AI AND MACHINE LEARNING REPORT

STUDENT ID: F418431

CONTENTS

TASK 1

A)

INTRODUCTION

The objective of the first task is to develop a machine learning model capable of classifying images into their respective categories. The dataset consists of images of various objects. To achieve accurate classification, an appropriate machine learning approach must be selected, implemented, trained, and evaluated in this report and shown in the code. For this first task, I used CNN's, which stands for **convolutional Neural Networks (CNNs)**, and these are a specialised class of neural networks designed to process grid-like data, such as images [27].

WHY CNNS?

CNNs are widely used in image classification because they can be used to detect patterns and features in images and recognise objects, classes, and categories [1]. CNNs automatically learn spatial hierarchies of data, making them ideal for this task. Apart from that overall, I picked CNNs for this task because;

- **FEATURE LEARNING FROM PIXELS:** CNNs automatically discover important feature from images directly from pixel data through convolutional layers, pooling, and non-linear activations, unlike traditional methods that often rely on manually designed features (e.g, edges).[5]

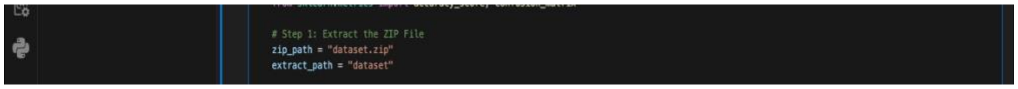
- **SPATIAL HIERARCHY:** In Convolutional Neural Networks (CNNs), the idea of "spatial hierarchies" explains how these networks gradually learn and pick out more complex details from images (or other types of data). Early layers focus on simple patterns like edges, corners, or textures. As we go deeper into the network, those basic building blocks get combined into more detailed shapes and object parts. Eventually, in the final layers, the CNN is able to recognise entire objects. This layered approach is a big part of why CNNs perform so well on tasks like image recognition, and this would be needed especially because the dataset consists of a large set of images.[6]

- **BUILT-IN TRANSLATIONAL INVARIANCE:** Because of how convolution filters move across the image, CNNs can recognise objects despite how the input is shifted [8].

- **WEIGHT SHARING FEATURES:** Techniques like pooling and weight sharing reduce the number of trainable network parameters, further helping the model generalise better from limited data and avoid overfitting [9].

CNN MODEL IMPLEMENTATION

To implement the CNN model, the code begins by **extracting** the dataset ZIP file into a specified folder; for this task I ran the zip locally on my laptop. Each subfolder under train or val corresponds to one class, ensuring that ImageDataGenerator can automatically label the images based on their directory names.



```
# Step 1: Extract the ZIP File
zip_path = "dataset.zip"
extract_path = "dataset"
```

The **Sequential** CNN is defined with the following layers:

CONVOLUTIONAL LAYERS

- Filters (kernel) scan over images to capture edges, textures, or other local details.
- Learns these filters during training.
- To generate a feature map that identifies regions where the pattern is present, the filter(kernel) computes dot products between its weights and the relevant picture pixels [10].
- The convolutional layer is the core CNN architectural function that allows the model to derive significant features from incoming data.

ACTIVATION FUNCTION

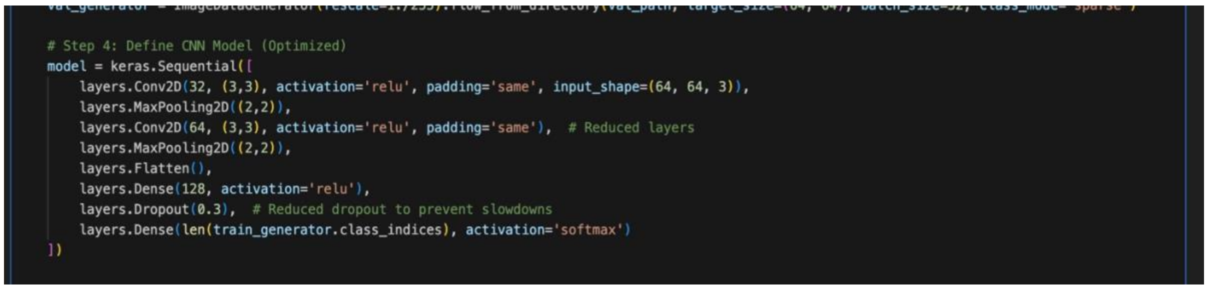
- Adds non-linearity, helping the network learn complex patterns.
- At the conclusion of the network, the activation function determines which model information should fire forward and which should not.

