AI & Machine Learning Coursework.

Task 1 – Image Classification.

Introduction.

A pivotal component of machine learning is image classification, where images are classified by an algorithm into a singular or several predetermined classes. This method is used in AI applications like object recognition, medical diagnostics and image searching. In this report, an image dataset, with meaningful objects such as a French horn, a golf ball and a parachute are used to create a machine learning model that classifies new test images into their classes accurately.

Focusing on deep learning approaches like convolutional neural networks (CNNs), I explored how machine learning models can perform image classifications by first developing a model and setting the parameters for the training data and testing data (validation data). Different techniques were executed to improve accuracy of the models, like modifications to the central architecture and data augmentation. An analysis of accuracy and loss curves provided an insight into how effective the models are. This report describes the methodology, experimental data and key findings from the varied machine language approaches.

Methods.

To create a model that accurately predicts image classes, the dataset first underwent data preprocessing. This is an essential procedure as the dataset must be explored to grasp the datasets features and distinguish any possible outliers that could result in misleading conclusions.

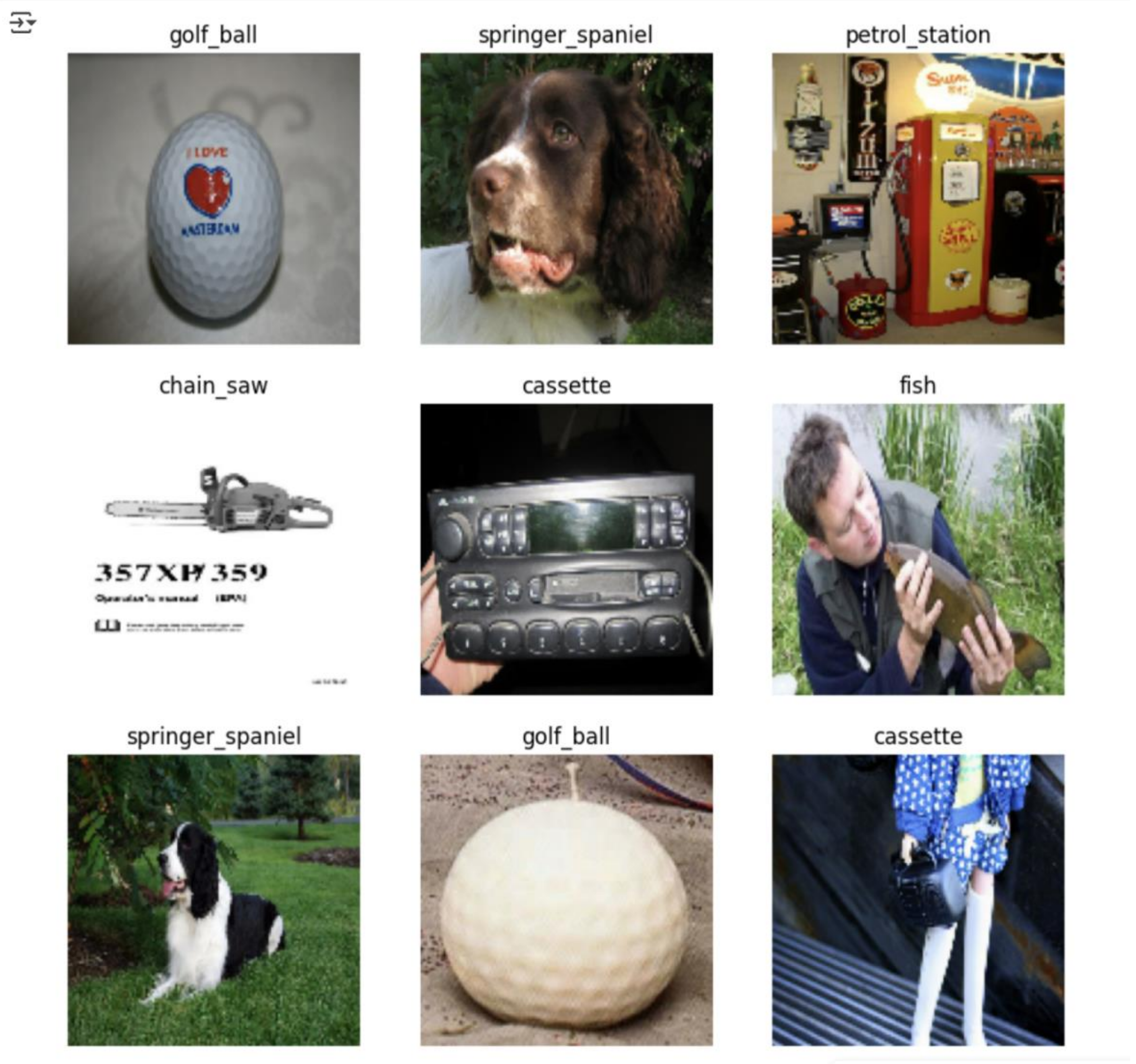
The images in the dataset were resized to 128x128 (width x height) and the batch size was set to 64. Batch size is an important hyperparameter in image processing that determines how many training samples are in each iteration. The batch size was initially 32, however this was later changed to 64 due to the slower training time. Although training time is often faster with a larger batch, it was important to consider the change to 64 might lead to a lower accuracy and overfitting. Fundamentally, a batch size of 64 was chosen over 32 due to being less time consuming and having a lower computational cost.

