Project 1 Report

Optimization II

Prof. Daniel Mitchell

Team: Jenna Ferguson, Mallika Singh, John Hwang, Nawen Deng

February 17th, 2025

Project Overview

Connect 4 is a game of strategy, foresight, and pattern recognition, one that has been mathematically solved, meaning a perfect game can be played with optimal moves every time. However, in this project, we are not aiming to replicate perfection. Instead, our goal is to train neural networks to predict the best possible move in any given board position, not through brute-force calculations, but by learning from the patterns of strong gameplay. By using Monte Carlo Tree Search (MCTS) to simulate competitive matches, we create a dataset of board states and corresponding recommended moves, forming the foundation for our model's training. The value of this approach lies in adaptability: rather than following a rigid, predetermined solution, our models will learn to recognize and respond to dynamic board states, making them more flexible when faced with real gameplay scenarios. This project develops both a Convolutional Neural Network and a Transformer capable of understanding and playing Connect 4 at a high level, providing insights into game strategy and decision-making through machine learning.

Generate Training Data

In order to build a model capable of providing valuable insights, it must be trained on valuable data. If this dataset were based on our own gameplays, the model wouldn't have much to learn, except maybe how to lose creatively. Nor did we have the time (or skill) to generate a truly valuable dataset. We instead used the logic of the Monte Carlo Tree Search algorithm to determine the best moves possible for a player given the results of many game situations.

In an ideal world, we could find the best move for a given board state in Connect 4 by searching every possible sequence of moves until the end of the game, ensuring that we always play optimally. This is the idea behind Depth-First Search (DFS), which is a brute-force method that explores every possible path to determine the best one. However, Connect 4 has about 4.5 trillion game states, making it computationally impossible to search through every scenario efficiently. At that rate, we'd risk our opponent celebrating their retirement before the game ever ends. Even though the game is solved, finding the perfect move for every board state this way would take far too long, especially for generating a large dataset.

This is where Monte Carlo Tree Search (MCTS) comes in. Instead of trying to evaluate every single move like DFS does, MCTS plays out a large number of simulated games from a given position, using random or strategic moves to see what the result of each potential decision might be. It then uses the results of these simulations to estimate which move is the strongest. The logic behind MCTS for Connect 4 can be broken down into four key steps:

Selection – Starting from the current board position, MCTS selects moves that seem promising based on previous simulations.
Expansion – If the selected move hasn't been explored enough, the search tree is expanded by adding new possible moves.
Simulation – From the newly added positions, the game is played out randomly (or semi-randomly) to estimate how good or bad that position is.
Backpropagation – The results of these simulated games are then used to update the earlier moves, reinforcing strong decisions and discarding weaker ones.

We used this MCTS logic to generate a dataset of board states and their best-recommended move. We now discuss the design choices we made to ensure variety and reliability in our data.

NSteps

First, we set our number of steps (nsteps) to be 5,000, meaning that for each move, MCTS simulated 5,000 possible games before making a decision. Nsteps is the skill level of the MCTS, and a larger number of nsteps means that the player considers more things before making a decision, but it also means the player is slower. While nsteps of 5000 was more time consuming, we felt it would result in a stronger and more strategic model. After all, a model is only as good as the data it learns from. This ensured that our dataset wasn't filled with completely random or weak moves but rather decisions based on well-explored gameplay.

Efficiency

The MCTS function makes a few optimizations to improve efficiency before running full simulations. Instead of blindly exploring thousands of game possibilities, it first checks for immediate wins. The function calls a look_for_win function, which scans the board to see if the current player has a winning move they could play. If so, the MCTS instantly returns that column and no longer has to simulate thousands of games since there is an immediately apparent victory. If no winning move is found, the function then calls a find_all_nonlosers function, which filters out moves that would immediately allow the opponent to win on their next turn. By eliminating these "self-sabotaging" moves early, we avoid wasting computational power to simulate an already doomed game. However, if all remaining moves would eventually lead to a loss, the function defaults to find_legal(board), selecting the "best losing move" to at least delay defeat. This approach ensures that we significantly reduce the number of unnecessary simulations while still making strong decisions.

