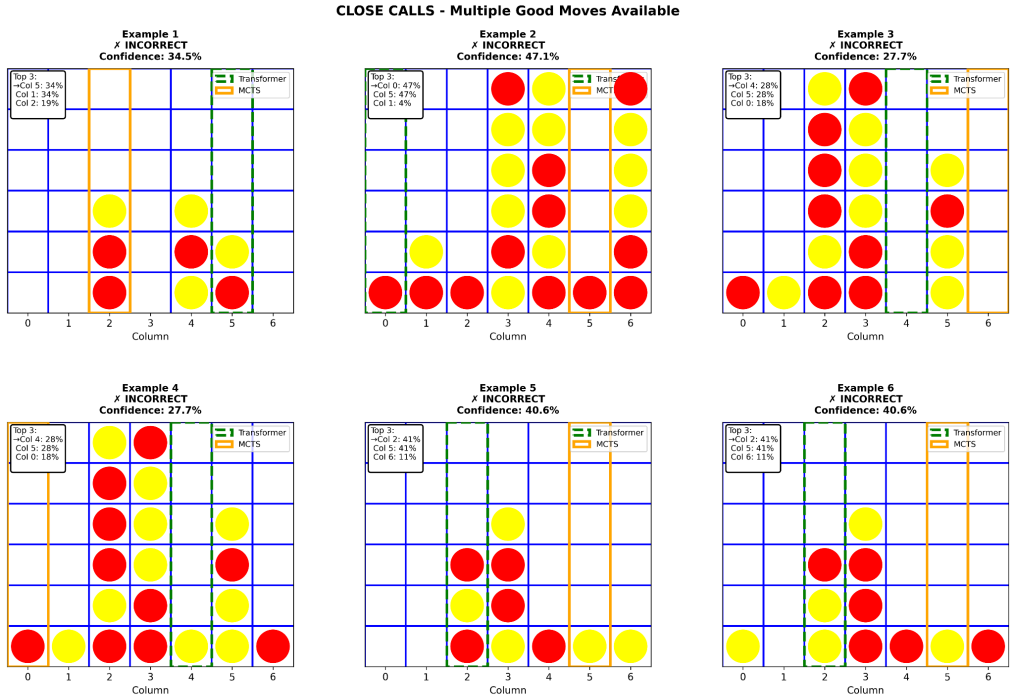
Mistakes (High Confidence, Incorrect)



The Transformer is confidently incorrect when:

- Throwaway moves: Struggles dealing with making moves which don't aid in the opponents win
- Multi-front threats: Misprioritizes the bigger threat for a smaller one

Close Calls



Positions where the Transformers' top two choices are within 15% probability. Often, both moves are actually good - this again highlights that our accuracy metric (agreement with MCTS) isn't perfect, additionally both the transformer and CNN both seem to struggle with earlier game moves as they lack direction and a direct win condition.

Comparative Summary and Model Behavior Analysis

While both architectures were trained on the same underlying game data, their learning dynamics and decision-making behaviors differed in meaningful ways.

The CNN architecture demonstrated strong performance in spatially obvious board states. Because convolutional filters specialize in detecting local spatial patterns, the model performed well in situations involving clear horizontal or vertical threats. Its inductive bias toward locality made it particularly effective at recognizing immediate win or block configurations.

The Transformer model, by contrast, exhibited stronger long-range reasoning behavior. Self-attention allowed it to evaluate relationships across the entire board simultaneously, enabling it to identify multi-step tactical threats more effectively in complex mid-game positions. However, this increased flexibility also introduced sensitivity to training distribution and positional encoding design.

In simpler board states, both models performed comparably. In highly ambiguous or multi-threat configurations, misclassifications were often linked to:

- Sparse representation of similar board states in training data
- Conflicting reward signals during training
- Ambiguity between defensive and offensive priorities

Overall, the CNN provided stable, computationally efficient performance with strong local pattern recognition, while the Transformer demonstrated higher strategic depth at the cost of increased architectural complexity and deployment considerations.

