Final Report


SIMON FRASER
UNIVERSITY

# Imitation Learning to Solve N-puzzle Problem

Using A* Search as an Expert Learner

CMPT417 Intelligent Systems

Fall 2022 Prof. Hang Ma

Alan Thomas, Neil Mukesh Shah, Tze Yan Cindy Ng

Dec 19, 2022

# Abstract

Imitation learning is a subfield of machine learning in which an agent learns to perform a task by observing and mimicking the actions of an expert. One application of imitation learning is in the domain of puzzle-solving, where an agent can learn to solve an 8 puzzle (a sliding puzzle consisting of a 3x3 grid with 8 numbered tiles and a blank space) by observing the actions of a human expert or an algorithm that is capable of solving the puzzle.

In this work, we propose a solution to the 8 puzzle problem using imitation learning and a* search as the expert learner. A* search is a popular algorithm for finding the shortest path between two points, and it can be used to solve the 8 puzzle problem by treating each state of the puzzle as a node in a graph and finding the shortest path from the initial state to the goal state.

To implement our solution, we first trained a deep neural network to predict the next best action to take at each step of the puzzle-solving process, using a dataset of expert demonstrations. We then used this trained model as an imitation learner to solve the 8 puzzle problem, using a* search as the expert.

We evaluated the performance of our solution using a set of test puzzles and found that it was able to solve the 8 puzzle problem with a high level of accuracy and efficiency. Our results demonstrate the effectiveness of using imitation learning and a* search for solving the 8 puzzle problem and suggest that this approach could be applied to other puzzle-solving tasks.

# Contents

# 1. Introduction

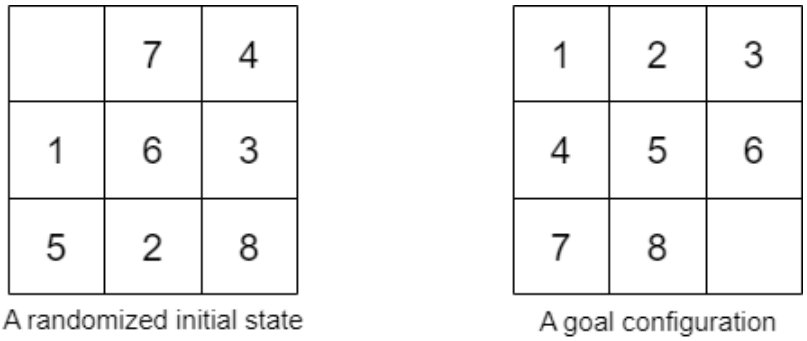



Figure 1: A 8-puzzle in a randomised initial state and its goal configuration

The 8-puzzle is a sliding tile puzzle that consists of 8 numbered tiles, where the tiles are numbered from 1 to 8, and one empty space where the tiles can be moved within a $\sqrt{(8+1)}$ by $\sqrt{(8+1)}$ frame; the 8-puzzle has a 3 by 3 frame with 9 numbered tiles in it. The puzzle can be solved by moving (sliding) the tiles one by one, horizontally or vertically, into the single empty space in order to proceed from a solvable randomised initial state to the goal configuration, where the numbered tiles are in numerical order.

This is a well-known combinatorial problem where the number of different tile arrangements is 8!; with the large number of permutations, it can be (time and space) costly to solve the puzzle.

The 8-puzzle is a commonly used problem for the implementation of A* search, a single-agent heuristic search algorithm. However, A* search is memory limited and time-consuming. Later on, the implementation of Iterative deepening A* (IDA*) improves runtime and space usage [3], however, since both implementations must search all possible ways to the solution, larger puzzles will still cause capacity (time/space) issues on regular-consumer machines due to their large state spaces.

Another approach that is often used to solve a combinatorial problem is reinforcement learning; a values-based algorithm, like Q-learning, where a function is used to calculate a value for each state and action is most appropriate. Consequently, reinforcement learning and search algorithms with heuristics can be combined to solve said problem; imitation learning is what we are using to demonstrate this idea. Imitation learning trains its policies based on provided observations and actions performed by an expert (human); in our paper, we will be using our A* search results to train our imitation learning agent.

We will implement A* to solve the 8-puzzle to produce data, then we will use that data to train our different imitation learning agents to solve the 8-puzzles. We aim to find the most accurate machine learning model using the same data set.

# 2. Algorithms

## 2.1 A* Search

A* search is a form of informed best-first search, a graph and/or tree search algorithm where its node expansion is based on an evaluation function. A* uses a function $f(n)$ to estimate the cost of the cheapest solution through node $n$, the function is calculated as follows:

$$f(n) = g(n) + h(n)$$

where $g(n)$ denotes the cost from the initial node to the current node, and a heuristic function $h(n)$ denotes the cost to get from current node to the goal node. As a tree search, by having a good heuristic function that is a lower bound on the actual solution cost (admissible), finding the lowest value of $f(n)$ will guide one to the optimal solution, assuming a solution exists.