Randomness

To maintain some diversity in our board, we introduced randomness at the start of each game by playing 0 to 3 random moves before MCTS determined the moves. This randomness is crucial because, without it, MCTS would always start from the exact same empty board, leading to a dataset that lacks variety. This forces MCTS to adapt to different scenarios rather than memorizing a fixed pattern. We choose randomly between 0 and 3 random initial moves to introduce enough variation to make the dataset diverse, but not so much that the board becomes chaotic before MCTS has a chance to make decisions. This helps make our model more robust by exposing it to a broader set of possible board positions.

Data Preparation

During the data generation process, we stored 3 things in a dictionary. Each dictionary contained the board state, the recommended move, and the player (who would be playing the recommended move). The MCTS logic was based on a 6x7 array board, but before it was added to the dataset it was converted to a 6x7x2 array. Using a 6x7x2 board state provided a clearer and more structured input for the CNN by separating the two players' moves into distinct channels, similar to how color channels work in image processing. This made it easier for the network to recognize patterns, as it didn't have to infer piece ownership from a single-layer representation. Each position is represented by two layers:

1) The first layer ([:,:,0]) marks where player +1 (the "plus" player) has pieces.
2) The second layer ([:,:,1]) marks where player -1 (the "minus" player) has pieces.
3) Empty spaces are 0 in both layers.

Although our models were ultimately trained to play as the 'plus' player, we chose to store moves for both players. Later, we standardized the dataset by converting all 'minus' boards into 'plus' boards, essentially just flipping the board.

Our final dataset consisted of 6x7x2 NumPy arrays, each representing the board state when it was the 'plus' player's turn, and the column that MCTS recommended as the best move.

Train CNN

When it comes to recognizing patterns and making smart decisions, Convolutional Neural Networks (CNNs) are ideal for visual learning. Even though they were originally designed for tasks like image recognition, they are surprisingly good at board games too. Why? Because a Connect 4 board is essentially an image: a grid where patterns emerge as players strategically drop their pieces. Similar to how a CNN can identify cats in photos by spotting whiskers and ears, it can learn to identify winning patterns, traps, and strategic setups on the Connect 4 board.

CNNs work by scanning small sections of the board at a time, using filters to detect important features such as a nearly completed four-in-a-row or a strategic setup for a future win. Instead of treating the board as a flat list of numbers, they recognize spatial relationships, which is crucial in a game where positioning matters a lot. By training our CNN on MCTS-generated data, we aimed to build a model that doesn't just memorize moves but actually understands the flow of the game, developing instincts that mimic strong human play. Our goal was to train a competitor, not a clueless beginner!

Our strategy for tuning our parameters focused on a subset of our data. If the model performed well on the subset, we would apply it to the full dataset (not including the validation set). This helped us with computational efficiency and allowed us to tune more things in a timely manner to get an overall idea of what was working well.

Pre-Processing

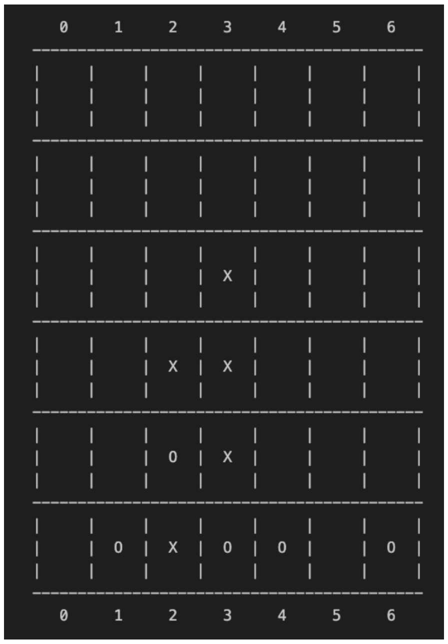
To prepare our data for the CNN, we first converted each board state into a NumPy array and stored the corresponding recommended move as a label. We then applied data augmentation by flipping the board horizontally, which doubled the dataset. Strategies on the left side of the board are equally valid on the right, so this flipping helped to ensure the model doesn’t favor one side and learned to recognize winning patterns and threats equally across the board. Since the board has 7 possible columns to play in, we one-hot encoded the labels, making it easier for the CNN to classify the best move as a probability distribution over all columns. Our final dataset contains 2,381,118 unique board positions, each represented as a 6x7x2 grid, with corresponding labels as one-hot encoded vectors of size 7, indicating the best column to play based on MCTS recommendations.