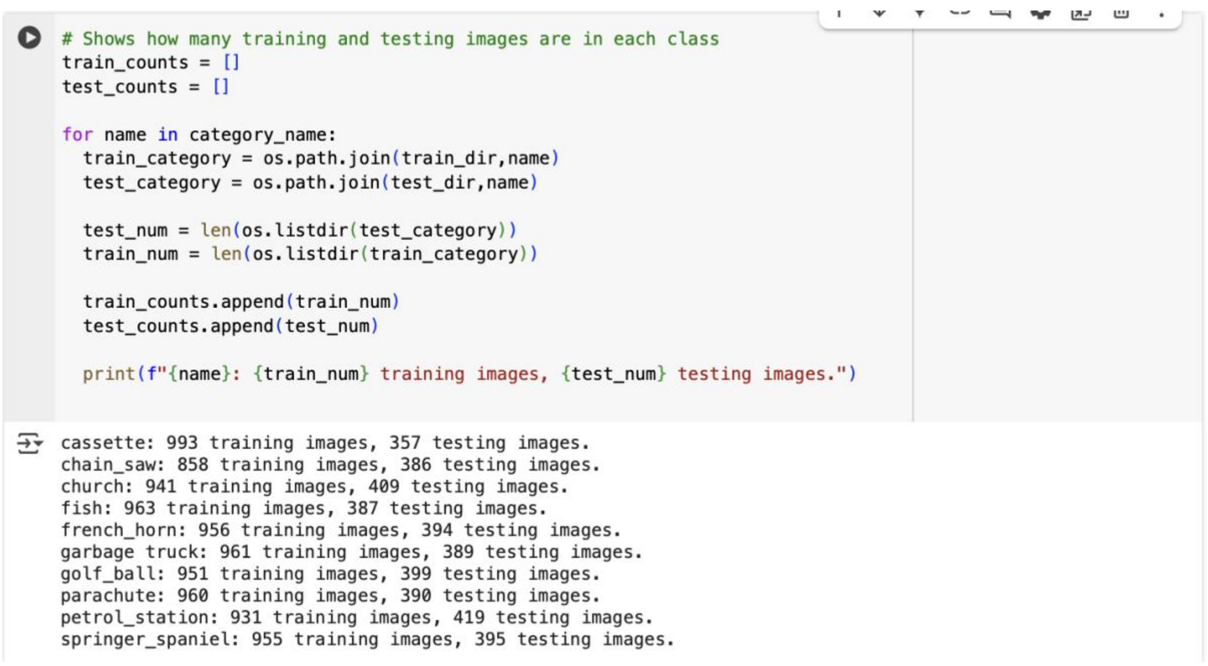
When exploring the data I noticed the images assigned numbers rather than descriptive labels, making them inaccessible. The folders were renamed by creating a dictionary which mapped the labels for easier readability. This can be seen below.

```
# The folders are renamed for classification.

rename_label = {
    'n01440764': 'fish',
    'n02102040': 'springer_spaniel',
    'n02979186': 'cassette',
    'n03000684': 'chain_saw',
    'n03028079': 'church',
    'n03394916': 'french_horn',
    'n03417042': 'garbage truck',
    'n03425413': 'petrol_station',
    'n03445777': 'golf_ball',
    'n03888257': 'parachute'
}
```

After the initial preprocess, 9469 files were found belonging to 10 classes in the training data and 3925 files were found belonging to 10 classes in the testing data.

To further understand the scope of the data the images were plotted and were checked for any imbalance.

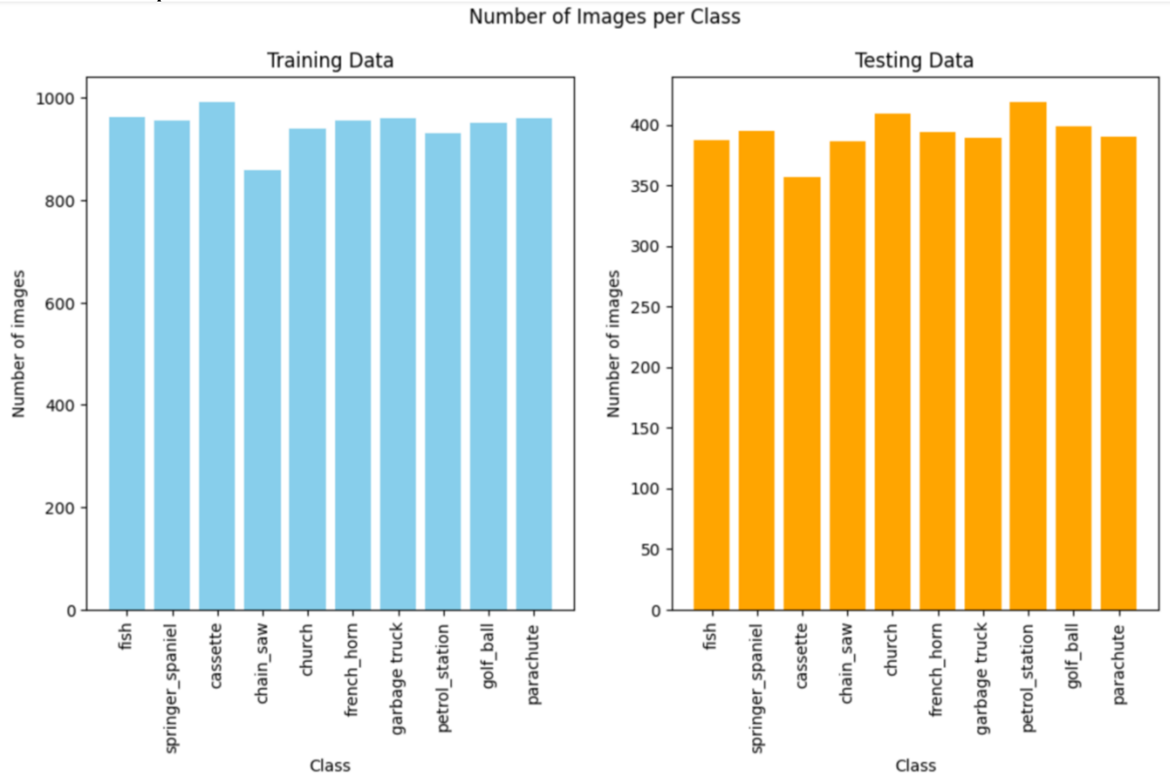




```python
# Shows how many training and testing images are in each class
train_counts = []
test_counts = []

for name in category_name:
  train_category = os.path.join(train_dir,name)
  test_category = os.path.join(test_dir,name)

  test_num = len(os.listdir(test_category))
  train_num = len(os.listdir(train_category))

  train_counts.append(train_num)
  test_counts.append(test_num)

  print(f"{name}: {train_num} training images, {test_num} testing images.")
```

```
cassette: 993 training images, 357 testing images.
chain_saw: 858 training images, 386 testing images.
church: 941 training images, 409 testing images.
fish: 963 training images, 387 testing images.
french_horn: 956 training images, 394 testing images.
garbage truck: 961 training images, 389 testing images.
golf_ball: 951 training images, 399 testing images.
parachute: 960 training images, 390 testing images.
petrol_station: 931 training images, 419 testing images.
springer_spaniel: 955 training images, 395 testing images.
```