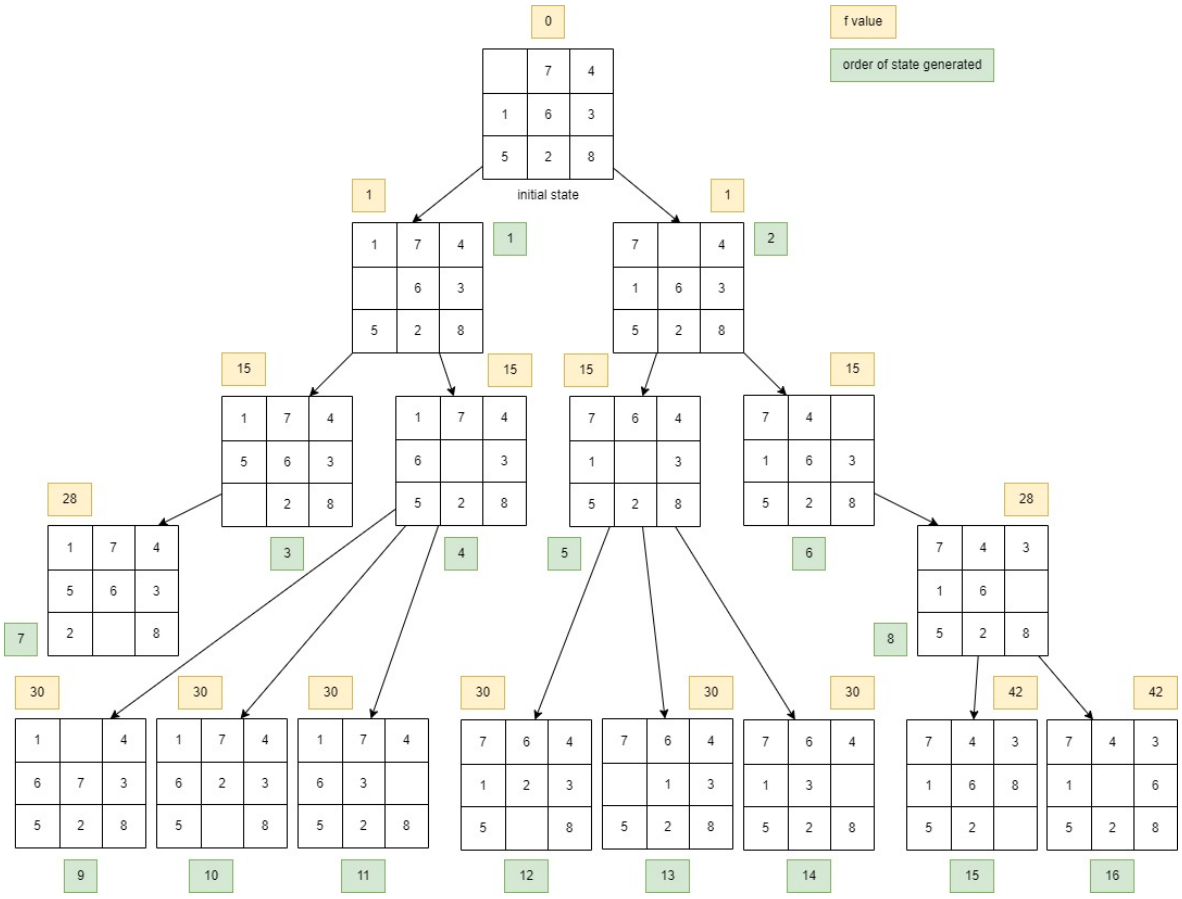



Figure 2: A* search with the first few iterations on an 8-puzzle using the Manhattan Distance heuristic

Although a heuristic function can provide guidance towards optimal paths, it must generate and consider all possible states (child nodes) as the algorithm proceeds. The child nodes are typically stored to a priority queue that sorts the nodes based on their f-values. As the search tree expands, the search space is still exponential; hence,

with A* search storing all generated nodes in memory during its run, it is memory limited; it is also time-consuming when problem size gets large [2].

## 2.2 Imitation Learning

Imitation learning (IL) is a form of reinforcement learning (RL) within machine learning. Reinforcement learning uses a series of observed rewards or penalties to learn an optimal policy for performing actions in an environment, without prior knowledge of the full environment and the rewards function. The goal of a reinforcement learning agent is to maximize the rewards it gets, however, the rewards function can affect performance of the agent if the agent is rewarded infrequently. Imitation learning does not limit itself with a rewards function to learn a policy, instead, demonstrations of observations and actions provided by an expert (human) is also used for the agent to learn the policy. It is similar to the idea of instead of trying to learn a task from the ground up, it is more efficient to learn by watching a demonstration performed by a matter expert.