POOLING LAYERS

- Down samples feature maps (e.g., 2×2 max pooling).
- Reduces the computation load and helps avoid overfitting.
- Simplifies the network and retains essential features for further processing [12].

FLATTEN + DENSE LAYERS

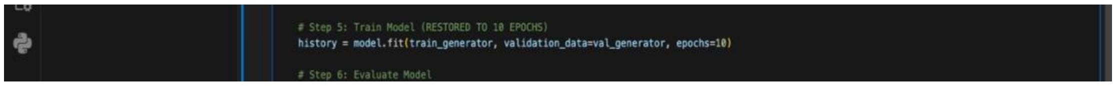
- Flatten transforms 2D feature maps into a 1D vector [11].
- Dense (fully connected) layers combine extracted features to make final classifications. [3]



```python
# Step 4: Define CNN Model (Optimized)
model = keras.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', padding='same', input_shape=(64, 64, 3)),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, (3,3), activation='relu', padding='same'),  # Reduced layers
    layers.MaxPooling2D((2,2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.3),  # Reduced dropout to prevent slowdowns
    layers.Dense(len(train_generator.class_indices), activation='softmax')
])
```

TRAINING AND TESTING THE MODEL

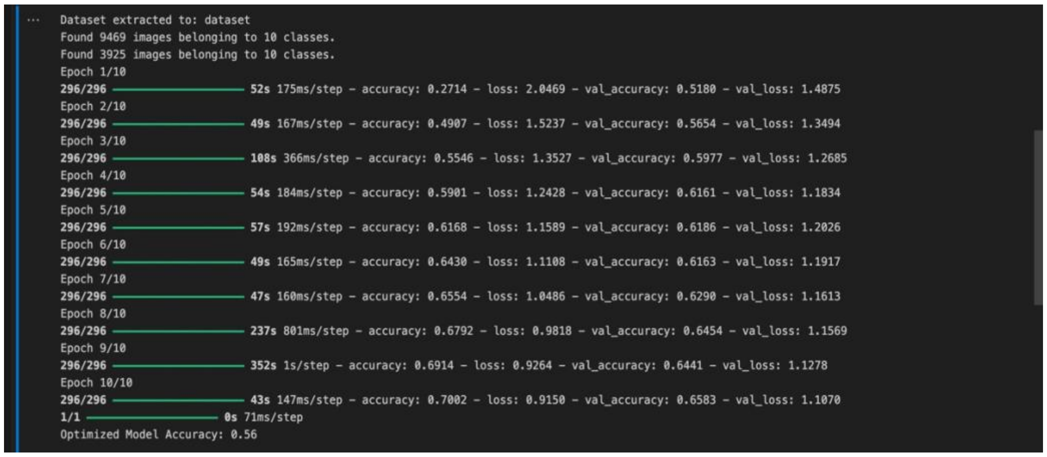
1) **DATASET PREPROCESSING:**
   - Images were normalised (rescaled to [0,1]) to improve training stability, which helps the model to converge faster and for performance [13].
   - Data augmentation techniques such as rotation and horizontal flipping were applied.
   - Training and validation data were loaded using ImageDataGenerator.
2) **TRAINING:**
   - The model was trained using batches of 32 images for 10 epochs.,Although 10 might seem brief, it’s enough to see meaningful trends in loss and accuracy.



```python
# Step 5: Train Model (RESTORED TO 10 EPOCHS)
history = model.fit(train_generator, validation_data=val_generator, epochs=10)

# Step 6: Evaluate Model
```

   - Training accuracy gradually improved, reaching around **70%**, while validation accuracy stabilized at **65%**.