This section of code shows the descriptive analysis of the image dataset, and the plot below is the visual representation.



The plots show an overall balanced data frame. This makes training the model easier as it guarantees that the models are trained fairly, avoiding biases. Comprehensively, lowering the likelihood of overfitting.

CNN was the model used for image classification because of its demonstrated effectiveness managing visual data. With their ability to recognise patterns and spatial hierarchies, CNNs are well suited for image data. Their main advantage is the ability to learn characteristics from pixel data without requiring manual feature extraction. Pooling layers, which record the hierarchical information and convolutional layers, which identify local patterns are used to accomplish this. CNNs feature fewer parameters than standard networks, resulting in less overfitting and better generalisation. Additionally, they scale well with big datasets, which makes them appropriate for image classification.

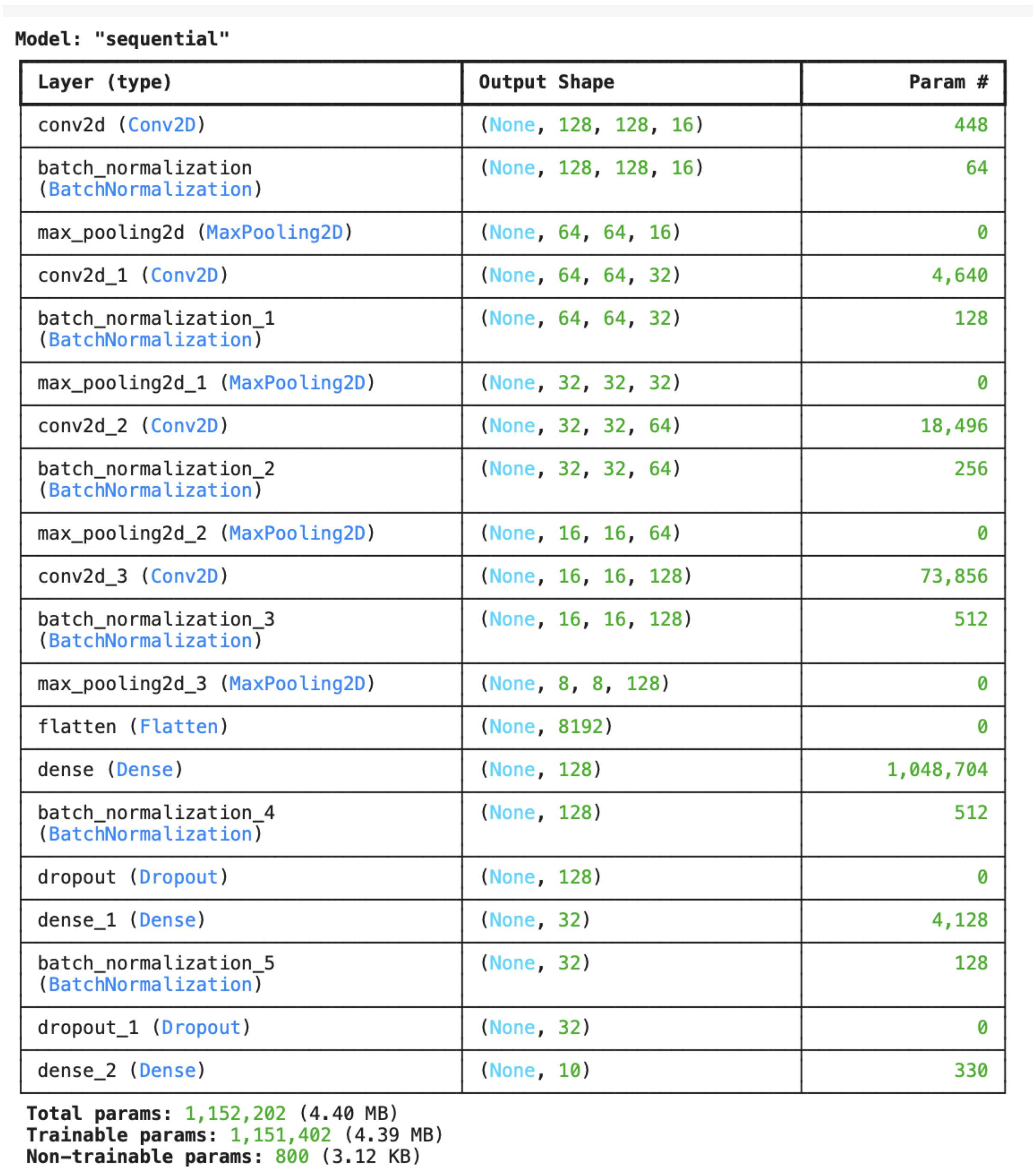
Due to CNNs local receptive fields the cost of training on images are reduced; they are computationally efficient. This effectiveness was essential when working with over 10,000 images where resource consumption and training time must be considered. Furthermore, advances in transfer learning allow CNNs to use pre-trained models, improving overall performance. Given their strengths, CNNs were the best fit for this task.