With an expert providing a series of states and actions in pairs, $(s, a)$, the IL agent will imitate the behaviour or to directly learn the policy from the demonstrations (known as trajectories). The actions in the trajectories are based on the expert's optimal policy, $\pi^*$.

### 2.2.1 Behavioural Cloning

With different learning algorithms, there are various IL methods. One of the methods is behavioural cloning (BC), which is what we will be implementing. BC learns $\pi^*$ via supervised learning, which means it uses labelled data from the expert; the series of $(s, a)$ is assumed to have a probability distribution that is independent and identically distributed.

# 3. Implementation

## 3.1 N-Puzzle Generation and Solvability

```
def generate_random_8_puzzle_state():
  # Create a list with the numbers 0-8
  numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8]

  # Shuffle the list to generate a random permutation
  random.shuffle(numbers)

  state = [numbers[0:3], numbers[3:6], numbers[6:9]]

  if not is_solvable(state):
      generate_random_8_puzzle_state()

  # Return the permuted list as a 2D array (3x3 matrix)
  return state
```

Figure 3: Function to randomly generate a 8-puzzle

```python
def is_solvable(state):
    # Convert the state to a 1-dimensional array of tiles
    tiles = [tile for row in state for tile in row]

    # Compute the parity of the permutation of the tiles
    parity = sum([tiles.index(i) for i in range(1, 9)]) % 2

    # If the parity is even, the puzzle is solvable
    return parity == 0
```

Figure 4: Function to check if an 8-puzzle is solvable

It is important to note that half of the starting positions for a N-puzzle are unsolvable due to parity of the permutation, only even permutations are solvable[1].

Although it is possible to implement an A* search to return without a solution (i.e. the puzzle is not solvable), we wanted to use successful search processes as data for the our imitation learning implementation; hence, we filtered out unsolvable puzzles by implementing a function to check whether an 8-puzzle is solvable or not.

## 3.2 A* Search Implementation

We implemented our A* search to store the process of the search, along with the solution, to export as training and testing data for our imitation learning implementation later on. See Figure 5 for our A* search function. Note that ”nodes” and ”states” are both referred to a state of a 8-puzzle in a 2D-array form; in our early stage of coding, we came to the conclusion that storing each state as a 2D-array (as supposed to an object with a custom class) will ease the data export process.

```python
def a_star_search(start, goal, heuristic):
  # create an empty set to store visited nodes
  visited = set()

  # create a priority queue to store nodes that have been visited but not yet expanded
  # the priority queue will be sorted by the total cost of the path to the node,
  # which is the sum of the cost to reach the node from the start and the heuristic
  # estimate of the cost to reach the goal from the node
  queue = PriorityQueue()

  # add the start node to the queue, with a cost of 0
  queue.put((0, start))

  # create a dictionary to store the previous node for each node that is visited
  # this will be used to reconstruct the path from the start to the goal
  previous = {str(start): start}

  # create a dictionary to store the cost of the path from the start to each node
  # this will be used to determine the total cost of the path to each node
  cost_from_start = {str(start): 0}

  data, solution, actions = [], [], []

  # while the queue is not empty, continue searching for the goal
  while not queue.empty():
    # get the node with the lowest total cost from the queue
    current_cost, current_node = queue.get()

    # if the current node is the goal, we are done
    solution.append((previous[str(current_node)], current_node))
    if current_node == goal:
      break

    # if the current node has already been visited, skip it
    if str(current_node) in visited:
      continue

    # mark the current node as visited
    visited.add(str(current_node))

    # get the neighbors of the current node;
    neighbors = get_neighbors(current_node)

    # for each neighbor of the current node...
    for neighbor in neighbors:
      # calculate the cost of the path from the start to the neighbor
      # by adding the cost to reach the current node to the edge cost
      # to reach the neighbor from the current node
      neighbor_cost = current_cost + 1

      # if the neighbor has not been visited, or if the current path
      # to the neighbor is shorter than the previous path to the neighbor,
      # update the cost and previous node for the neighbor
      if str(neighbor) not in cost_from_start or neighbor_cost < cost_from_start[str(neighbor)]:
        cost_from_start[str(neighbor)] = neighbor_cost
        total_cost = neighbor_cost + heuristic(neighbor, goal)
        queue.put((total_cost, neighbor))
        previous[str(neighbor)] = current_node

        # store the action taken to reach the neighbor
        action = get_action(current_node, neighbor)
        actions.append(action)

        # add the current and next states, along with the action, to the data list
        data.append((current_node, neighbor, action))

  df = pd.DataFrame(data, columns=["current_state", "next_state", "action"])
  df2 = pd.DataFrame(solution, columns=["previous_state", "current_state"])
  return df, df2
```