Finally, we split the dataset into training and validation sets, ensuring that the model is tested on unseen boards during training. 80% of the boards are used for training while, 20% are held for validation, allowing us to assess the model's performance on unseen data. The sklearn.model_selection.train_test_split function shuffles the data by default, which is crucial so that our validation is not an ordered subset of games.

Convolutional Layers

Filters

Our initial model used 3x3 filters for all convolutional layers in an attempt to capture fine details. A 3x3 filter can detect small-scale patterns, such as adjacent pieces forming potential four-in-a-row sequences, while also allowing deeper layers to combine these features into more complex strategic insights. This model had a higher accuracy (~78% validation accuracy), but a few inherent flaws. When playing against this model, we found that it was very good at playing strategically in later parts of the game and that its opponent was mostly playing defensively. Unfortunately, it was really easy to beat in early moves. The example below shows one of the examples of this, a game where the player played column 3 > 4 > 1 > 2 > 6, leaving the 5 column open for the opponent to place a winning move. The bot did not block this move, but instead played column 3. We decided to experiment with filter sizes to combat this weakness.



```
   0     1     2     3     4     5     6
-----------------------------------------
|     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |
-----------------------------------------
|     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |
-----------------------------------------
|     |     |     |     |     |     |     |
|     |     |     |  X  |     |     |     |
|     |     |     |     |     |     |     |
-----------------------------------------
|     |     |     |     |     |     |     |
|     |     |  X  |  X  |     |     |     |
|     |     |     |     |     |     |     |
-----------------------------------------
|     |     |     |     |     |     |     |
|     |     |  O  |  X  |     |     |     |
|     |     |     |     |     |     |     |
-----------------------------------------
|     |     |     |     |     |     |     |
|     |  O  |  X  |  O  |  O  |     |  O  |
|     |     |     |     |     |     |     |
-----------------------------------------
   0     1     2     3     4     5     6
```

We then changed the initial filter size to 4x4. While the 3x3 filter seemed ideal for fine details, in Connect 4, strategic patterns often extend beyond just three adjacent positions. A 4x4 filter allowed the CNN to understand positional relationships across a wider area, making it better at spotting those early on threats while still setting up strong plays.

After multiple iterations of adjusting filter sizes, our best-performing model used a progressive reduction in filter sizes, allowing the network to capture both broad and fine-grained spatial relationships in the board. The first convolutional block uses 5x5 filters, which help extract high-level game patterns, such as overall board structure and early-stage positioning. The second block transitions to 4x4 filters, allowing the model to focus on medium-range relationships, such as piece formations and potential traps. Finally, the third block further reduces to 4x4 and 3x3 filters, refining the model's ability to detect precise move sequences, blocking strategies, and immediate threats. We found this model to effectively combat early wins while also setting up for a strategic game that keeps its opponent on defense. This structured approach ensured that early layers focus on general strategy, while deeper layers learned tactical, board-specific interactions. We think this was one of the key components that determined the success of our CNN.

Number of Filters

In our initial model, we used 64 filters across all blocks in order to reduce model complexity and manage overfitting. We found that this model struggled to capture deeper strategic patterns, and that the model was not a very strong opponent for any player that approached the game with a sense of strategy.

In the next iteration, we doubled the filters per block (64-128-256), and found that this increased the training time greatly and was leading to overfitting. We decided to scale back a bit and see if changes elsewhere in our model could lead to improvement without this level of complexity.

In our final model, we progressively increase the number of filters from 64 in the first convolutional block to 128 in the later blocks, hoping to capture increasingly complex patterns in the Connect 4 board. This gradual increase helped us balance computation and feature extraction to ideally identify broad strategic play and precise tactical moves.