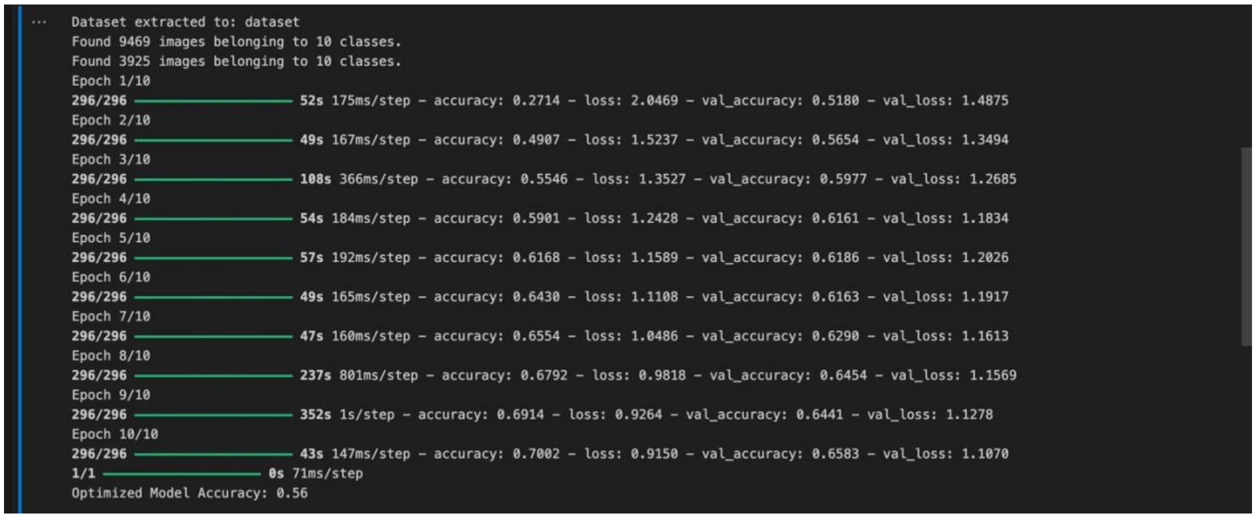
```
···  Dataset extracted to: dataset
     Found 9469 images belonging to 10 classes.
     Found 3925 images belonging to 10 classes.
     Epoch 1/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 52s 175ms/step - accuracy: 0.2714 - loss: 2.0469 - val_accuracy: 0.5180 - val_loss: 1.4875
     Epoch 2/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 49s 167ms/step - accuracy: 0.4907 - loss: 1.5237 - val_accuracy: 0.5654 - val_loss: 1.3494
     Epoch 3/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 108s 366ms/step - accuracy: 0.5546 - loss: 1.3527 - val_accuracy: 0.5977 - val_loss: 1.2685
     Epoch 4/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 54s 184ms/step - accuracy: 0.5901 - loss: 1.2428 - val_accuracy: 0.6161 - val_loss: 1.1834
     Epoch 5/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 57s 192ms/step - accuracy: 0.6168 - loss: 1.1589 - val_accuracy: 0.6186 - val_loss: 1.2026
     Epoch 6/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 49s 165ms/step - accuracy: 0.6430 - loss: 1.1108 - val_accuracy: 0.6163 - val_loss: 1.1917
     Epoch 7/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 47s 160ms/step - accuracy: 0.6554 - loss: 1.0486 - val_accuracy: 0.6290 - val_loss: 1.1613
     Epoch 8/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 237s 801ms/step - accuracy: 0.6792 - loss: 0.9818 - val_accuracy: 0.6454 - val_loss: 1.1569
     Epoch 9/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 352s 1s/step - accuracy: 0.6914 - loss: 0.9264 - val_accuracy: 0.6441 - val_loss: 1.1278
     Epoch 10/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 43s 147ms/step - accuracy: 0.7002 - loss: 0.9150 - val_accuracy: 0.6583 - val_loss: 1.1070
     1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step
     Optimized Model Accuracy: 0.56
```

3) **TESTING & EVALUATION:**
   - The model was tested on unseen validation images.
   - A **confusion matrix** was generated to analyse classification performance per class.
   - The final model accuracy was **56%**, indicating that some classes were harder to classify than others.