The Adam optimiser was chosen because of its adjustable learning rate. It uses gradient moments to modify each parameter. This results in faster convergence, particularly in models like CNNs where adjusting the learning rate is difficult. Essentially, Adam is effective at managing gradients, making it a good fit for image classification.

To evaluate performance I used recall, accuracy and the F1 score. Class imbalance is addressed by precision and recall, while accuracy gauges overall correctness. While recall the model’s capacity to recognise suitable instances, precision guarantees meaningful positive predictions. These indicators are balanced by the F1-score, which yields a reliable performance measure. To gain a better understanding of the model’s weakness, I also analysed false positives and false negatives using a confusion matrix. Lastly, tracking the loss curve allowed for the identification of problems like overfitting during the training process.

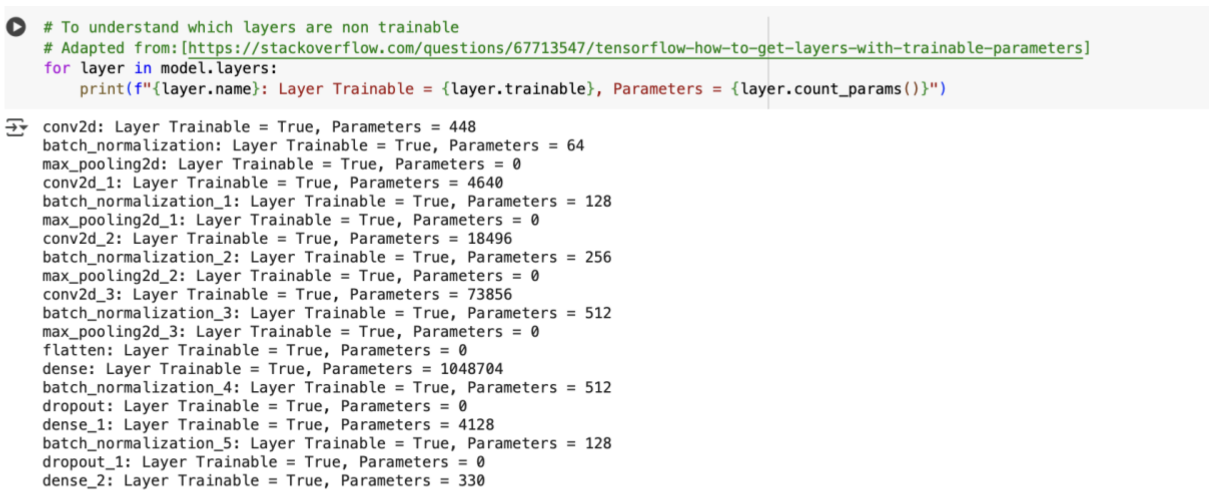
Experiments.

The figure below shows a summary table of the CNN model, breaking down the architecture.



```
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ conv2d (Conv2D)                      │ (None, 128, 128, 16)        │             448 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ batch_normalization                  │ (None, 128, 128, 16)        │              64 │
│ (BatchNormalization)                 │                             │                 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d (MaxPooling2D)         │ (None, 64, 64, 16)          │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_1 (Conv2D)                    │ (None, 64, 64, 32)          │           4,640 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ batch_normalization_1                │ (None, 64, 64, 32)          │             128 │
│ (BatchNormalization)                 │                             │                 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d_1 (MaxPooling2D)       │ (None, 32, 32, 32)          │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_2 (Conv2D)                    │ (None, 32, 32, 64)          │          18,496 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ batch_normalization_2                │ (None, 32, 32, 64)          │             256 │
│ (BatchNormalization)                 │                             │                 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d_2 (MaxPooling2D)       │ (None, 16, 16, 64)          │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_3 (Conv2D)                    │ (None, 16, 16, 128)         │          73,856 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ batch_normalization_3                │ (None, 16, 16, 128)         │             512 │
│ (BatchNormalization)                 │                             │                 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d_3 (MaxPooling2D)       │ (None, 8, 8, 128)           │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ flatten (Flatten)                    │ (None, 8192)                │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense (Dense)                        │ (None, 128)                 │       1,048,704 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ batch_normalization_4                │ (None, 128)                 │             512 │
│ (BatchNormalization)                 │                             │                 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dropout (Dropout)                    │ (None, 128)                 │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_1 (Dense)                      │ (None, 32)                  │           4,128 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ batch_normalization_5                │ (None, 32)                  │             128 │
│ (BatchNormalization)                 │                             │                 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dropout_1 (Dropout)                  │ (None, 32)                  │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_2 (Dense)                      │ (None, 10)                  │             330 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
 Total params: 1,152,202 (4.40 MB)
 Trainable params: 1,151,402 (4.39 MB)
 Non-trainable params: 800 (3.12 KB)
```

The neural network model has a sequential architecture that was built for image classification. This model uses several convolutional layers that extract features while lowering spatial dimensions followed by layers for max pooling and then batch normalisation. Starting with a Conv2d layer with 16 filters, the model progresses to deeper layers with 32, 64 and 128 as filters. Following three rounds of conv2d, batch normalisation and pooling these features are flattened to a vector size of 8192. In order to minimise overfitting, drop out layers are included. Out of the 1,152,202 parameters in the model, a total of 1,151,402 can be trained. Initially, there were over 1,000 untrainable parameters but this value was lowered after changing the filters around, starting with 16 instead of 32. With 800 non-trainable parameters, this suggests that some layers might be frozen (untrainable). To understand which layers were not trainable, the code below was used.

