5T: fine-Tuned Tic-Tac-Toe Thinker

Joshua Hahn, Jimmy Zhang

December 8th, 2023

1 Abstract

Computer vision has often taken a backseat role in games like tic-tac-toe, in which it is used primarily for object detection, whereas a separate minimax algorithm performs the actual evaluation of a given game state. In this study, we implemented three different deep learning models, including a convolutional neural network (CNN) built from scratch and two state-of-the-art pre-trained models (ResNet and EfficientNet) that we transferred to our problem using network surgery. We generated all of our data from scratch, first generating all 5,478 valid tic-tac-toe states and then converting each of those states into a 28x28 image of a tic-tac-toe board, augmented with random shifts based on a distribution and heatmap, as well as random sampling from a set of pre-drawn X's, O's, and lines. We also generated all of the labels using a custom-made scoring algorithm. Our goal was to solve the following three-class classification problem: to predict whether X is winning, O is winning, or the position is tied in any given tic-tac-toe state. After training each of our models for 100 epochs, using a validation set to prevent overfitting, our CNN model from scratch achieved an accuracy of 93.87% on the test set; ResNet achieved an accuracy of 96.94%; and EfficientNet achieved an accuracy of 81.97%. The strong predictive performance of our models gives us some assurance that deep learning models can, indeed, be used to evaluate tic-tac-toe boards, even in the absence of a minimax algorithm. Future research should consider experimenting with larger board sizes, introducing even more random noise, and inducing a turn mechanic (with which our models struggled most).

2 Introduction

With the advent of convolutional neural networks and, more recently, vision transformer architectures, computer vision technologies have grown increasingly more powerful and robust over the years. As a result, games ranging from board games like chess to MOBA video games like Dota to augmented reality games like Pokémon GO have become some of the major targets of new computer vision systems. However, one game that has not received quite as much attention amidst the computer vision boom is a childhood classic: tic-tac-toe.

As a relatively simple game that has long ago been ”solved” mathematically by other AI algorithms, tic-tac-toe has often been overshadowed in the computer vision world by more challenging games with more complicated rules to learn and more complex features to extract. Nonetheless, the one main application of computer vision to the game of tic-tac-toe has been to use image processing techniques to detect the position of X's and O's on the board, which are then used to produce the current game state.[1] For this reason, computer vision models are generally not used as stand-alone interfaces to play a game (even a simple one like tic-tac-toe). Instead, once the current game state is generated by a computer vision model, it is then handed off to a separate minimax algorithm[2], which is the actual mastermind behind *playing* the game. In this case, the minimax algorithm takes care of evaluating the current game state and actually generating a next move. Meanwhile, the computer vision aspect of these gameplay pipelines is relegated to the role of picking out individual components of a board, without a true understanding of the strategy and rules of tic-tac-toe.

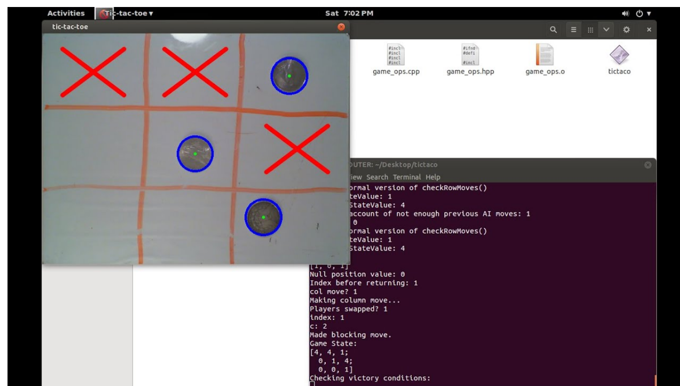

Figure 1: A use case of OpenCV for detecting X's and O's in a tic-tac-toe board.