EVALUATION PROTOCOL

To understand the model's strengths and weaknesses, the following methods were used:

- **ACCURACY SCORE:** This metric gives a straightforward snapshot of how many labels were predicted correctly. During training, the network was observed gradually improving its accuracy across epochs [15].
- **CONFUSION MATRIX:** A confusion matrix provides a grid showing which classes are predicted accurately and which ones the model confuses with others. This proved especially valuable, revealing that some visually similar classes frequently get mixed up.
- **LOSS AND ACCURACY CURVES:** By plotting these metrics over training epochs, we could watch for signs of overfitting [14].

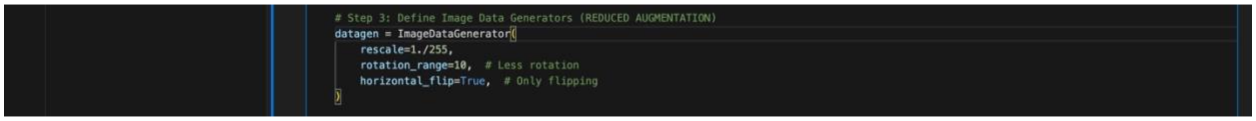


```
...  Dataset extracted to: dataset
     Found 9469 images belonging to 10 classes.
     Found 3925 images belonging to 10 classes.
     Epoch 1/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 52s 175ms/step - accuracy: 0.2714 - loss: 2.0469 - val_accuracy: 0.5180 - val_loss: 1.4875
     Epoch 2/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 49s 167ms/step - accuracy: 0.4907 - loss: 1.5237 - val_accuracy: 0.5654 - val_loss: 1.3494
     Epoch 3/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 108s 366ms/step - accuracy: 0.5546 - loss: 1.3527 - val_accuracy: 0.5977 - val_loss: 1.2685
     Epoch 4/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 54s 184ms/step - accuracy: 0.5901 - loss: 1.2428 - val_accuracy: 0.6161 - val_loss: 1.1834
     Epoch 5/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 57s 192ms/step - accuracy: 0.6168 - loss: 1.1589 - val_accuracy: 0.6186 - val_loss: 1.2026
     Epoch 6/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 49s 165ms/step - accuracy: 0.6430 - loss: 1.1108 - val_accuracy: 0.6163 - val_loss: 1.1917
     Epoch 7/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 47s 160ms/step - accuracy: 0.6554 - loss: 1.0486 - val_accuracy: 0.6290 - val_loss: 1.1613
     Epoch 8/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 237s 801ms/step - accuracy: 0.6792 - loss: 0.9818 - val_accuracy: 0.6454 - val_loss: 1.1569
     Epoch 9/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 352s 1s/step - accuracy: 0.6914 - loss: 0.9264 - val_accuracy: 0.6441 - val_loss: 1.1278
     Epoch 10/10
     296/296 ━━━━━━━━━━━━━━━━━━━━ 43s 147ms/step - accuracy: 0.7002 - loss: 0.9150 - val_accuracy: 0.6583 - val_loss: 1.1070
     1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step
     Optimized Model Accuracy: 0.56
```

B)

DATA AUGMENTATION

The data was also augmented to increase the size and diversity of a training dataset artificially which was done by applying transformations to existing data. This reduces the likelihood that the network overfits to a particular orientation, which is a common problem in machine learning. This then improves its generalisation to new images by making it more robust to different variations in the data. Data augmentation also improves the accuracy due to learning from more diverse datasets and is a very cost-effective way to increase diversity and size of a training dataset, which is why it was used [16].



```
# Step 3: Define Image Data Generators (REDUCED AUGMENTATION)
datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=10,  # Less rotation
    horizontal_flip=True,  # Only flipping
)
```