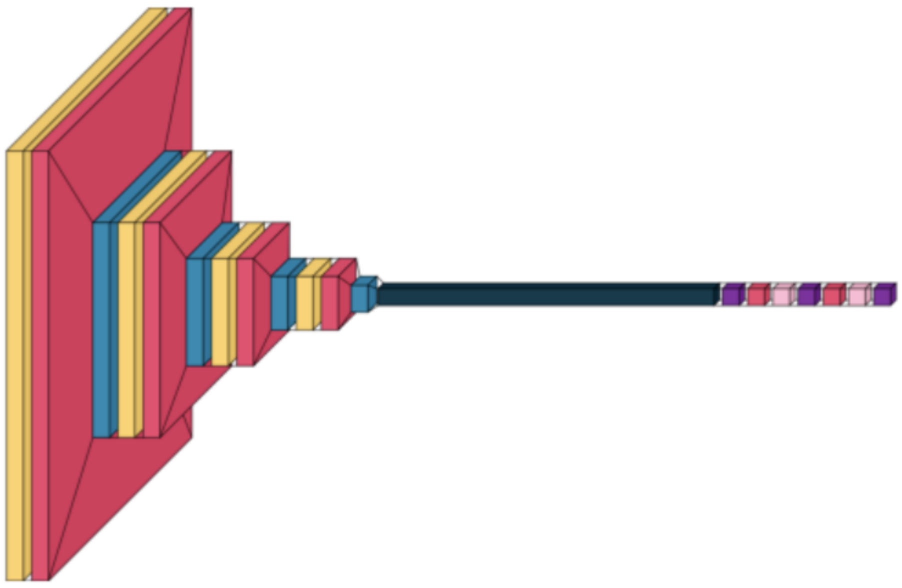


```python
# To understand which layers are non trainable
# Adapted from:[https://stackoverflow.com/questions/67713547/tensorflow-how-to-get-layers-with-trainable-parameters]
for layer in model.layers:
    print(f"{layer.name}: Layer Trainable = {layer.trainable}, Parameters = {layer.count_params()}")
```

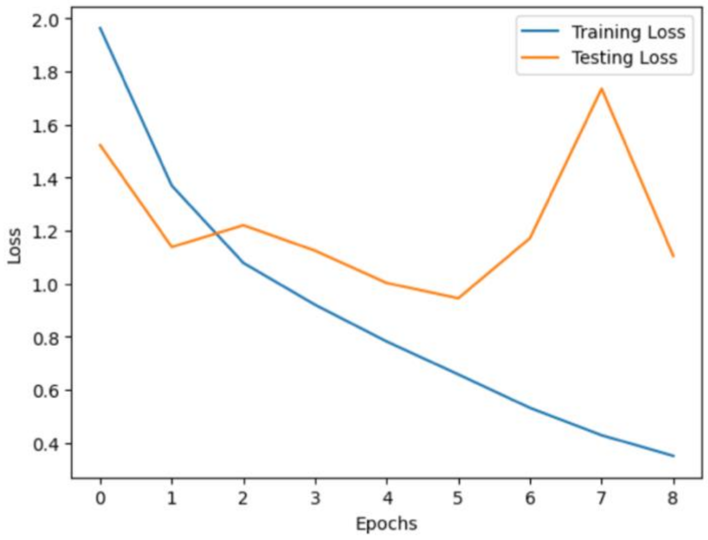
```
conv2d: Layer Trainable = True, Parameters = 448
batch_normalization: Layer Trainable = True, Parameters = 64
max_pooling2d: Layer Trainable = True, Parameters = 0
conv2d_1: Layer Trainable = True, Parameters = 4640
batch_normalization_1: Layer Trainable = True, Parameters = 128
max_pooling2d_1: Layer Trainable = True, Parameters = 0
conv2d_2: Layer Trainable = True, Parameters = 18496
batch_normalization_2: Layer Trainable = True, Parameters = 256
max_pooling2d_2: Layer Trainable = True, Parameters = 0
conv2d_3: Layer Trainable = True, Parameters = 73856
batch_normalization_3: Layer Trainable = True, Parameters = 512
max_pooling2d_3: Layer Trainable = True, Parameters = 0
flatten: Layer Trainable = True, Parameters = 0
dense: Layer Trainable = True, Parameters = 1048704
batch_normalization_4: Layer Trainable = True, Parameters = 512
dropout: Layer Trainable = True, Parameters = 0
dense_1: Layer Trainable = True, Parameters = 4128
batch_normalization_5: Layer Trainable = True, Parameters = 128
dropout_1: Layer Trainable = True, Parameters = 0
dense_2: Layer Trainable = True, Parameters = 330
```

From these results we can say all layers were trainable, this indicates that the non-trainable parameters have come from batch normalisation. During training, batch normalisation maintains the mean and variance moving averages that are used for inference. The moving averages are non-trainable since a running update method is used to update them rather than backpropagation.

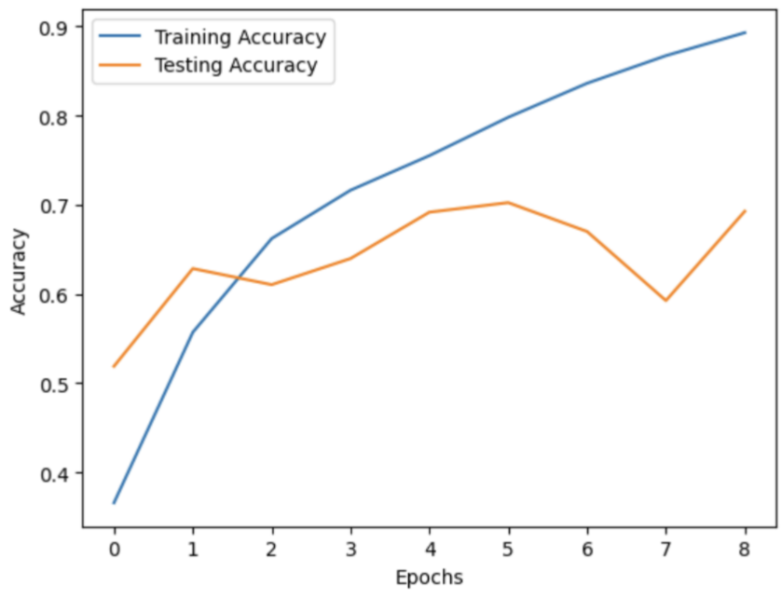
The figure below is a 3D representation of the CNN model. This model indicates potential overfitting. The input image is represented by the layers at the start and is then processed by convolutional layers that reduces spatial dimensions and extracts information. Through pooling procedures, the layers get deeper but narrower, suggesting that the filter depth is increasing while the spatial resolution is decreasing. The features are then flattened and run through layers that are fully connected in order to classify them. The CNN's middle layer (dark blue) consists of flattening and dense layers. This section demonstrates that the model has a large feature space before proceeding to the final output layer. Many neurons are used in these layers which is common when moving from feature extraction to classification. This design enables the network to learn complex correlations between the extracted features, but it can raise computational cost and increase the risk of overfitting if not regularised correctly.