Figure 5: A* search implementation

## 3.3 Imitation Learning Implementation

Steps for implementing A* search with imitation learning:

1. Import the following libraries: NumPy, Pandas, and scikit-learn
2. Identify the problem, the state space, the possible actions, and the goal for the learning agent.
3. Implement the A* search algorithm: define a function that takes an initial state and a goal state as inputs, returns the optimal series of actions to reach the goal.
4. Compile a set of data as the "expert demonstrations" using the A* search implementation; data should include the sequence of states and actions taken for the search to reach the goal.
5. Pre-process the expert demonstrations dataset: extract, clean and format the data to use in the supervised learning model.
6. Train a supervised learning model using the formatted data to predict the actions to take given a specific state.
7. Evaluate the imitation learning agent by checking its accuracy.

The data extracted from our A* search implementation is saved in as a series of pairs, $(s, a)$ where $s$ (observation) is represented as the variable *previous_state* and $a$ (action) as *current_state*; our optimal policy $\pi^*$, is our A* search data.

We moved our work space from our local machines to virtual machines through Google Colaboratory, to implement our machine learning algorithm due to the increase in memory usage. We created several files to run different machine learning models, aiming to find the best accuracy using the same data set.

```
df = pd.read_csv("/content/drive/MyDrive/Courses/CMPT 417/Final Project/s3_train3_csv.csv")
df = df.iloc[:,1:]
df.head()

df_test = pd.read_csv("/content/drive/MyDrive/Courses/CMPT 417/Final Project/s3_test2_csv.csv")
df_test = df_test.iloc[:,1:]
df_test.head()

pList = []
cList = []
pList.append(df['previous_state'])
cList.append(df['current_state'])

pList_test = []
cList_test = []
pList_test.append(df_test['previous_state'])
cList_test.append(df_test['current_state'])
```

Figure 6: Importing data, header removal and formatting data into arrays

```
def flatten_list(configList):
  flatten_list = []
  for x in configList:
    for y in x: #this gets [[0, 1, 3], [7, 8, 2], [5, 4, 6]] [[0, 1, 3], [7, 8, 2], [5, 4, 6]]
      nested_list = []
      for s in y: #this gets every element as a string
        if s.isdigit():
          nested_list.append(int(s))
      flatten_list.append(nested_list)
  print(flatten_list)
  return flatten_list

X = flatten_list(pList)
Y = flatten_list(cList)

X_test = flatten_list(pList_test)
Y_test = flatten_list(cList_test)

x = np.array(X)
x_test = np.array(X_test)

x = pd.DataFrame(X)
x_test = pd.DataFrame(X_test)

y = np.array(Y)
y_test = np.array(Y_test)
```

Figure 7: Pre-processing data before machine learning implementation

```
param_grid = {
    'model__estimator__n_estimators': [10, 50, 100, 200, 500, 750],
    'model__estimator__max_depth': [None, 2, 5, 7, 10],
    'model__estimator__min_samples_split': [2, 5, 10],
    'model__estimator__criterion': ['gini', 'entropy', 'log_loss']
    }
```

Figure 8: Using GridSearchCV to obtain the best hyper-parameters

GridSearchCV finds the optimal parameter, Hyperparameter, to get improvements in predictions.

```
model = RandomForestClassifier(n_estimators = 750, criterion='entropy', min_samples_split=10)
classifier = MultiOutputClassifier(model, n_jobs=-1)

# Train the model on the input data
classifier.fit(x, y)

classifier.score(x_test,y_test)
```

Figure 9: Random Forest Classifier Implementation

We imported and pre-processed our data, trained, and evaluated the models using the data. We used the following models: random forest, SVM, and XGBoost.

## 3.4 Imitation Learning with Neural Network Implementation

The extracted data is converted to tensor which is a matrix of n-dimensions that represents all types of data. All values in a tensor hold identical data type with a known (or partially known) shape. The shape of the data is the dimensions of the matrix or array.