A Live Connect 4 System: Deployment, Integration, and Engineering

Training a neural network to play Connect 4 is one challenge. Turning that network into a live, interactive system that people can actually play against is another.

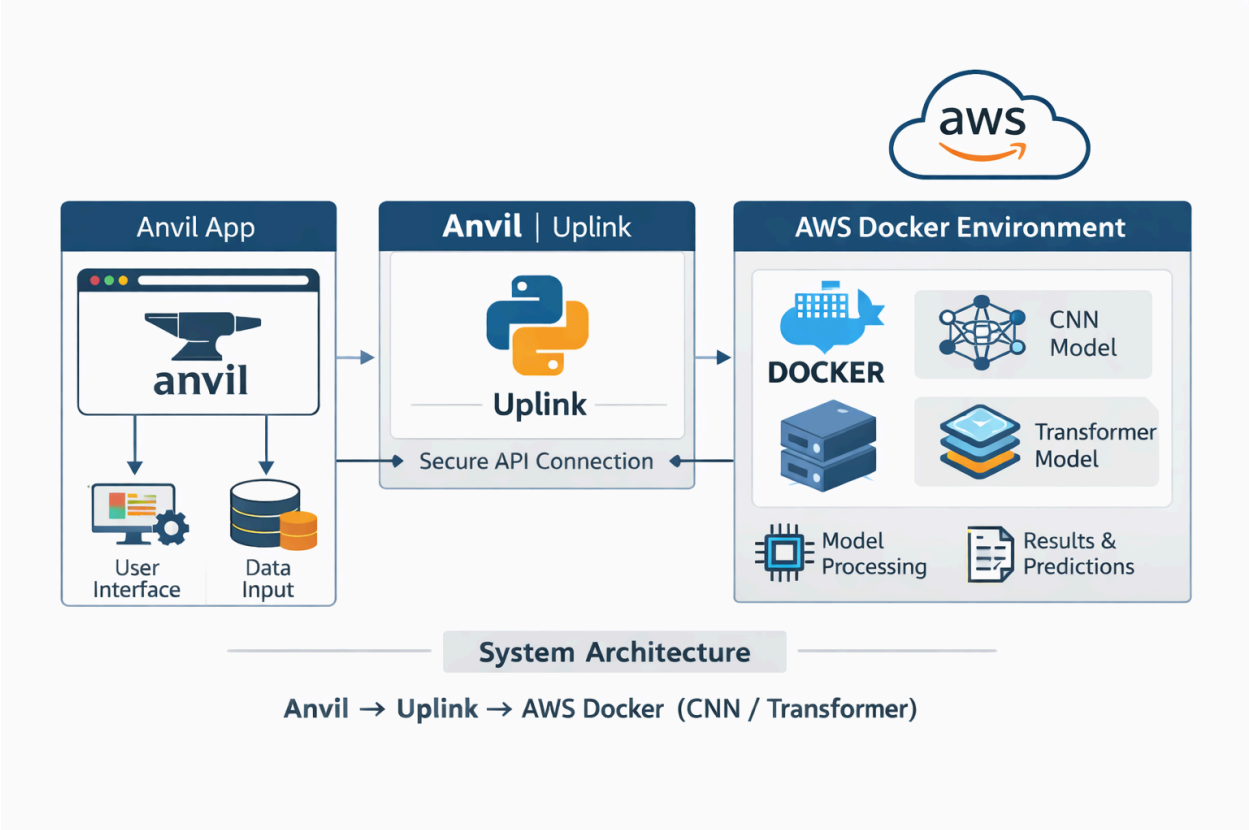
After finalizing the CNN and Transformer architectures described above, the next step was to operationalize them. This required building a production-style pipeline that could host both models, handle inference in real time, and integrate seamlessly with a frontend interface. What followed was a multi-stage engineering effort involving Anvil, AWS Lightsail, Docker, and extensive debugging to ensure cross-version compatibility.

This section documents how we transformed trained models into a working ML application.