Though this combination of computer vision (for object detection) and minimax algorithms (for generating optimal computer moves) creates an interactive PvE (player vs. environment) simulation in which a human is matched up against an AI opponent, as in the example using OpenCV above[3], we wanted to know whether computer vision can be used to actually evaluate an existing game state (similar to what minimax does, minus the generation of a next move). Specifically, we wanted to know whether a neural network can take a tic-tac-toe board (in any valid but arbitrary state) as input and determine whether

[1] `https://arxiv.org/abs/2205.03663`

[2] `https://www.pnas.org/doi/abs/10.1073/pnas.39.1.42`

[3] `https://www.youtube.com/watch?app=desktop&v=O1l3xKv4VLU`

X is winning, O is winning, or neither player is winning (i.e., the position is drawn/tied).

Initially, we will experiment with 3×3 boards that are already in a terminal state, in which either one player has already one by achieving 3 in a row or the game is drawn with all squares filled but neither player achieving 3 in a row. We anticipated that this preliminary problem would not be too difficult, as any sufficiently deep model should be able to determine that 3 of the same symbol (either X or O) in successive horizontal, vertical, or diagonal tiles indicates a win for that symbol. However, to make the problem more difficult, we want to then see if the models can *learn* the rules of tic-tac-toe and understand strategies that result in an advantage for one player or the other. Accordingly, we will then experiment with boards that are not yet in a terminal state but will inevitably lead to either a win for X or a win for Y (i.e., one player has the positional advantage), or that will lead to a draw (i.e., neither player has the advantage). Moreover, we hope that our models can also extract information about whose move it is given an input board (i.e., by learning that X plays first and determining whose move it is by counting how many X's and O's there are on the board).

3 Project Outline

1. **Data Preparation/Generation**: manually generating the tic-tac-toe boards and labeling them according to which player is currently winning
2. **Model Design and Architecture**: (1) implementing a CNN from scratch and (2) using two different pre-trained models (ResNet and EfficientNet) with fine-tuning
3. **Training**: training the models and evaluating their per-epoch validation loss and accuracy
4. **Evaluation**: evaluating the accuracy of models on the test data
5. **Insight Extraction**: analyzing test examples on which the models struggled and drawing insights about the models' "game knowledge"

4 Methods

4.1 Data Generation

The data generation step for this project entailed a three-step process:

1. Generating valid game states in an intermediate representation
2. Applying a scoring algorithm to associate each game state with a label
3. Turning the intermediate representation into a 28x28 image

Following this process, we were able to generate a total of 5,478 valid game states (i.e., respecting turn mechanics and terminal conditions) and then generate 3 images for each game state (all slightly different from each other) for a total of 16,434 images. In the following sections, we will explain in detail the motivations and mechanisms of the data generation process.

During the data generation process, we placed heavy emphasis on creating variety and randomness in our experiment. Because we were working on models that could be capable of deterministically learning the search space if we had no randomness, we wanted to make the task more challenging and interesting by creating small perturbations in the position of the tiles and the shape of the Xs and Os, thus creating multiple representations of any single game state. We felt that this would be a better representation of real-life tic-tac-toe, where players agree to draw on a 3x3 grid, but their tiles may be placed off-center or drawn differently each time depending on various uncontrollable factors (e.g., handwriting, writing instrument, etc.).

Valid State Generation

In total, there are 5,478 valid tic-tac-toe boards[4]. For this computer vision task, we considered any two game boards to be in unique states if their tile values were not completely identical, even if they were the same board when transformed with rotations. We thought that this was an appropriate decision to make, given that our vision model would be learning local patterns specific to an area of the board, as opposed to being able to understand that two boards are congruent. To generate the set of all valid tic-tac-toe states, we wrote a depth-first search (DFS) algorithm that would start with an empty board, and then simulate players adding X tiles and O tiles one by one, until a game ends in a terminal state (i.e. one player achieves 3 in a row or there are no valid moves remaining). Throughout this process, we keep track of every unique board that we have seen, giving us the final set of intermediate board states. Below are some examples of intermediate board states. We use a 3x3 array to represent a given state, where 0 correspond to an empty tile, 1 corresponds to an O tile, and 2 corresponds to an X tile.