MODEL ARCHITECTURE ADJUSTMENTS

- **REDUCED DEPTH:** The CNN is intentionally kept to just a couple of convolutional blocks. A deeper CNN might overfit or require a larger dataset.
- **DROPOUT RATE TUNING:** A dropout rate of 30% helped strike a balance between regularisation and feature learning.
- **ADAM OPTIMIZER:** The 0.001 learning rate is a solid default, though further fine-tuning could help the model converge more optimally.

FINDINGS AND REFLECTION

- **MISCLASSIFICATIONS:** Some classes were predictably harder to classify because of similar shapes or even color distributions, e.t.c. Adjusting the model in later improvements to be deeper or using specialised layers might help in distinguishing these subtle differences.
- **ROOM FOR IMPROVEMENT:** Though 65% validation accuracy is decent, methods like **transfer learning**, where a pre-trained network like ResNet is fine-tuned on this dataset, would give a bigger performance boost and could be considered for later Improvement[17].
- **Parameter Exploration:** While 10 epochs revealed the CNN's potential, testing more epochs, varying batch sizes, or tuning learning rates might lead to further performance gains, which could also lead to more accuracy. Different types of augmentation could further be used to improve generalisation

Overall, the CNN demonstrated reasonable classification ability for this dataset. The learning curves suggest that longer training or more extensive augmentations might push accuracy further. Despite the challenges, the model confidently classifies a majority of images correctly, highlighting the power of CNNs in automated image analysis.

TASK 2

B)

INTRODUCTION

In this task, we use two methods which are **Value Iteration** and **Q-Learning** to help our agent navigate a GridWorld. The GridWorld is a maze-like environment where each cell can be one of the following:

- **Wall (w):** The agent cannot pass through.
- **Obstacle (o):** Stepping here incurs a penalty.
- **Goal (g):** final goal to return a big reward to the agent.
- **Free space (or agent start 'a')**

The **agent** is our decision-making entity that starts at a designated position (usually marked with an 'a' or at the top-left corner) and takes actions (up, down, left, right) to reach the goal. It learns what to do either by using a model of the environment (Value Iteration) or by interacting with the environment and updating its knowledge (Q-Learning).

HOW VALUE ITERATION WORKS

Value Iteration is a dynamic programming approach for finding the optimal value function V* by solving the Bellman equations iteratively, it does this by Utilising the idea of dynamic programming, it continuously improves until it converges to (or approaches) the optimal value function, then maintaining a value function that comes close to it. [18]. In my code, I first extracted all valid states (all non-wall cells) and initialised the value function to zero for each state.

I then loop over all states repeatedly. For each state, I consider all possible actions (up, down, left, right) available. For each action, I simulate taking that action by temporarily setting the environment's state and calling the step or action function. This function returns the next state, which could be the immediate reward and whether the episode is done or finished per se (for instance, if the agent hits a wall or reaches the goal).

I calculated a candidate value for each action as the sum of the immediate reward and the discounted value of the next state (if the episode isn't over). Mathematically, it looks like this:

candidate=reward+$\gamma \times V(s')$ (if not done)

If the episode ends after taking the action, I simply take the reward. I then choose the maximum candidate value over all actions and update V accordingly. I also keep track of the maximum change over all states to ensure accuracy, and once a state falls below a small threshold, which will be close to the optimal value function, I consider V to have converged [19].

After the value function converges, I extract a policy by going through each state and picking the action that yields the highest $R(s,a)+\gamma\, V(s')R(s,a)+\gamma V(s')$ [20].

This process is very systematic. Since the environment is small, Value Iteration converges quickly. It gives a clear, optimal policy for navigating the GridWorld.