System Architecture

At a high level, the deployed system consists of four components:

- **Anvil** - Frontend UI and application logic
- **Anvil Uplink** - Secure communication channel
- **AWS Lightsail (Ubuntu)** - Cloud compute host
- **Docker Containers** - Isolated model-serving environments



Instead of running inference locally, each move request is routed through Anvil's Uplink to a Docker container hosted on AWS. The container loads the model into memory and returns the predicted move. This separation keeps the frontend lightweight and ensures reproducibility of the model environment.

We deployed two separate inference containers:

- **CNN Container (Hard difficulty)**
- **Transformer Container (Medium difficulty)**

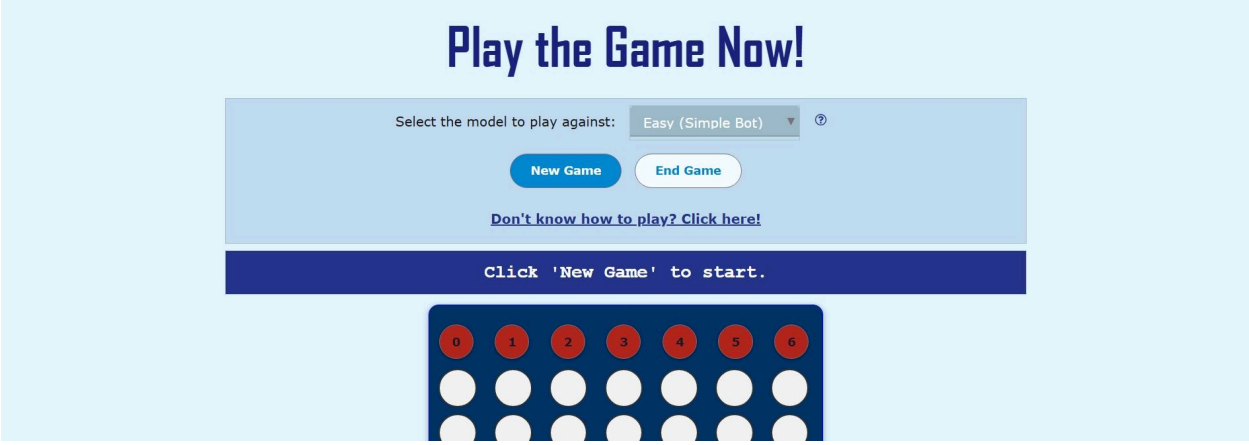
Additionally, a simple heuristic bot was retained for **Easy difficulty**, providing a baseline comparison.

Engineering the Frontend

The Anvil frontend was built to reflect the game mechanics clearly while maintaining strict separation between UI logic and backend inference.

UI Decisions

The grid is rendered dynamically as a 6×7 board. Users do not click grid cells directly; instead, they click numbered column buttons (0-6). This design choice prevents ambiguity in move interpretation and ensures clean mapping to backend logic.



Game statistics are tracked separately for each difficulty level. This allows direct experiential comparison between the CNN and Transformer models.

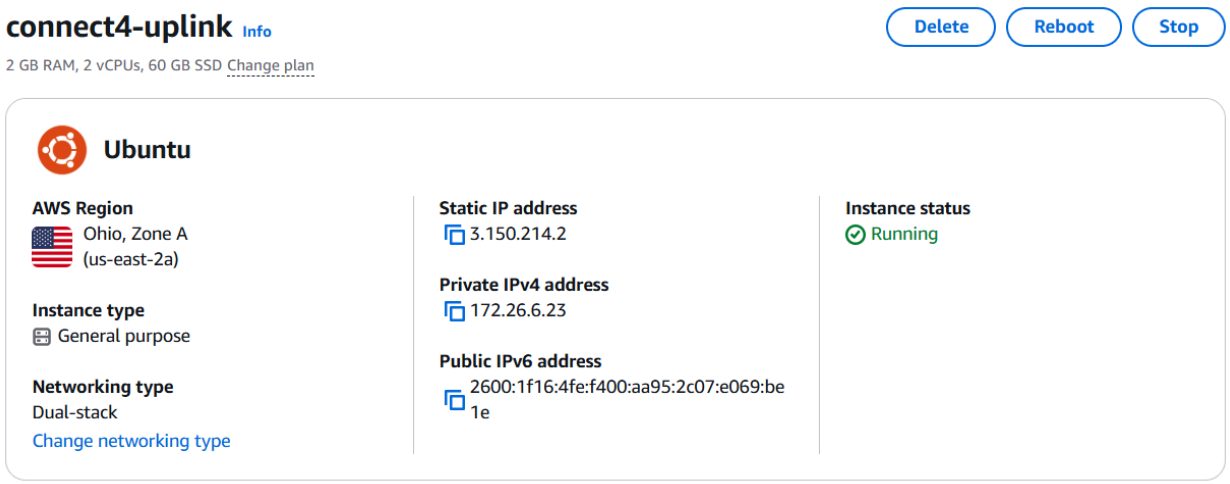
To improve usability:

- The UI locks during inference to prevent double moves.
- A brief "Bot thinking…" delay is shown for realism.
- End-of-game popups summarize results and updated statistics.
- Mobile users are restricted (the interface is optimized for desktop play).

These choices ensure that the user experience supports evaluation of the neural networks rather than distracting from them.

Deploying to AWS Lightsail

We chose **AWS Lightsail** for its simplicity and persistent compute capabilities. A lightweight Ubuntu instance was provisioned, Docker installed, and firewall rules configured.



Since no GPU was available, all inference runs on CPU. TensorFlow emitted CUDA-related warnings, but these were expected and harmless in a CPU-only environment.

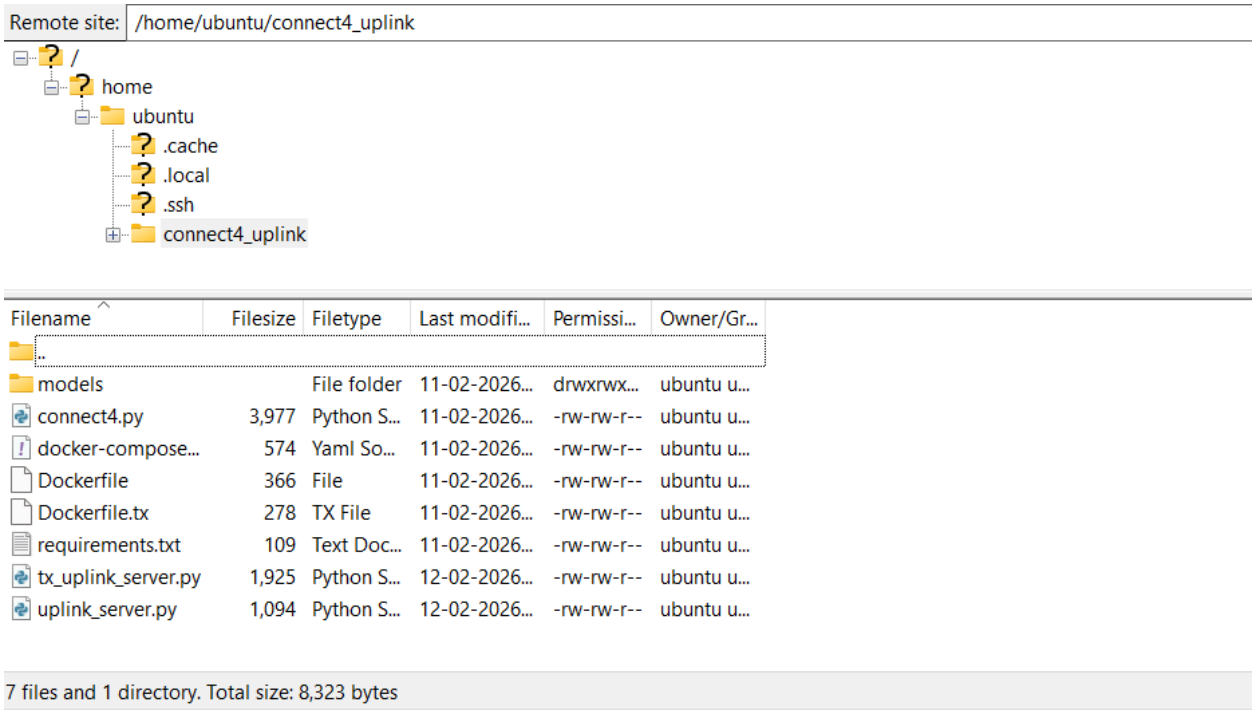
Memory usage during CNN loading required monitoring, particularly because the CNN architecture is relatively parameter-heavy. However, after optimization, both models load reliably at container startup.