Activation Function

We used different activation functions at various stages. In the convolutional layers, we use ReLU (Rectified Linear Unit) because it helps avoid vanishing gradients. We experimented with using GeLU (Gaussian Error Linear Unit) in the convolutional layers, but found that GELU was more useful in the dense layers where final move decisions are made. We also tried using a Sigmoid, but found the ReLU was consistently better, even when we used some ReLU and some Sigmoid. In the dense layers, we incorporate Leaky ReLU to prevent neurons from "dying" by allowing small negative values, ensuring a smoother gradient flow, especially in deeper layers. We found this to have a significant improvement over ReLu, though it’s effects were not isolated since we also made this change alongside adding another dense layer. Additionally, we use GELU (Gaussian Error Linear Unit) in later dense layers, which dynamically scales activations. Finally, for the output layer, we apply Softmax, which converts raw scores into probabilities to predict the best column to be played. This was the only logical activation function for the output layer and was never changed.

Zero Padding

The default in tensorflow is that no padding is applied. We did not try this because we felt it would cause the feature maps to shrink quickly and spatial information would have been lost. Our first choice was to explicitly set padding="same" which pads the input so the output has the same shape as the input, preserving the full board structure across layers. This seemed the most intuitive and was not tinkered with.

Max Pooling

After each convolutional block, we use 2x2 Max Pooling to select the highest value in each small region and filter out the less relevant details to keep only the strongest features. We did not tinker with this much, since we felt that 3x3 would shrink the board too fast and may remove small features. This would not have been ideal for our Connect4 use case.

Batch Normalization

We used Batch Normalization after every convolutional layer to stabilize training and speed up convergence. This normalizes activations at each layer and reduces internal covariate shift, preventing large fluctuations in weight updates. This helps our network handle varying board states in Connect 4 without becoming too sensitive to specific positions. This also allows us to use higher learning rates without instability.

Dense Layers

The dense layers take all the extracted board features from the convolutional layers and learn how to translate them into the best move selection. This is where the model can actually decide what moves it should play, based on its analysis of the boards. Here is where the real decisions are made. The tweaks to the dense layers are where we saw the biggest improvements in our model results!

Regularization

We incorporated regularization here to prevent overfitting. In our first few iterations, we did not include any regularization in the dense layers and it was apparent by divergence of loss in the plots that it was necessary. We applied Ridge (L2) with a 0.001 penalty to the first dense layer, which discourages overly large weights, ensuring the model doesn't rely too heavily on specific patterns. This performed slightly better than Lasso (L1), so we stuck with it.

Additionally, we used Dropout to randomly deactivate neurons during training and make the model distribute its learning across multiple pathways. Our initial iteration used 0.1 and 0.2, but we saw some overfitting and decided to increase it. We increased it to 0.4 and 0.5 but thought that this was contributing to the problem mentioned earlier where the board was not blocking early plays. Our final model used dropout rates of 0.3 and 0.2.

Number of Dense Layers

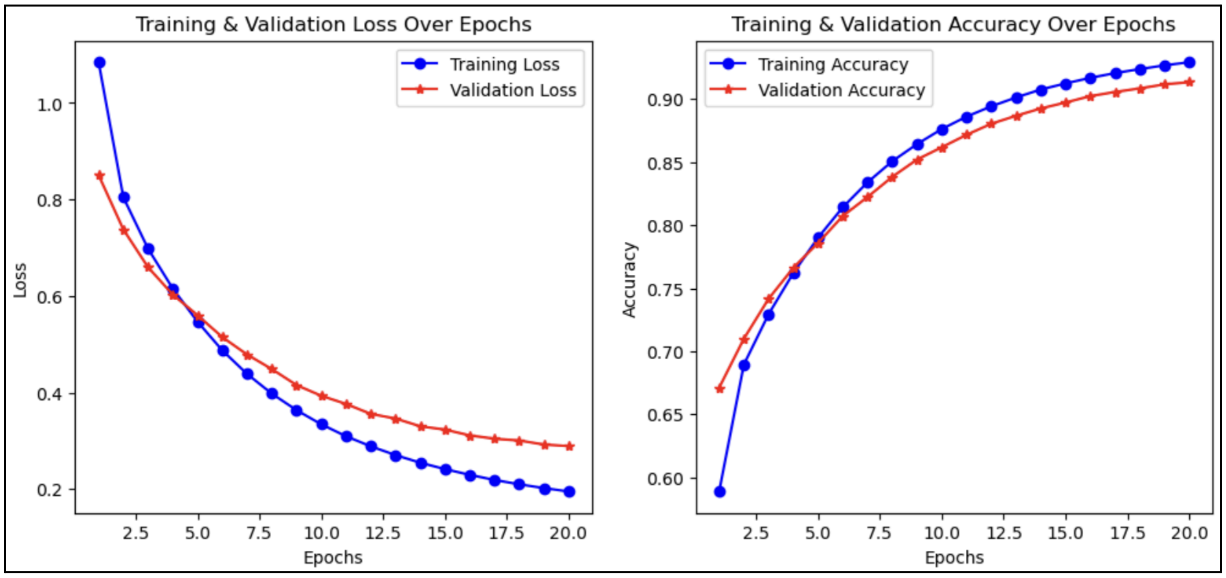
Our largest improvement in accuracy came when we added a third dense layer. This took us from a validation accuracy of 81% to a validation accuracy of 91%! We didn't isolate this change from the final change in filter sizes, and we believe that this increase in dense layers made the model more competitive against strategic opponents later in the game, but that the change in filter sizes is what allowed us to prevent those earlier wins.