We then define the CNN and LSTM architecture which involves choosing the number of layers and the specific types of layers to use in the network, as well as the hyper-parameters for each layer.

```
import tensorflow as tf
from tensorflow.keras import layers

x_array = X_train.values
x_tensor = tf.convert_to_tensor(x_array)

Val_array = Val_train.values
Val_tensor = tf.convert_to_tensor(Val_array)

y_tensor = tf.convert_to_tensor(y_train)
Val_y_tensor = tf.convert_to_tensor(Val_y_train)

x_tensor = tf.reshape(x_tensor, (X_train.shape[0], X_train.shape[1], 1))
Val_tensor = tf.reshape(Val_tensor, (Val_train.shape[0], Val_train.shape[1], 1))

y_tensor = tf.reshape(y_tensor, (y_train.shape[0],1 , y_train.shape[1]))
Val_y_tensor = tf.reshape(Val_y_tensor, (Val_y_train.shape[0], 1, Val_y_train.shape[1]))

x_tensor[0].shape

y_tensor[0].shape

epochs = 20
model = tf.keras.Sequential()
model.add(layers.Conv1D(32, 2, activation='relu', input_shape=x_tensor[0].shape))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling1D(pool_size=(2), strides=(2)))

model.add(layers.Conv1D(64, 2, activation='relu'))
model.add(layers.Conv1D(64, 2, activation='relu'))

model.add(layers.BatchNormalization())
model.add(layers.MaxPooling1D(pool_size=(2), strides=(2)))
```

Figure 10: Pre-processing data for CNN

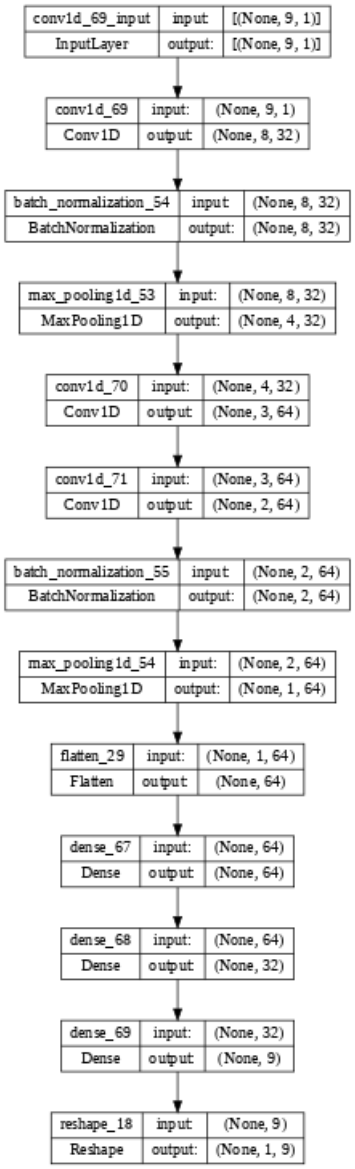



Figure 11: CNN Architecture

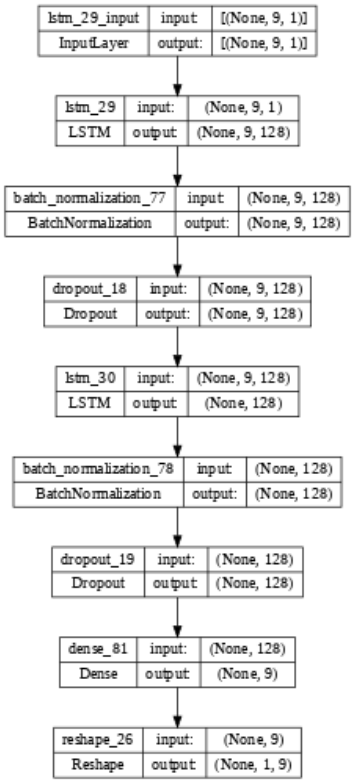



Figure 12: LSTM Architecture

# 4. Methodology

## 4.1 Problem size

We followed Korf's path to implement A* search with the Manhattan Distance heuristic function to solve the 8-puzzle [2]. We generated an 8-puzzle instance to solve and export as data; we took note of the file size and considered the memory needed for running that file for machine learning. We then produced ten instances (files) ensure we will have enough data to use for training and testing in our imitation learning implementation.

Initial we had aimed to use seven files for training and three for testing. However, due to insufficient memory on the machines we used, at the end, we wrote a function to randomly choose and merge a subset of three files for training, and two files for testing.

## 4.2 Manhattan Distance Heuristic

The Manhattan distance refers to the sum of the vertical and horizontal distance.

$$d_{(x,y)} = \sum_{i=1}^{n} |(x_i - y_i)|$$

In the case of N-puzzles, it is the minimum number of moves that each tile needs to be moved in order to achieve the goal state. This is used as our heuristic function, $h(n)$.