0	0	1
0	1	2
2	0	2

0	2	0
1	2	0
0	2	1

1	1	1
0	2	0
2	0	2

0	2	0
1	0	2
0	1	0

Scoring Algorithm

After generating the intermediate game states, we labeled each board with a score corresponding to who is currently in a winning/advantageous position. 0 represents a tied game, 1 represents a winning (or won) state for O, and 2

[4] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2021-2022/Makalah2021/Makalah-Matdis-2021\%20(148).pdf

represents a winning (or won) state for X. To quantify the game's status, we used the following rules and heuristics:

- If a game is won or tied, we immediately return the label corresponding to the outcome.
- If a player has a game-winning move and it is their turn, we return the label corresponding to the player's turn.
- For all games that are not completely won or tied, we use a scoring mechanism as follows:
  - We add 5 points to the player whose turn it currently is.
  - For every line or diagonal with two of the same tile and an empty tile, we add 10 points to that player.
  - For every line or diagonal with two empty tiles and one of that player's tile, we add 5 points to that player.
  - If the difference in points is ≤ 5, then we label the game as being tied. Otherwise, we label the board 1 or 2 corresponding to which player is winning.

There are many other scoring heuristics and weights that we could have chosen to score our boards. However, after running the algorithm and inspecting individual board states and their labels, we fine-tuned the weights until our interpretation of the game state lined up with the scoring algorithm's interpretation of who was winning.

Image Generation

Once we generated our board states and labels, we converted each board state into an image that our models could take in as input. We built the image using the intermediate state representation array by first initializing an empty 28x28 numpy array, adding the horizontal and vertical lines from a predrawn set of lines, and then finally adding in the tiles. Because this was a learning task in which we were also the ones creating the dataset, we were cognizant of the possibility that we would be making the learning task trivial by not introducing enough variance, noise, or complexity. Thus, to encourage our model to generalize its understanding of the sample space, we used the following methods.

The first method was to sample tiles from a pool of different images. To do this, we manually drew a set of Xs, Os, horizontal lines, and vertical lines that our algorithm could randomly sample to use in the image.

By including several versions of the tiles that we can choose from, we ensure that even if we run the same image data generation algorithm on the same intermediate representation, the images can turn out very different. More importantly, this encourages the model to learn the structure of the Xs and the

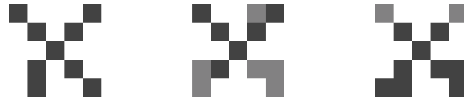

Figure 2: Sample X tiles

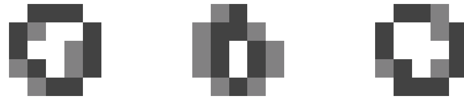

Figure 3: Sample O tiles

Os, rather than hard-coding the image of the X and O to determine the score of the board.

Another mechanism that we used to encourage the model to not overfit to our generated images was to randomly shift the tiles off of their center based on a distribution and a heatmap.

The heatmap for the image generation process is a 28x28 image in which the different colors correspond to the likelihood that a tile or line will be placed there. After experimenting with different weights for each of the colors, we arrived at the following distributions, in order of increasing darkness:

- Tiles (blue): [0, 0.01, 0.14, 0.35, 0.35, 0.14, 0.01, 0]
- Lines (red): [0, 0.1, 0.4, 0.4, 0.1, 0]

We have included an example of an intermediate board and the corresponding image that was generated from it on the following page.

The final mechanism that we used to increase variability in our boards was to create 3 images (with varying amounts of noise and sampling of X/O tiles) for every intermediate board representation we had. This ensured that (1) the data we used to train our model had more information on which to base its decisions and (2) that we would have sufficient points in the sample space.