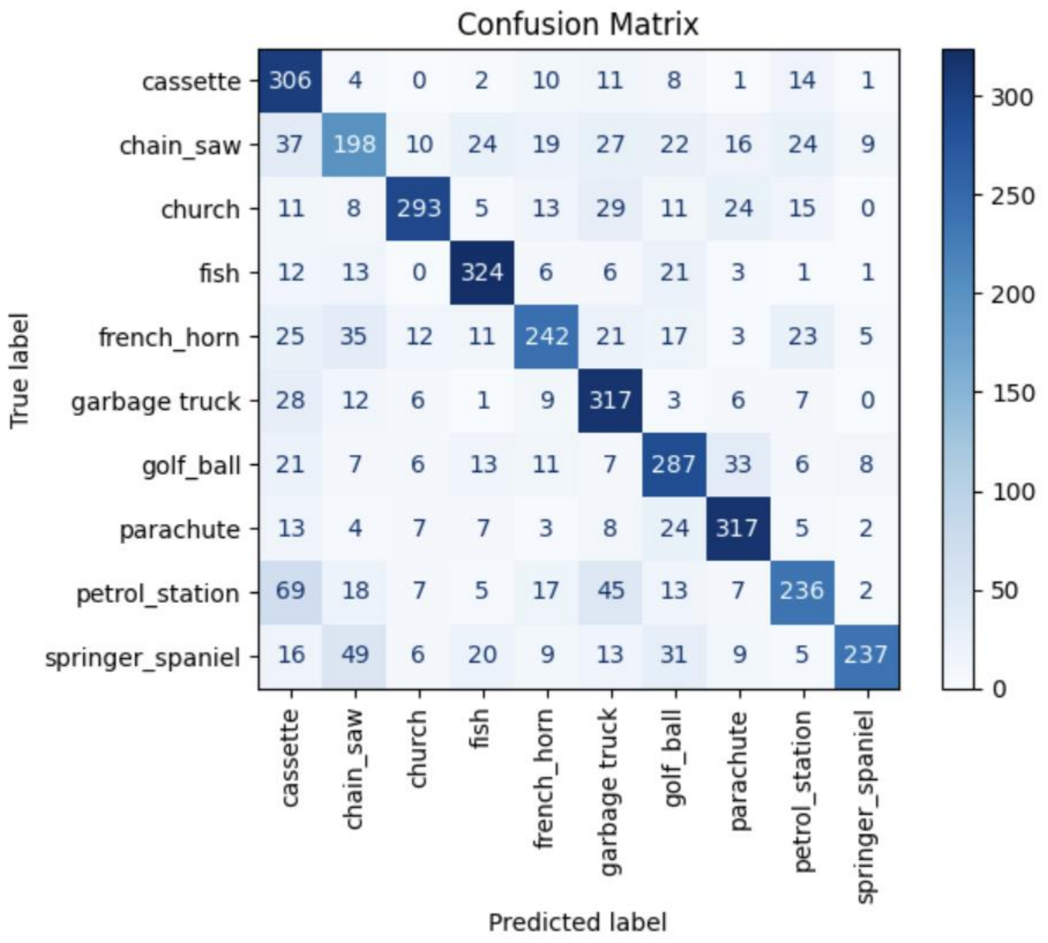
The line graph below shows how the model's loss varies over time throughout training and validation. The training loss line decreases suggesting that the model is learning the patterns from the data. The model is memorising the training data but is not generalising, as seen by the line for testing loss, which indicates significant overfitting.



The line graph below shows the model's accuracy across training and validation. As the line continuously rises, the model is learning patterns from the training set. Training accuracy is high, however the gap between the training set and testing set increases, indicating significant overfitting.



The figure below shows a confusion matrix. It highlights that the best performing class is fish, accurately classifying 324 images out of 387 fish images. The worst performing classification is the chainsaw. Overall, the matrix shows that the model is performing well as it has a strong diagonal. Although some classes have significant misclassifications such as chainsaw and petrol station.