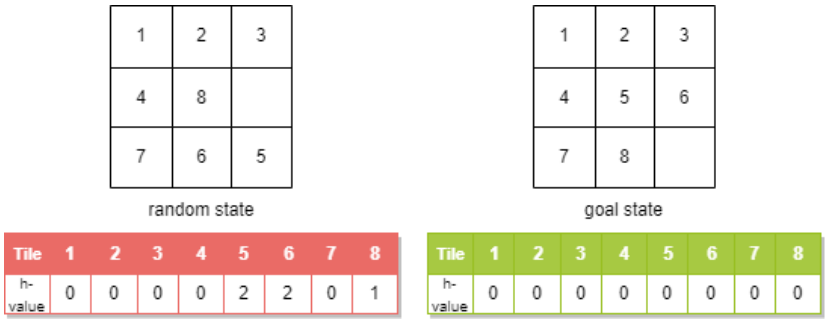



Figure 13: Manhattan Distance on a 8-puzzle state

## 4.3 Machine Learning Models

After running our search algorithm, the data is used to train various machine learning models to predict the target (path). We have tried the following machine learning models:

### 4.3.1 Random Forest

Random forest is a machine learning algorithm that is used for classification and regression tasks. It is a type of ensemble learning method, where a group of weak models are combined to create a strong predictive model.

In a random forest classifier, a large number of decision trees are trained on random subsets of the data and the final prediction is made by averaging the predictions of all the individual decision trees. This process is repeated several times, and each time a new random subset of the data is used to train the decision trees. The final prediction is made by taking the majority vote of all the individual decision tree predictions.

### 4.3.2 Support Vector Machine (SVM)

Support Vector Machines (SVMs) are a type of supervised machine learning algorithm that can be used for classification or regression tasks. The goal of an SVM is to find the hyperplane in a high-dimensional space that maximally separates the different classes.

In an SVM classifier, the data is transformed into a higher-dimensional space using a kernel function, and the classifier finds the hyperplane that maximally separates the classes in this higher-dimensional space. The classifier then uses the support vectors, which are the data points closest to the hyperplane, to make predictions on new data.

### 4.3.3 XGBoost

XGBoost is a machine learning algorithm that stands for ”eXtreme Gradient Boosting.” It is a type of boosting algorithm that is used for classification and regression tasks. Boosting algorithms are a type of ensemble learning method, where a group of weak models are combined to create a strong predictive model. In XGBoost, decision trees are used as the weak models, and the final prediction is made by combining the predictions of all the individual decision trees.

### 4.3.4 Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) are a type of neural network specifically designed for image classification tasks. They are particularly well-suited for image classification because they are able to automatically learn hierarchical feature representations from the raw pixel data of the image. In a CNN classifier, the input image is passed through a series of layers, each of which applies a series of filters to the image to extract different features.

Using a CNN as an imitation learner to solve a puzzle problem is to train the CNN on a large dataset of expert puzzle-solving actions and use it to predict the most appropriate action to take at each step of the puzzle-solving process.

### 4.3.5 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) that is specifically designed to model long-term dependencies in sequence data. RNNs are a type of neural network that are able to process sequential data, such as time series data or natural language text.

In an LSTM classifier, the input data is processed through a series of LSTM cells, which are able to retain information from previous time steps and use it to inform the prediction at the current time step.

An LSTM network could potentially be used as an imitation learner to solve a puzzle problem by learning to predict the next action to take based on the sequence of actions taken so far in the puzzle-solving process, allowing it to reason about the long-term dependencies and relationships between different actions. The LSTM is able to learn and incorporate information about the long-term dependencies and relationships between different actions, allowing it to effectively solve the puzzle by making informed decisions about which actions to take. This approach can be particularly useful in situations where the solution to the puzzle involves considering the long-term consequences of different actions, or where there are complex relationships between different elements of the puzzle.

# 5. Experimental Setup

## 5.1 Local Machine

The local machine we used to run our A* search for data has the following properties:

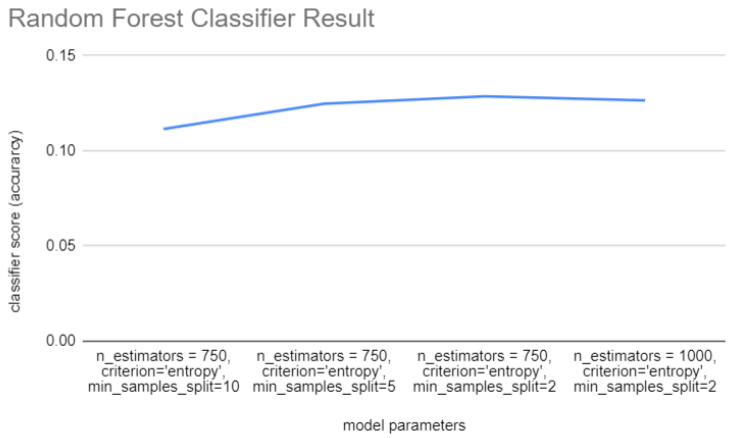
- Python version: Python 3.9.7
- Windows version: Windows 11
- CPU: Intel core i7 9th Gen
- RAM: 16GB RAM

## 5.2 Google Colaboratory

The virtual machine, Python 3 Google Compute Engine Backend, we used through Google Colab uses Python 3.8, has 12.68GB of RAM, and Nvidia's T4 GPU. Later on, we purchased plan that would allow us to use more RAM, the new virtual machine has 166.77GB of RAM with V100 or A100 Nvidia GPU.