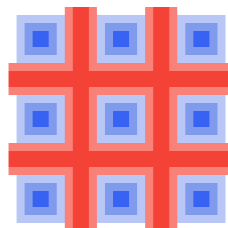

Figure 4: Heatmap used for image generation

0	2	1
1	1	2
2	2	0

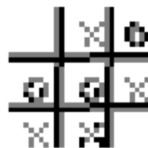

Figure 5: An intermediate board state and its corresponding image

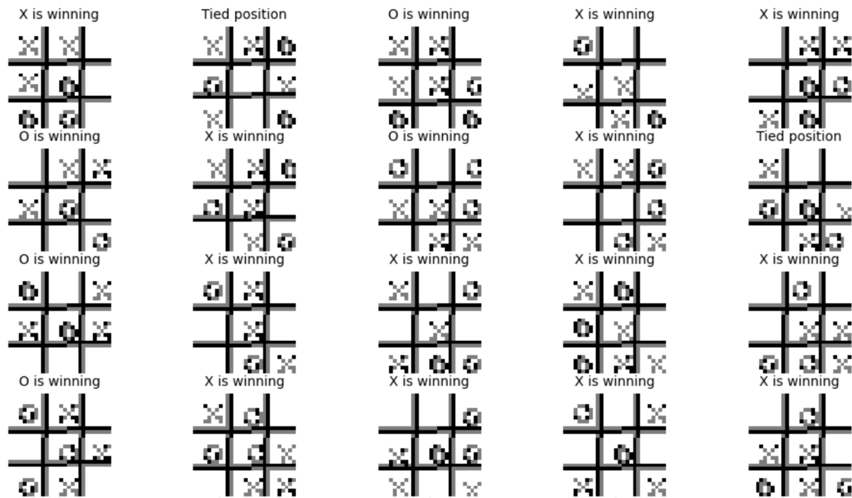

Figure 6: A sample of 20 different boards and their corresponding labels

Combining all of these mechanisms, we were able to generate a dataset of 16434 images, of which 13147 were used for training, 1643 were used for testing, and 1644 were used for validation. Above is a small sample of the data that we generated.

Varying the Data Generation Heatmap

During the initial training phases of our model, we found that accuracies were often much poorer than we expected. For instance, in our CNN model from scratch, we saw an accuracy of 59%, which certainly is not horrible for a 3-class classification problem, but we wanted to make sure that our model was learning much more. However, we soon realized that the search space of the data was much larger than we had realized. To illustrate this point, we first considered the high-level features that we hoped to identify in the tiles: Xs, Os, and the grid. However, each one of these features are composed of many smaller features, which take up many parameters to identify. Once we began introducing variation in position, it became very hard for our vision model to pick up on all of the parameters, regardless of how many iterations we had, simply because there were just too many parameters to learn in the search space.

Thus, we modified the heatmap probability distribution, as well as the number of possible Xs and Os from which we were randomly drawing, until we found a sweet spot where our models were able to robustly represent the search space with their existing architectures.

4.2 Model Selection and Architecture

To build our models, PyTorch[5] was our deep learning framework of choice. We experimented with three different models: a CNN that we built from scratch and two pre-trained models (ResNet50[6] and EfficientNetV2[7]) to which we applied domain transfer with network surgery.

CNN from Scratch

For our custom-made CNN, we created a LeNet[8]-style model that included 3 convolutional layers, 2 fully connected hidden layers, 2 max-pooling layers, 2 dropout layers, 1 linear output layer, and 1 softmax layer. Since our input data was grayscale, the first convolutional layer had 1 input channel and 32 output channels. The second convolutional layer had 32 input channels and 64 output channels, and the last convolutional layer had 64 input channels and 128 output channels. We opted to use a kernel size of 3 for the first and third convolutional layers and a kernel size of 5 for the second convolutional layer so that we could extract more features in the middle layer without having to pass too many features to the hidden layers. All of the hidden layers had a stride of 1 and same padding, except the second convolutional layer, which had a padding of 1. ReLU was applied after every convolutional and hidden layer; max pooling with a kernel size of 2 and a stride of 2 was applied after the second and third convolutional layers; flattening was applied after the last convolutional layer; and dropout with 0.3 probability was applied after each fully connected linear layer. The linear layers had 1024 and 512 neurons, respectively. For weight initialization, we used Kaiming initialization[9]. Below is the full network architecture of our custom-made CNN.

Domain Transfer with Network Surgery

We also experimented with 2 state-of-the-art pre-trained models: ResNet50 and EfficientNetV2. To perform domain transfer, we imported the pre-trained weights of these models from the PyTorch library and replaced each of their heads with a new head consisting of a fully connected layer followed by a softmax layer.