Dockerization

One major design decision was to separate the CNN and Transformer into different Docker containers rather than combining them.

We created:

- `Dockerfile` → CNN container
- `Dockerfile.tx` → Transformer container
- `uplink_server.py` → CNN inference endpoint
- `tx_uplink_server.py` → Transformer inference endpoint



This separation proved crucial. The two models had different serialization formats, dependency sensitivities, and loading requirements. Keeping them isolated simplified debugging and prevented cross-contamination of environments.

CNN Deployment

The CNN model was saved in `.h5` format under a different TensorFlow/Keras version than the deployment environment. This resulted in several errors during deserialization, including:

- `batch_shape` argument mismatches
- `dtype` policy issues
- incompatibility with newer Keras versions

To resolve this, we:

- Loaded the model with `compile=False`
- Adjusted compatibility layers
- Ensured consistent TensorFlow versions in Docker

Once patched, the CNN loaded successfully and performed inference consistently.

```
ubuntu@ip-172-26-6-23:~/connect4_uplink$ docker logs -f anvil-uplink
2026-02-13 02:20:26.920097: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared object file: No such file or directory
2026-02-13 02:20:26.920132: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
Connecting to wss://anvil.works/uplink
Anvil websocket open
Connected to "Default Environment" as SERVER
🔗 CNN uplink connected. Waiting for calls...
Human played 0
Loading CNN model from /models/CNN_v2_deep_best.h5...
2026-02-13 02:22:57.893942: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.1: cannot open shared object file: No such file or directory
2026-02-13 02:22:57.893983: W tensorflow/stream_executor/cuda/cuda_driver.cc:269] failed call to cuInit: UNKNOWN ERROR (303)
2026-02-13 02:22:57.894006: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (d014ac290175): /proc/driver/nvidia/version does not exist
2026-02-13 02:22:57.895486: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2026-02-13 02:22:58.073462: W tensorflow/core/framework/cpu_allocator_impl.cc:82] Allocation of 44040192 exceeds 10% of free system memory.
2026-02-13 02:22:58.125769: W tensorflow/core/framework/cpu_allocator_impl.cc:82] Allocation of 44040192 exceeds 10% of free system memory.
2026-02-13 02:22:58.147524: W tensorflow/core/framework/cpu_allocator_impl.cc:82] Allocation of 44040192 exceeds 10% of free system memory.
2026-02-13 02:22:58.571931: W tensorflow/core/framework/cpu_allocator_impl.cc:82] Allocation of 44040192 exceeds 10% of free system memory.
✓ CNN loaded!
CNN Bot played 3
Human played 6
CNN Bot played 3
Human played 0
CNN Bot played 3
Human played 3
```

This experience highlighted how tightly coupled `.h5` models can be to their training environment.

Transformer Deployment

The Transformer model posed a more complex challenge.

The original `.keras` model file failed to load due to:

- Unknown custom layers
- `dtype` policy errors
- Cross-version serialization incompatibilities
- Keras 3 vs TensorFlow 2.x conflicts

Rather than attempting to manually register and patch every incompatibility, we returned to the original training environment (Google Colab) and re-exported the model using TensorFlow's SavedModel format:

```
tf.saved_model.save(model, "tx_savedmodel")
```