Learning Rate Scheduler

We use an Exponential Decay schedule, starting with a learning rate of 0.001, which drops by 4% every 5000 steps. This allows the model to learn quickly in the early stages, making large weight updates, while later fine-tuning its strategy with smaller, more precise adjustments. This helps prevent the model from getting stuck in sharp local minima or overshooting optimal weights, leading to smoother convergence.

Monitoring Loss

To prevent overfitting and unnecessary computation, we implemented early stopping, which monitors the validation loss (val_loss) and stops training when the model stops improving. The patience parameter was initially 5, but this was too high. We changed it to 2 and felt this stopped the model too soon when there was a slight fluctuation which didn't necessarily mean it was done improving. We ultimately set the patience parameter to 3, but early stopping never stepped in and was a rather anticlimactic assistant.



We see our strongest progress early in the early 5-7 epochs, as expected. After that, we still see both the training and validation loss decrease. It is apparent that training past 20 epochs would lead to overfitting, as we see the validation loss start to level off around the later epochs. The model shows strong performance, achieving around 90% training accuracy and 88% validation accuracy, with both loss and accuracy curves following a smooth, steady improvement. While the validation accuracy slightly lags behind training accuracy, indicating mild overfitting, the gap is small. Overall, the final model is improving well.

Final CNN Specs

```python
def build_cnn():
   model = models.Sequential([
       # conv1
       layers.Conv2D(64, (5, 5), activation='relu', input_shape=(6, 7, 2), padding="same"), #increased from 4,4
       layers.BatchNormalization(),
       layers.Conv2D(64, (5, 5), activation='relu', padding="same"), #increased from 4,4
       layers.BatchNormalization(),
       layers.MaxPooling2D((2, 2)),

       # conv2
       layers.Conv2D(128, (4, 4), activation='relu', padding="same"), #changed from gelu
```

```
        layers.BatchNormalization(),
        layers.Conv2D(128, (4, 4), activation='relu', padding="same"), #changed from gelu
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),

        # conv3
        layers.Conv2D(128, (4, 4), activation='relu', padding="same"),  #changed from sigmoid and 3,3
        layers.BatchNormalization(),
        layers.Conv2D(128, (3, 3), activation='relu', padding="same"),
        layers.BatchNormalization(),
        layers.Conv2D(128, (3, 3), activation='relu', padding="same"),
        layers.BatchNormalization(),

        # flatten & dense layers
        layers.Flatten(),
        layers.Dense(256, activation=tf.keras.layers.LeakyReLU(alpha=0.01),
kernel_regularizer=regularizers.l2(0.001)),
        layers.Dropout(0.3),  # reduced from 0.5 to 0.3 to prevent overfit
        layers.Dense(256, activation=tf.keras.activations.gelu),
        layers.Dropout(0.3),
        layers.Dense(128, activation=tf.keras.activations.gelu),
        layers.Dropout(0.2),
        layers.Dense(7, activation='softmax'),  # Output layer

    ])

    # lr scheduler
    lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.001,  #same is initial learning rate
    decay_steps=5000,            #number of steps before applying decay
    decay_rate=0.96,              #4% decrease at every decay step
    staircase=True                 # change from continous (=false)
    )


    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lr_schedule),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

    return model
```

Train Transformer

The transformer is designed to predict the best move in a Connect 4 game against an opponent. We are using a ViT architecture to handle board state representations more effectively.