---

[5] `https://pytorch.org/`
[6] `https://arxiv.org/abs/1512.03385`
[7] `https://arxiv.org/abs/1905.11946`
[8] `http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf`
[9] `https://arxiv.org/abs/1502.01852`

```
CNN(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=same)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=same)
  (fc1): Linear(in_features=4608, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=512, bias=True)
  (output): Linear(in_features=512, out_features=3, bias=True)
  (relu): ReLU()
  (pool): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1, ceil_mode=False)
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (dropout): Dropout(p=0.3, inplace=False)
  (softmax): Softmax(dim=1)
)
```

Figure 7: The architecture of our CNN from scratch

4.3 Training

From the dataset of 16434 generated images, we stratified the image-label pairs into a training set, a test set, and a validation set, using an 80/10/10 train/test/validation split. We made sure that we had enough data points to train our model on and learn the parameters, but also report a testing accuracy that used sufficient data points to be accurate. Having a large validation set also helped us prevent overfitting, since our training process took advantage of early-stopping using the validation accuracy and loss metrics (specifically, we stopped training if validation loss increased or stayed the same in 5 consecutive epochs). Moreover, this allowed us to set our upper bound on the number of epochs to 100, encouraging the models to continue learning the parameters of the search space until they reached a point where they were just beginning to overfit. Having a large validation set allowed us to accurately measure when this inflection point occurred, and our model almost never reached the 100th epoch. For the pre-trained models, we initially first froze all of the pre-trained weights, allowing only the weights of the final output layer to be updated. Afterwards, we unfroze the weights of all layers and then continued to fine-tune the models on the training set for 100 more epochs.

To train our models, we used a learning rate of 1e-4 (for the pre-trained models, we used a learning rate of 1e-3 in the first round of training and 1e-4 in the second round) and a batch size of 32. We used cross-entropy loss as our loss function, with weights set to be inversely equal to the proportion of each class in the training set in order to combat class imbalance, and we used Adam as our optimizer. The training was done on an NVIDIA T4 GPU hosted on Google Cloud, on a collaborative Google Colab file.

5 Results

5.1 Model Performance

To illustrate the degrees of success of our models, we have included graphs of the validation loss as a function of the number of training epochs, which will demonstrate how each of the models learns over time.