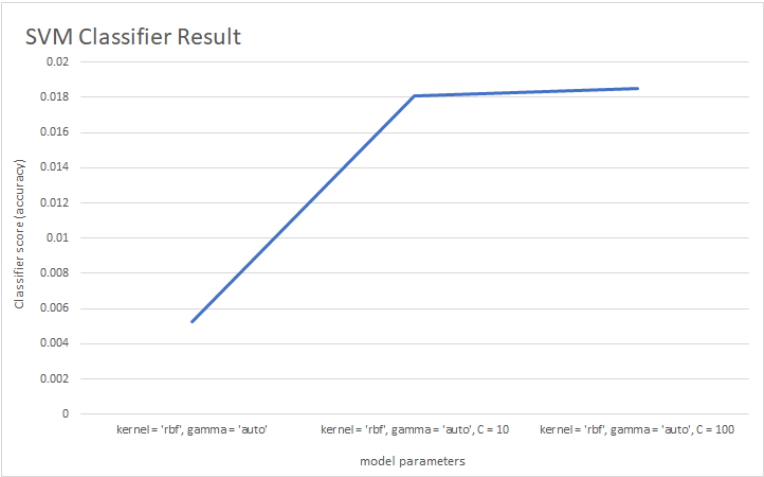
# 6. Results

## 6.1 Random Forest Model




Best Classification score for Random Forest Model: 12.9%

## 6.2 Support Vector Machine (SVM) Model

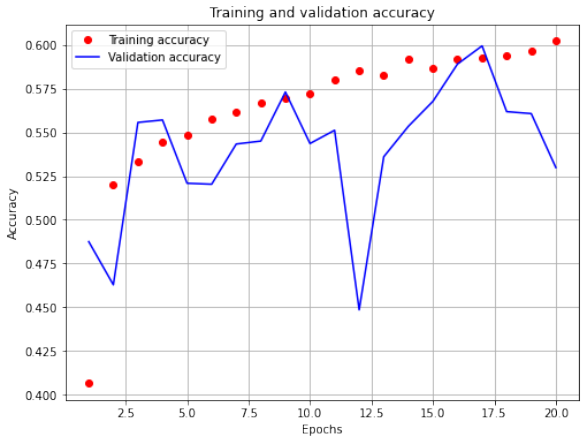


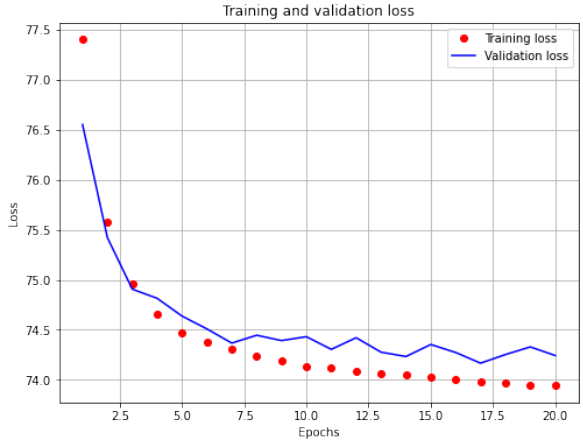

Best Classification score for SVM Model: 1.85%

## 6.3 XGBoost Model

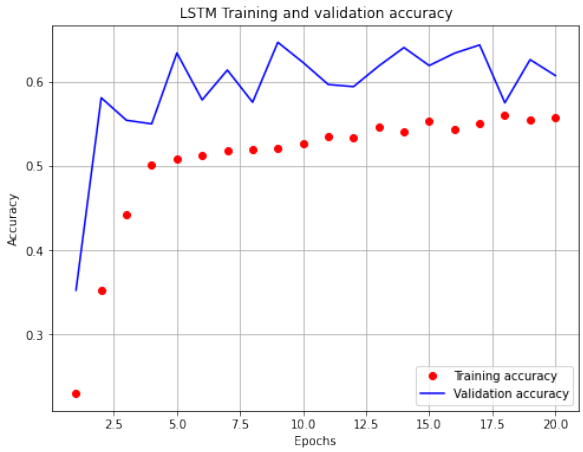

Best Classification score for XGBoost Model: 14.9%

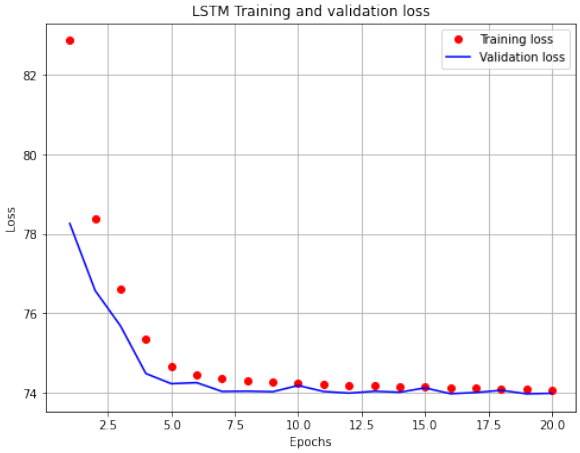
## 6.4 Convolutional Neural Network (CNN)



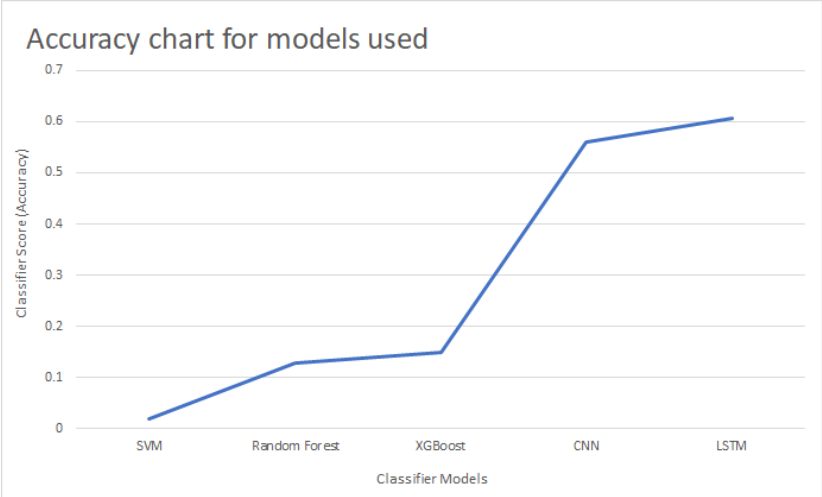




## 6.5 Long Short-Term Memory (LSTM)






## 6.6 Compare All Models




Our best reults came from the CNN and LSTM models with an accuracy of 55.9% and 60.6% respectively.

LSTM models can perform better than other models in tasks that involve sequential data and long-term dependencies. This is because LSTMs are specifically designed to model long-term dependencies in sequence data, and are able to retain information from previous time steps and use it to inform the prediction at the current time step.

# 7. Conclusions

As shown in Figure 6.6, with various machine learning models we tested, LSTM is significantly more accurate in predicting whilst using A* search.

It is important to note that we have only tested the models with a small dataset due to memory capacity issues. As mentioned in section 4.1, problem size is crucial. With larger datasets, the accuracy of the models are bounded to increase, but household machines are not capable of running the process without crashing. While we were only using a subset of our produced A* search data, when we started implementing the neural network machine learning algorithms, we were forced to pay for a better virtual machine to run the algorithms.