Confusion Matrix

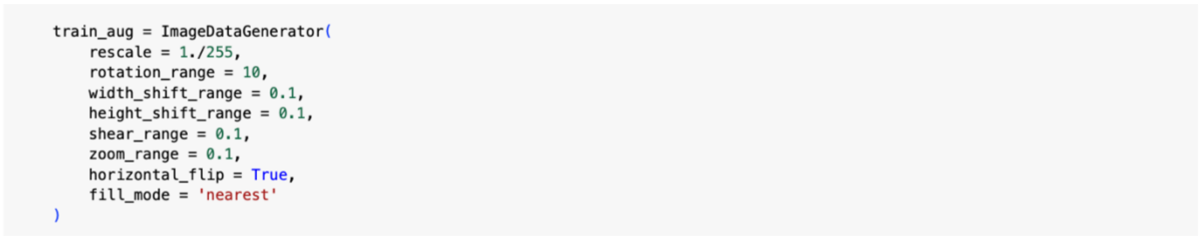
True label \ Predicted label	cassette	chain_saw	church	fish	french_horn	garbage truck	golf_ball	parachute	petrol_station	springer_spaniel
cassette	306	4	0	2	10	11	8	1	14	1
chain_saw	37	198	10	24	19	27	22	16	24	9
church	11	8	293	5	13	29	11	24	15	0
fish	12	13	0	324	6	6	21	3	1	1
french_horn	25	35	12	11	242	21	17	3	23	5
garbage truck	28	12	6	1	9	317	3	6	7	0
golf_ball	21	7	6	13	11	7	287	33	6	8
parachute	13	4	7	7	3	8	24	317	5	2
petrol_station	69	18	7	5	17	45	13	7	236	2
springer_spaniel	16	49	6	20	9	13	31	9	5	237

The figure below shows a classification report. Fish has the highest f1 score (0.81) indicating a high level of recall and precision in its classification. Parachute and church also show a strong performance with an f1 score of 0.78. With a precision score of 0.89, springer spaniel has the highest. This means its predicted label is usually correct. The cassette class is the best at minimising false negatives, having a recall score of 0.86. Cassette and chain saw have the lowest precision score, 0.57. Petrol station has the lowest recall score, 0.56. This means the class has a high false negative rate.

The model overall has an accuracy score of 70.2% showing a good overall performance. As the data is balanced, the macro averages will be used. The model predicted classes 71% (precision) accurately. Since precision and recall values are nearly equal, the model does not under or over predict a given class.

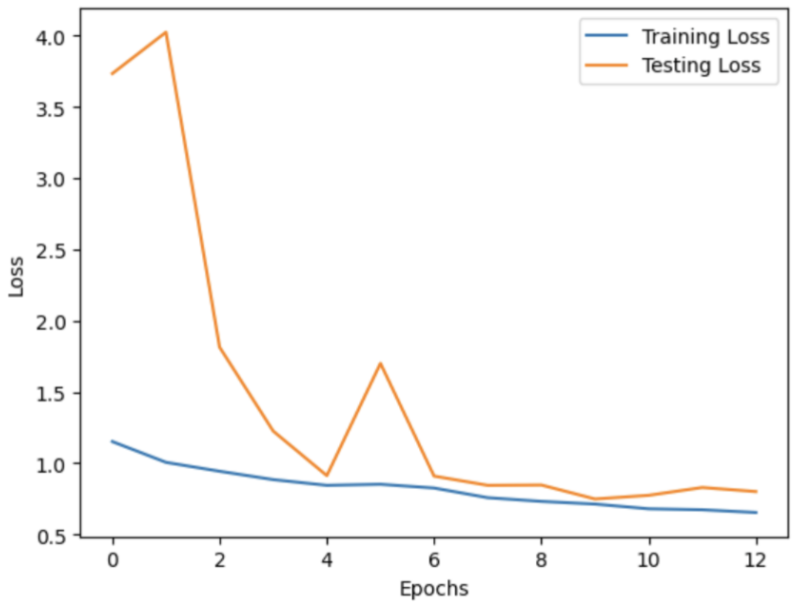
```
Accuracy: 0.7024203821656051
                  precision    recall  f1-score   support

        cassette       0.57      0.86      0.68       357
       chain_saw       0.57      0.51      0.54       386
          church       0.84      0.72      0.78       409
            fish       0.79      0.84      0.81       387
     french_horn       0.71      0.61      0.66       394
   garbage truck       0.65      0.81      0.73       389
       golf_ball       0.66      0.72      0.69       399
       parachute       0.76      0.81      0.78       390
  petrol_station       0.70      0.56      0.63       419
springer_spaniel       0.89      0.60      0.72       395

        accuracy                           0.70      3925
       macro avg       0.71      0.70      0.70      3925
    weighted avg       0.72      0.70      0.70      3925
```

To see if there could be any improvements in accuracy of the model, I added data augmentation. These parameters allow the model to train on different variations of the images, making it more accurate to real life.

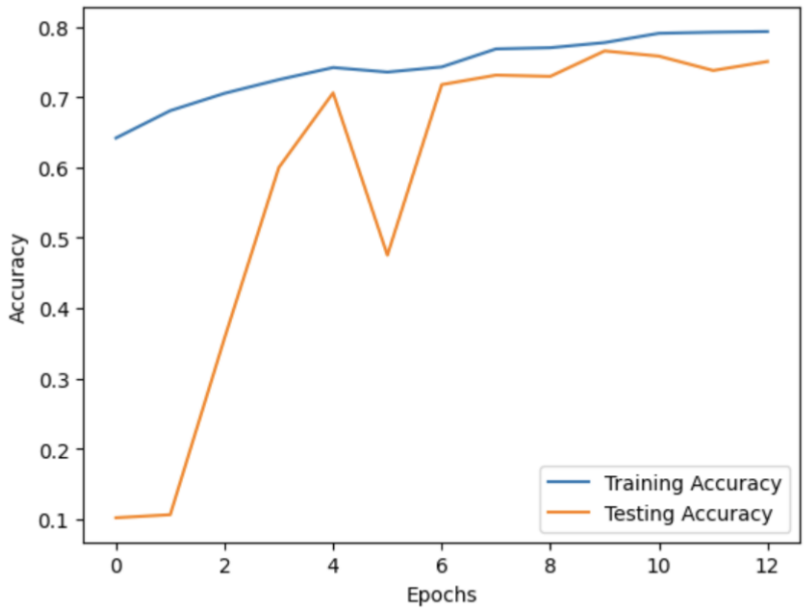


```
train_aug = ImageDataGenerator(
    rescale = 1./255,
    rotation_range = 10,
    width_shift_range = 0.1,
    height_shift_range = 0.1,
    shear_range = 0.1,
    zoom_range = 0.1,
    horizontal_flip = True,
    fill_mode = 'nearest'
)
```