HOW Q-LEARNING WORKS

Q-Learning, in contrast, is a model-free, value based, off policy algorithm that determines the optimal course of action depending on the agent's present condition(state). Quality is what the "Q" stands for. Quality is a measure of the action's worth in optimising future rewards. Essentially, I don't assume that I know the full transition dynamics of the environment; instead, the agent learns by exploring. Overall, Model-free algorithms use experience without a reward system or transition to learn the effects of their actions. Using a value-based approach, the value function is trained to determine which states are more valuable and behave accordingly, and finally, in the off-policy, the algorithm evaluates and updates a policy that is different from the policy used to take an action [21].

I begin by initialising a Q-table, stored as a dictionary of dictionaries. For every action available from that state, I set the initial Q-value to zero. Then, I run many episodes—say, 1000. In each episode, I reset the environment so that the agent starts at the initial position.

Within each episode, for a fixed maximum number of steps, the agent chooses an action using an epsilon-greedy policy. This means that with probability (I used 0.1), the agent picks a random action; otherwise, it picks the action that currently has the highest Q-value.

Once the agent takes an action, it receives a reward and transitions to a new state. I update the Q-value for that state-action pair using the formula:

$$Q(s,a)\leftarrow Q(s,a)+\alpha[r+\gamma a'\max Q(s',a')-Q(s,a)]$$

Here, α is the learning rate (set to 0.1), and γ is the discount factor (again, 0.9). This update rule gradually improves the Q-values as the agent gathers more experience. After each episode, the Q-table becomes a better approximation of the true action values.

Finally, once training is complete, I extract the policy by choosing, for each state, the action with the highest Q-value.

COMPARING THE TWO METHODS

I compared Value Iteration and Q-learning on my Grid World, which would essentially be the advantages and disadvantages of both, and here's what I found:

- **MODEL DEPENDENCE:**
  Value Iteration requires a complete model of the environment while Q-Learning doesn't need this information as instead of requiring prior knowledge about an environment, the Q-learning agent can learn about the environment as it trains (it learns purely from experience, making it more flexible in unknown environments) [25].
- **CONVERGENCE SPEED:**
  Value Iteration is guaranteed to converge to the optimal value function under certain conditions, However, it can converge slowly for certain problems [26]. Q-Learning, however, Finding the ideal balance between experimenting with new actions and

sticking to what is previously understood can be challenging which is called the exploration vs. exploitation tradeoff, learning could also take a long time to figure out the best method if there are several ways to approach a problem in Q learning[25].

- **QUALITY OF THE POLICY:**
  In my experiments, both methods eventually produced a policy that guided the agent from its starting position to the goal. Value Iteration produced an optimal policy once the value function converged. Q-Learning's policy, while eventually nearly identical, depended on careful tuning of hyperparameters .

Overall, in this GridWorld, Value Iteration was easier to implement and faster to converge because the environment was small and deterministic. Q-Learning is more general, but it comes at the cost of needing more interactions and careful parameter tuning.

TASK 2

C)

EFFECTS OF DISCOUNT FACTOR AND EXPLORATION RATE

UNDERSTANDING THE PARAMETERS

In Q-Learning, two critical hyperparameters are the discount factor (γ) and the exploration rate (ϵ). Let me explain what they do in plain terms:

- **Discount Factor (γ)**:
  This parameter balances immediate and future rewards [22]. A higher discount factor γ (close to 1) indicates that the agent will give more emphasis to long-term benefits, whereas a lower discount factor indicates that the agent would focus more on immediate rewards. In my GridWorld, where reaching the goal gives a large reward (+80) but may require several moves (each with a penalty of -1 or -15), a higher discount factor (around 0.9) helps the agent see that taking a few penalties along the way is worth it and in order to eventually get the reward so they would priotise this. If I set the discount factor too low, the agent might avoid moving too much or miss the goal because the future reward is heavily discounted [23].
- **Exploration Rate (ϵ)**:
  The exploration rate defines the probability that the agent will select a random action instead of the one with the highest Q-value or defines how often the agent explores new actions instead of using known best actions (exploits) [22]. A higher ϵ means the agent explores more, which is good early on to ensure that all parts of the maze are tried out. However, if ϵ remains high, the agent may continue

taking random actions and not fully exploit what it has learned. Conversely, if ϵ is too low, the agent might not explore enough, risking not exploring enough. In my experiments, I found that an ϵ of about 0.1 strikes a good balance, allowing the agent to explore occasionally while mostly following the best-known strategy once it has gathered sufficient information from exploring or from experience.