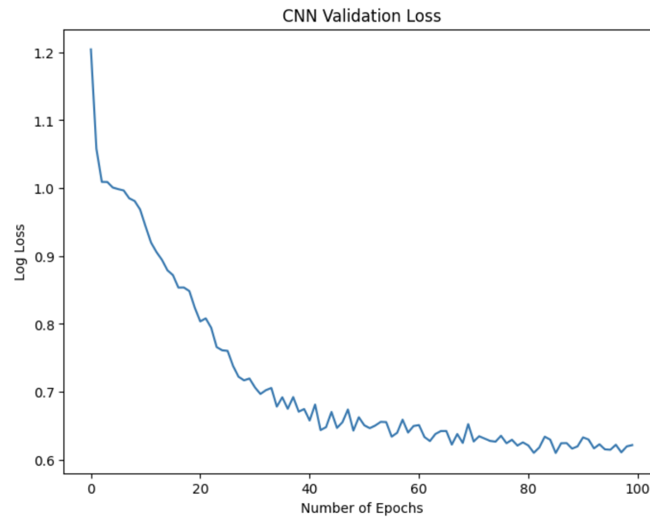

Figure 8: Validation loss plot for the CNN from scratch

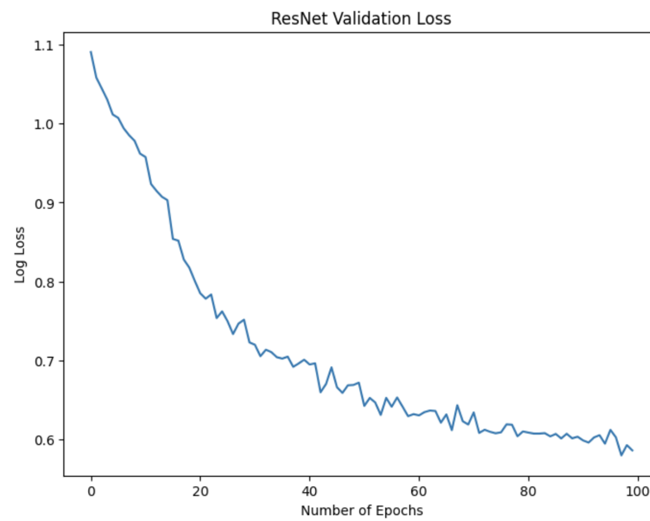

Figure 9: Validation loss plot for the ResNet model

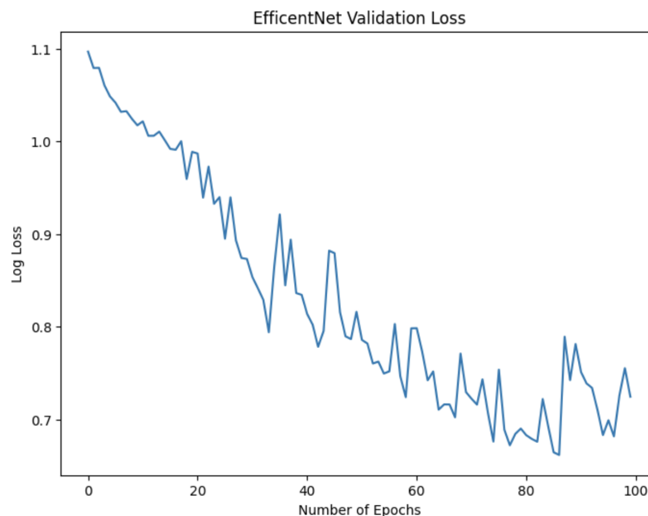

Figure 10: Validation loss plot for the EfficientNet model

After we finished training each of the three models, we evaluated the performance of each of the models on the held-out test set. In the end, all of our models performed very well in terms of test accuracy, with ResNet achieving the highest test accuracy, CNN from scratch second, and EfficientNet third. The exact test accuracies are as follows:

- CNN from Scratch: 93.87%
- ResNet: 96.94%
- EfficientNet: 81.97%

5.2 Sample Correct Boards and Mistakes

Thinking about future work, we also investigated some sample cases where the models incorrectly predicted the label of the outcome (shown on the following page).

We see that sometimes, the model is unable to understand the turn mechanic of tic-tac-toe. For instance, the model sometimes predicts that X is about to win, even though it is O's turn and O has an immediately wining move (or vice versa). On the other end of the spectrum, the models will sometimes predict that X and O are tied when the positioning of the tiles are equal, but one side actually has an advantage because it is their turn to play.

The encouraging of the models to consider the turn-based nature of the game was enforced by adding heuristic points to whichever player's turn it was, but it seems that the models were unable to completely catch the nuance of the turns.