One potential use of imitation learning in puzzle solving is in the field of computer science, where it could be used to develop algorithms that can solve complex problems more efficiently. For example, an algorithm trained using imitation learning could be used to solve optimization problems or to find the most efficient solution to a particular problem. Imitation learning could also be used to solve puzzle problems in fields such as biology, where it could be used to analyze large datasets and identify patterns or trends that might not be apparent to humans. For example, it could be used to identify relationships between different genes or to predict the outcomes of experiments based on past results. Overall, the potential uses for imitation learning in puzzle solving are vast and will depend on the needs and goals of various fields and industries. As technology continues to advance, it is likely that imitation learning will play a significant role in solving complex problems in a variety of fields.

In conclusion, imitation learning can be a useful approach for solving puzzle problems using expert search algorithms. By observing and learning from the actions of a human expert or a pre-trained model, the imitation learning model can effectively find solutions to the puzzle problem. This approach can be particularly useful in situations where it is difficult to encode a precise set of rules or heuristics for solving the problem, or when the solution space is large and complex. However, it is important to carefully consider the limitations of imitation learning, as it may not be suitable for all types of puzzle problems and may require a large amount of expert data in order to learn effectively.

# References


[1] W. W. Johnson and W. E. Story. Notes on the ”15” puzzle. *American Journal of Mathematics*, 2(4):397–404, Dec 1879.

[2] R. E. Korf. Depth-limited search for real-time problem solving. *Real-Time Systems*, 2:7–24, 1990.

[3] R. E. Korf and L. A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *AAAI/IAAI, Vol. 2*, 1996.