EXPERIMENTAL EFFECTS

In practical terms, here's how these parameters affect the agent's performance:

1. **DISCOUNT FACTOR (Γ):**

   With a **high γ** (e.g., 0.9 or above), I observed that the agent is more willing to move away from the immediate penalties to eventually reach the goal. This leads to more optimal, long-term strategies, though it might also slow down learning because the agent has to consider the long chain of rewards and penalties, while with a **low γ** (e.g., 0.4), the agent might ignore the potential future reward at the goal because the rewards are heavily discounted. This can result in the agent taking a "safer" but could not be the optimal route, or even failing to reach the goal if the immediate costs are too high as the agent will not be willing to explore.

2. **EXPLORATION RATE (ϵ):**
   - A **high ϵ** (like 0.3) would make the agent try many random actions. This is useful to discover parts of the maze that might not be immediately obvious.
   - In my setup, an ϵϵof **0.1** worked well, as it allowed enough exploration during the learning phase while still letting the agent capitalise on its accumulated knowledge from exploring.

CONCLUSION

In summary, my experiments showed that both Value Iteration and Q-Learning can effectively solve the GridWorld problem, but each has its own advantages and limitations:

- **Value Iteration** is fast and reliable in a small, deterministic environment since it uses the known model to update a value function quickly.
- **Q-Learning** is more flexible and doesn't require a known model, making it suitable for larger or unknown environments. However, it relies heavily on the right tuning of the discount factor (γ) and exploration rate (ϵ).

REFERENCES

1. MathWorks (n.d.). *What Is a Convolutional Neural Network? | 3 things you need to know*. [online] uk.mathworks.com. Available at: https://uk.mathworks.com/discovery/convolutional-neural-network.html
2. Lyzr. (2024). *Understanding CNNs: A Comprehensive Guide to Convolutional Neural Networks*. [online] Available at: https://www.lyzr.ai/glossaries/cnn/
3. Gurucharan, M. (2020). *Basic CNN Architecture: Explaining 5 Layers of Convolutional Neural Network*. [online] upGrad blog. Available at: https://www.upgrad.com/blog/basic-cnn-architecture/.
4. KALRA, K. (2023). *Convolutional Neural Networks for Image Classification*. [online] Medium. Available at: https://medium.com/@khwabkalra1/convolutional-neural-networks-for-image-classification-f0754f7b94aa
5. Team, F.A. (2025). *Image Recognition Algorithms: CNN, R-CNN, YOLO Explained*. [online] Flypix. Available at: https://flypix.ai/blog/image-recognition-algorithms/
6. Ohiri, E. (2024). *What are Convolutional Neural Networks (CNNs)?*[online] CUDO Compute. Available at: https://www.cudocompute.com/blog/what-are-convolutional-neural-networks-cnns
7. Rosebrock, A. (2021). *Are CNNs invariant to translation, rotation, and scaling? - PyImageSearch*. [online] PyImageSearch. Available at: https://pyimagesearch.com/2021/05/14/are-cnns-invariant-to-translation-rotation-and-scaling
8. Cross Validated. (n.d.). *machine learning - What is translation invariance in computer vision and convolutional neural network?* [online] Available at: https://stats.stackexchange.com/questions/208936/what-is-translation-invariance-in-computer-vision-and-convolutional-neural-netwo
9. Alzubaidi, L., Zhang, J., Humaidi, A.J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M.A., Al-Amidie, M. and Farhan, L. (2021). Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data*, [online] 8(1). doi:https://doi.org/10.1186/s40537-021-00444-8
10. Linkedin.com. (2025). *Discover thousands of collaborative articles on 2500+ skills*. [online] Available at: https://www.linkedin.com/pulse/understanding-convolutional-neural-networks-cnns-elhousieny-phd
11. Yamashita, R., Nishio, M., Do, R.K.G. and Togashi, K. (2018). Convolutional Neural networks: an Overview and Application in