The SavedModel directory structure proved far more robust across environments.

```
ubuntu@ip-172-26-6-23:~/connect4_uplink$ docker logs -f anvil-uplink-tx
2026-02-14 21:07:02.273341: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:32] Could not find cuda drivers on your machine, GPU will
not be used.
2026-02-14 21:07:02.545696: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:32] Could not find cuda drivers on your machine, GPU will
not be used.
2026-02-14 21:07:02.839988: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:467] Unable to register cuFFT factory: Attempting
 to register factory for plugin cuFFT when one has already been registered
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
E0000 00:00:1771103223.068226       1 cuda_dnn.cc:8579] Unable to register cuDNN factory: Attempting to register factory for plugin cuD
NN when one has already been registered
E0000 00:00:1771103223.133594       1 cuda_blas.cc:1407] Unable to register cuBLAS factory: Attempting to register factory for plugin c
uBLAS when one has already been registered
W0000 00:00:1771103223.579191       1 computation_placer.cc:177] computation placer already registered. Please check linkage and avoid
linking the same target more than once.
W0000 00:00:1771103223.579258       1 computation_placer.cc:177] computation placer already registered. Please check linkage and avoid
linking the same target more than once.
W0000 00:00:1771103223.579262       1 computation_placer.cc:177] computation placer already registered. Please check linkage and avoid
linking the same target more than once.
W0000 00:00:1771103223.579265       1 computation_placer.cc:177] computation placer already registered. Please check linkage and avoid
linking the same target more than once.
2026-02-14 21:07:03.585443: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available C
PU instructions in performance-critical operations.
To enable the following instructions: AVX2 AVX512F FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
Connecting to wss://anvil.works/uplink
Anvil websocket open
Connected to "Default Environment" as SERVER
 TX uplink connected. Waiting for calls...
Loading Transformer SavedModel from /models/tx_savedmodel...
2026-02-14 21:07:32.889104: E external/local_xla/xla/stream_executor/cuda/cuda_platform.cc:51] failed call to cuInit: INTERNAL: CUDA er
ror: Failed call to cuInit: UNKNOWN ERROR (303)
Using input: input_layer
Using output: output_0
✓ Transformer SavedModel loaded!
Human played 6
```

This eliminated the need for custom object registration and resolved version conflicts cleanly.

The key lesson: **SavedModel is significantly more portable than `.keras` for cross-environment deployment.**

Logging and Real-Time Inference Monitoring

To validate inference correctness, we implemented turn-by-turn logging in both containers:

- `Human played [column]`
- `Transformer/CNN Bot played [column]`

```
✓ Transformer SavedModel loaded!
Human played 6
Transformer Bot played 3
Human played 1
Transformer Bot played 3
Human played 3
Transformer Bot played 2
Human played 1
Transformer Bot played 4
Human played 5
Transformer Bot played 4
Human played 1
Transformer Bot played 1
Human played 2
Transformer Bot played 6
Human played 5
Transformer Bot played 4
```

This allowed us to:

- Debug invalid move cases
- Confirm legal move masking
- Observe difficulty differences between models

The Transformer displayed more balanced positional play, while the CNN exhibited stronger defensive blocking patterns - consistent with our training observations.

Bringing It All Together

When a user plays a move:

1. The frontend validates the column.
2. The board state is updated locally.
3. The UI locks.
4. The board is sent to the AWS container.
5. The selected model predicts the best legal move.
6. The move is returned and rendered.
7. Stats update and results display if the game ends.

The final system integrates neural network inference with UI responsiveness and cloud-hosted reliability.

Reflection: Beyond Training

The earlier sections of this report focus on architecture design, hyperparameter tuning, and model performance. This deployment phase revealed a different layer of machine learning engineering:

- Serialization format matters.
- Version consistency matters.
- Docker isolation simplifies reproducibility.
- SavedModel is the one of the safest export strategies for Transformers.
- UI decisions affect inference behavior.

Ultimately, this project evolved from training neural networks to building a complete ML-powered application.

The CNN and Transformer are no longer just trained models - they are interactive agents capable of playing Connect 4 in real time within a production-style environment.

Conclusion

This project began with a simple goal: train neural networks to play Connect 4. Through experimentation, we developed both a CNN that captures local board patterns and a Transformer that reasons more globally about game state.

But the real impact came from turning those models into a live system. By integrating them into a web application and deploying them in a production-style environment, we moved beyond training into real-world machine learning engineering.

In the end, this project was not just about model performance - it was about building an interactive, end-to-end ML-powered experience.