In Connect 4, the game is played on a 6x7 board in which players can determine where to put their color piece and make their move. For the transformer, we utilized a (6, 7, 2) array where each cell held information about whether it was occupied by player 1, 2 or if it was an empty slot. Before we processed the board, it was flattened into a (42, 2) shape to conform to the Transformer input format specified.

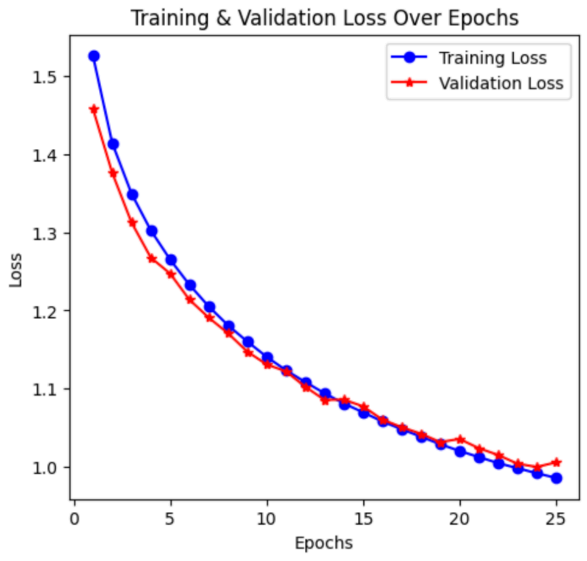
In terms of **Transformer Architecture**, there were a variety of layers that went into building this. The Transformer consists of 8 transformer encoder layers made of up multi-headed self-attention and a feedforward network. The multi-headed self-attention allowed the model to weigh the importance of different board positions relative to one another. This helped determine which positions were essential to winning and which were not. The feedforward layers with GELU activation helped improve the model's expressiveness by allowing smoother gradients and better feature extraction. Dropout for regularization was at 0.1 and helped prevent overfitting by randomly deactivating neurons during training. In turn, this helped with generalization.

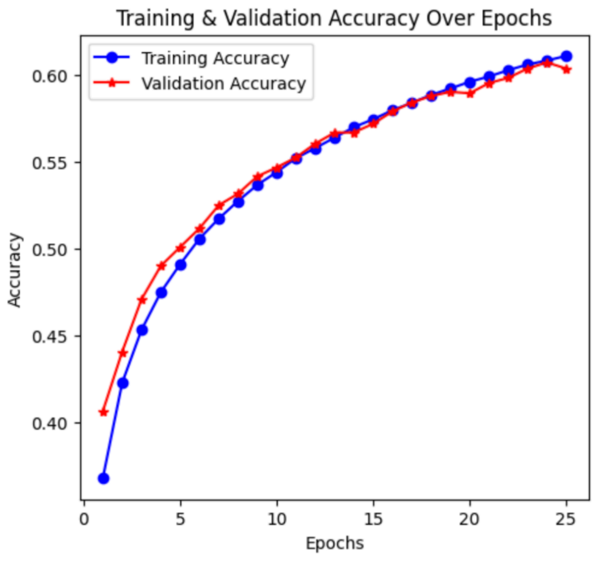
We also defined a positional index for embeddings, which was used to provide location information to the board representation. This helped the model retain spatial structure, letting it differentiate between board positions in spite of the input flattening. A class token was also utilized to aggregate information from all the board positions. The class token's output was ultimately used to make the final move selection for the transformer during prediction.

The training setup for the transformer was with an Adam optimizer with a fixed learning rate of 0.001. This was not set too high to avoid overreaching with an aggressive learning rate. Early stopping was also implemented as a call-back to prevent excessive training if validation loss did not improve.