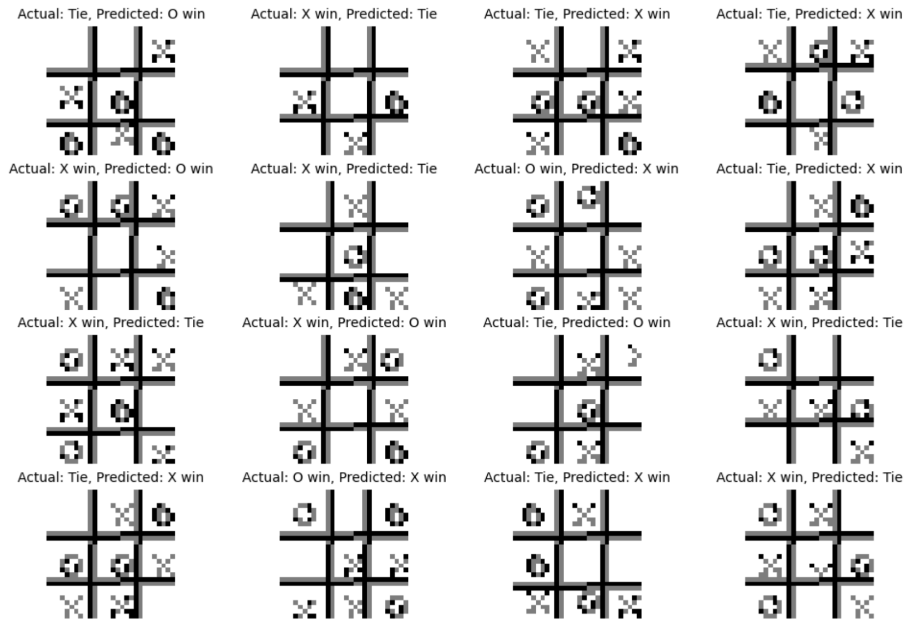

Figure 11: Sample mistakes that the CNN from scratch made

For future work, engineering the scoring and labeling algorithm's heuristics to reflect the game's mechanics may increase accuracy in more varied or noisy data.

6 Discussion

6.1 Conclusions

Based on the accuracies of our models, we felt that we were able to create models that were able to capture the sample space well. All three of our models achieved an accuracy of 80%, and our CNN from scratch and ResNet model achieved accuracies of well over 90%. However, the first version of our CNN, which was a LeNet-style model with 3 convolutional layers and 3 fully-connected layers yielded an accuracy of about 59%. Moreover, our initial trials of the CNN model also yielded much lower accuracies. As mentioned in the **Varying the Data Generation Heatmap** section, we varied the difficulty and variation of our model to find a place where the variation encouraged the model to learn the nuances and not overfit, but was not too difficult that the model could not learn the patterns well enough in a reasonable time. We felt that the final dataset and heuristics we created were appropriate, and our models' test accuracies seemed to reflect this as well. Overall, the strong predictive performance of our models gives us some assurance that deep learning models can, indeed, be used to evaluate tic-tac-toe boards, even in the absence of a minimax algorithm.

6.2 Limitations and Suggestions

Given that our models were trained only on 3x3 boards, it may be difficult to generalize the game to larger board sizes, even though 3x3 is the standard board size for tic-tac-toe. Moreover, we used an image size of 28x28 to decrease the amount of data that our training set was taking up and also to encourage faster training times. Perhaps successive iterations of this project can take advantage of more powerful machines that can take larger boards (or more complex boards with more noise) as input.

In addition, we also experimented with a vision transformer using torchvision's Vit_B_32, but we found that the training time required for the vision transformer model was excessively long (even on an NVIDIA T4 GPU) due to the sheer size of the model architecture; the vision transformer model also struggled to converge, which we attribute to both the time required to train the model and the model complexity, which may even be too intricate for our problem. This phenomenon may also explain why EfficientNet, which we expected to achieve the highest accuracy on the test set, did not perform as well as we anticipated. It may simply be the case that more complex model architectures are not as well suited for this problem, though future research can further investigate this hypothesis.

7 Contributions

JZ implemented the algorithm to generate the tic-tac-toe states, and JH implemented the algorithm to label/score individual states and generate a 28×28 image for each board. JZ and JH both researched and implemented different neural network architectures and pre-trained models for the task of evaluating tic-tac-toe game states. JZ and JH both built the PyTorch models and evaluated their performance on the tic-tac-toe boards. JZ and JH both wrote, edited, and approved the final project report.

8 Data and Code Availability

The project source code, as well as the generated data, is available at `https://github.com/joshuahahn/fineTunedTicTacToeThinker/`.

9 References

1. https://arxiv.org/abs/2205.03663
2. https://www.pnas.org/doi/abs/10.1073/pnas.39.1.42
3. https://www.youtube.com/watch?app=desktop&v=Oll3xKv4VLU
4. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2021-2022/Makalah2021/Makalah-Matdis-2021\%20(148).pdf

5. https://pytorch.org/
6. https://arxiv.org/abs/1512.03385
7. https://arxiv.org/abs/1905.11946
8. http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf
9. https://arxiv.org/abs/1502.01852