Radiology. *Insights into Imaging*, 9(4), pp.611–629. doi:https://doi.org/10.1007/s13244-018-0639-9

12. FyndAcademy (2024). *Pooling Layers in CNN: Complete Explanation in 2025*. [online] Fynd.academy. Available at: https://www.fynd.academy/blog/pooling-layers-in-cnn
13. Github.io. (2025). *All in One View*. [online] Available at: https://carpentries-incubator.github.io/intro-image-classification-cnn/aio.html
14. Srivastava, T. (2019). *Evaluation Metrics Machine Learning*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2019/08/11-important-model-evaluation-error-metrics/.
15. Medium.com. (2025). *Blocked Page*. [online] Available at: https://medium.com/towards-data-science/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234
16. Hallaj, P. (2023). *Data Augmentation: Benefits and Disadvantages*. [online] Medium. Available at: https://medium.com/@pouyahallaj/data-augmentation-benefits-and-disadvantages-38d8201aead
17. Fagbuyiro, D. (2024). *Guide To Transfer Learning in Deep Learning - David Fagbuyiro - Medium*. [online] Medium. Available at: https://medium.com/@davidfagb/guide-to-transfer-learning-in-deep-learning-1f685db1fc94
18. gibberblot.github.io. (n.d.). *Value Iteration — Introduction to Reinforcement Learning*. [online] Available at: https://gibberblot.github.io/rl-notes/single-agent/value-iteration.html
19. Shivang Shrivastav (2024). *Bellman Equation and Value iteration in Dynamic Programming*. [online] Medium. Available at: https://shivang-ahd.medium.com/bellman-equation-and-value-iteration-in-dynamic-programming-609219f5d3e1
20. Hatcher, E. (2022). *Reinforcement Learning: From A to Q - Eli Hatcher - Medium*. [online] Medium. Available at: https://medium.com/@eli.hatcher/reinforcement-learning-from-a-to-q-7c1b2f96748
21. Abid Ali Awan (2022). *An Introduction to Q-Learning: A Tutorial For Beginners*. [online] Datacamp.com. Available at: https://www.datacamp.com/tutorial/introduction-q-learning-beginner-tutorial

22. Saxena, A. (2024). *Q Learning in Machine Learning [Explained by Experts]*. [online] Applied AI Blog. Available at: https://www.appliedaicourse.com/blog/q-learning-in-machine-learning/.

23. udit (2023). *The Q in Q-learning: A Comprehensive Guide to this Powerful Reinforcement Learning Algorithm*. [online] Medium. Available at: https://itsudit.medium.com/the-q-in-q-learning-a-comprehensive-guide-to-this-powerful-reinforcement-learning-algorithm-896cbbedcd33.

24. Cross Validated. (n.d.). *Understanding the role of the discount factor in reinforcement learning*. [online] Available at: https://stats.stackexchange.com/questions/221402/understanding-the-role-of-the-discount-factor-in-reinforcement-learning

25. Kerner, S.M. (2023). *What is Q-learning?* [online] Enterprise AI. Available at: https://www.techtarget.com/searchEnterpriseAI/definition/Q-learning

26. Shivang Shrivastav (2024). *Bellman Equation and Value iteration in Dynamic Programming*. [online] Medium. Available at: https://shivang-ahd.medium.com/bellman-equation-and-value-iteration-in-dynamic-programming-609219f5d3e1

27. geeksforgeeks (2020). *Convolutional Neural Network (CNN) in Machine Learning*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/convolutional-neural-network-cnn-in-machine-learning/