We reached an initial level of accuracy of around ~58% and decided to make some tweaks to the model in an attempt to improve accuracy. We implemented a Mixup function to blend two board states and increase generalization while decreasing overfitting. We wanted to give the model an opportunity to look at different combinations of boards, however, this ended up decreasing accuracy. After further investigation, it is possible that this mixup confused the model since the board states were otherwise discrete. The main architectural change made was changing the hidden_dim values from 32 to 64. This increased the model parameters from around 39,000 to close to 140,000. With this change, the model accuracy went up to around 61% accuracy. Determining epoch and batch size were also factors that affected and improved the accuracy. Switching from 20 epochs to 50 epochs initially helped the model's accuracy but slowly reached a plateau. I found that scaling back the epochs from 50 to 25 allowed the model to learn long-term dependencies without overfitting and switching from a batch size of 64 to 128 helped stabilize training by reducing variance in weight updates.

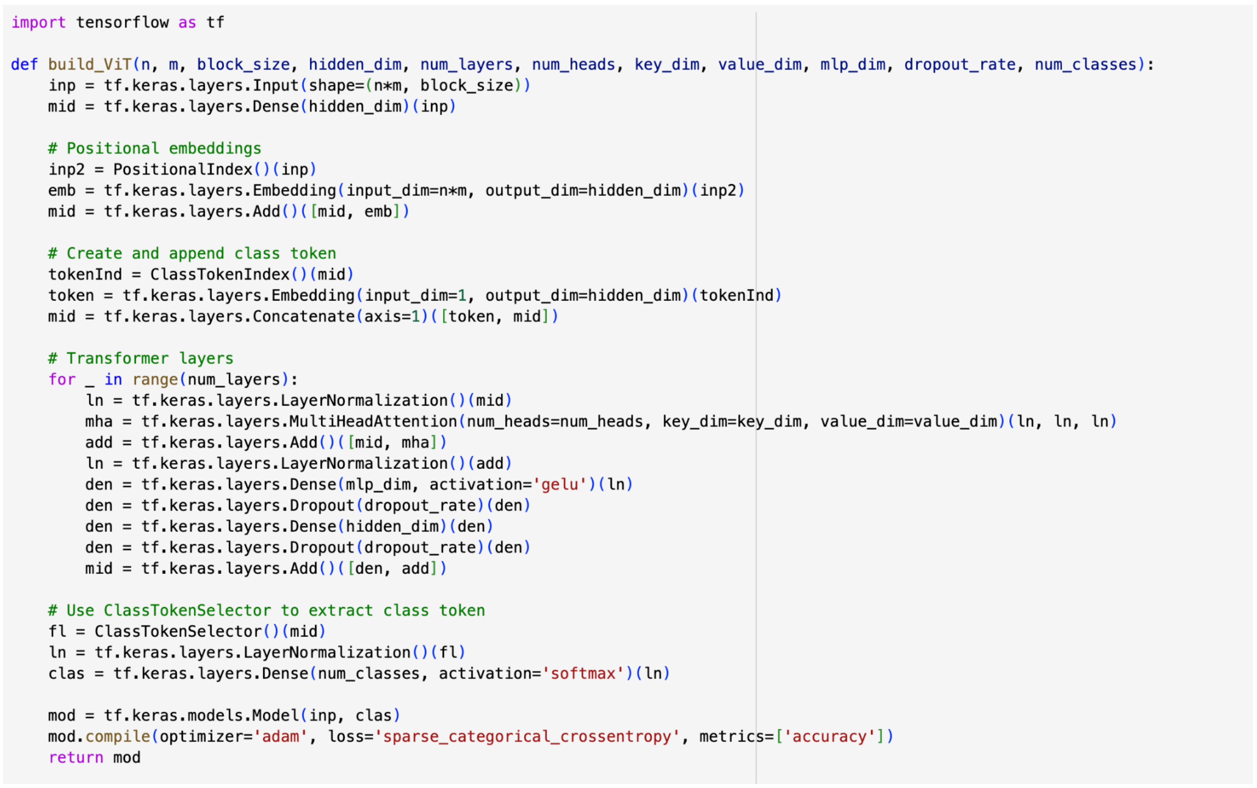
Monitoring Loss





We observed heavy loss minimization over the first 5 epochs and we also saw a jump in accuracy over the first 5 epochs. From there on, the accuracy increase slowed for both training and validation. A notable point at epoch 20 is when validation training accuracy surpassed validation accuracy, likely due to overfitting. We implemented early stopping to prevent this from continuing as the model was trained, however, we didn't cut off training immediately. The patience parameter was set to 5 epochs so that the model continued for another 5 epochs before terminating. This fluctuation happened towards the end of training so the model was able to complete its 25 epochs.

Transformer Specs



```python
import tensorflow as tf

def build_ViT(n, m, block_size, hidden_dim, num_layers, num_heads, key_dim, value_dim, mlp_dim, dropout_rate, num_classes):
    inp = tf.keras.layers.Input(shape=(n*m, block_size))
    mid = tf.keras.layers.Dense(hidden_dim)(inp)

    # Positional embeddings
    inp2 = PositionalIndex()(inp)
    emb = tf.keras.layers.Embedding(input_dim=n*m, output_dim=hidden_dim)(inp2)
    mid = tf.keras.layers.Add()([mid, emb])

    # Create and append class token
    tokenInd = ClassTokenIndex()(mid)
    token = tf.keras.layers.Embedding(input_dim=1, output_dim=hidden_dim)(tokenInd)
    mid = tf.keras.layers.Concatenate(axis=1)([token, mid])

    # Transformer layers
    for _ in range(num_layers):
        ln = tf.keras.layers.LayerNormalization()(mid)
        mha = tf.keras.layers.MultiHeadAttention(num_heads=num_heads, key_dim=key_dim, value_dim=value_dim)(ln, ln, ln)
        add = tf.keras.layers.Add()([mid, mha])
        ln = tf.keras.layers.LayerNormalization()(add)
        den = tf.keras.layers.Dense(mlp_dim, activation='gelu')(ln)
        den = tf.keras.layers.Dropout(dropout_rate)(den)
        den = tf.keras.layers.Dense(hidden_dim)(den)
        den = tf.keras.layers.Dropout(dropout_rate)(den)
        mid = tf.keras.layers.Add()([den, add])

    # Use ClassTokenSelector to extract class token
    fl = ClassTokenSelector()(mid)
    ln = tf.keras.layers.LayerNormalization()(fl)
    clas = tf.keras.layers.Dense(num_classes, activation='softmax')(ln)

    mod = tf.keras.models.Model(inp, clas)
    mod.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return mod
```

This image depicts the model architecture, which incorporates the positional embeddings, a class token, and the multi-head self-attention layers mentioned previously. The Transformer encoders consist of layer normalization, self-attention, and feedforward layers with GELU activation to get meaningful patterns from the game board. The final move is selected using a class token which aggregates information from all board positions and is passed through a softmax layer for classification. The model is compiled finally with the Adam optimizer to improve predictions.

The changes made to the original model provided the best balance in terms of accuracy, training time, and stability. Moving forward, with more time, it would be interesting to explore reinforcement learning approaches or even the implementation of a hybrid CNN-Transformer model to improve move prediction in Connect 4.

Board States

Flattening of Board State and Spatial Disruption

One major reason the Transformer model may have failed is the flattening of the board state before feeding it into the model. In the preprocessing step, the board is reshaped from (6,7,2) to (42,2), which removes the critical spatial structure of the game. Connect 4 relies heavily on positional relationships, such as vertical stacks, horizontal rows, and diagonal sequences. By flattening the board into a simple sequence of numbers, the model loses its ability to recognize these formations effectively. Unlike natural language processing tasks—where Transformers excel at sequential dependencies—Connect 4 requires an understanding of grid-based relationships, making CNNs or hybrid CNN-Transformer architectures a better fit. This issue likely caused the model to miss the diagonal win condition in this board state, leading to incorrect predictions.

Improved Learning Through Additional Training

In this board state, the Transformer AI secured victory with a diagonal sequence of four Xs, starting from column 6 at the bottom and moving backward toward column 3. This is a significant improvement over previous failures, where the model struggled to recognize diagonal win conditions. We continued to train the model, allowing it to develop a stronger understanding of spatial dependencies beyond simple horizontal and vertical patterns. Additional training exposed the model to more diverse board states, enabling it to recognize diagonal formations more reliably. This indicates that either data augmentation (including more examples of diagonal wins) or improved model tuning (such as refined attention mechanisms or better reward shaping) has helped enhance its ability to plan multi-move strategies. After all, diagonal wins are essential in Connect 4!