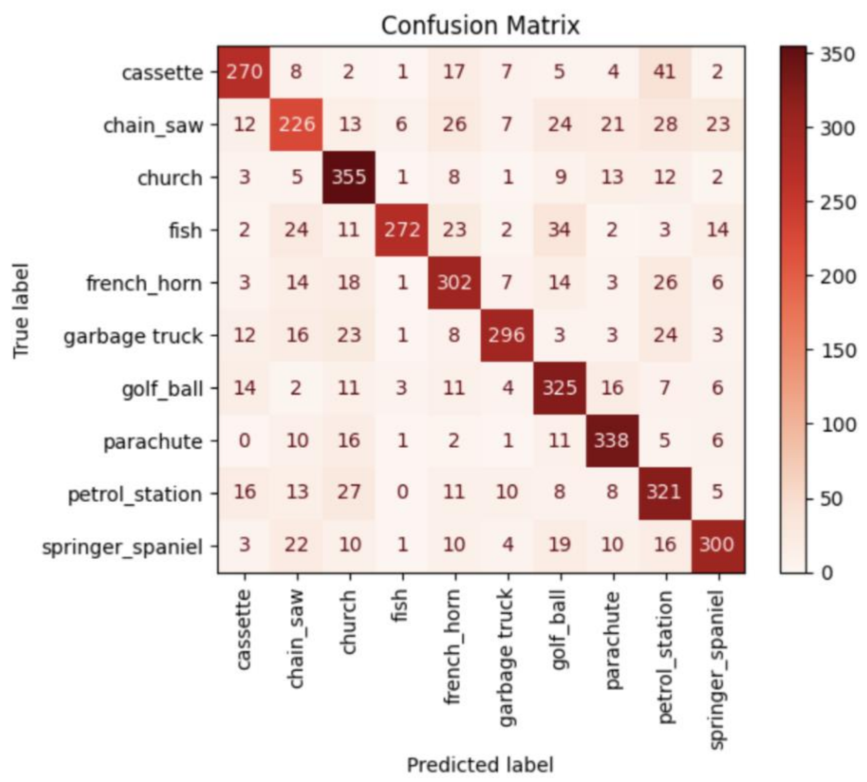
The line graph below represents the loss. The training loss with data augmentation starts with lower values than the model without suggesting that this model is better at generalising and the data augmentation prevents the model from overfitting acting as a control. The testing data initial has a large gap with the first few epochs but after a while, this gap decreases and is much closer to the training data suggesting less overfitting with data augmentation than without.



This line graph following the accuracy shows a similar pattern as the loss graph. The testing accuracy significantly underfits for the first few runs but this gap decreases and follows the training accuracy closer. The testing accuracy shows a slight underfit. These graphs show that the model with the data augmentation process added preforms better.

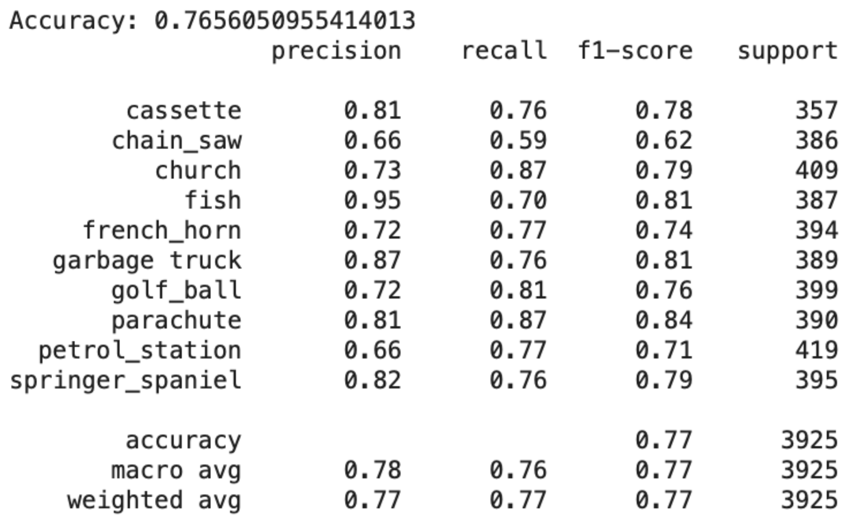


The confusion matrix below shows a better outcome with data augmentation. This time the church appears to have the most correctly identified class. The chain saw is still the class that is the least accurate even with data augmentation.



	cassette	chain_saw	church	fish	french_horn	garbage truck	golf_ball	parachute	petrol_station	springer_spaniel
cassette	270	8	2	1	17	7	5	4	41	2
chain_saw	12	226	13	6	26	7	24	21	28	23
church	3	5	355	1	8	1	9	13	12	2
fish	2	24	11	272	23	2	34	2	3	14
french_horn	3	14	18	1	302	7	14	3	26	6
garbage truck	12	16	23	1	8	296	3	3	24	3
golf_ball	14	2	11	3	11	4	325	16	7	6
parachute	0	10	16	1	2	1	11	338	5	6
petrol_station	16	13	27	0	11	10	8	8	321	5
springer_spaniel	3	22	10	1	10	4	19	10	16	300

The figure below shows the classification report of the model with data augmentation. This model has 76% accuracy. Parachute has the highest f1 score, 0.84. With data augmentation, parachute has a higher f1 score than fish (0.81) differing from the model without data augmentation. Although fish has a precision score of 0.95, indicating that the model is good at predicting a positive class. The classes with the lowest precision are the chain saw and petrol station, similar to the model without data augmentation. These classes struggling to be identified by the model indicates that the features are not distinct enough or the data is noisy. The model's overall precision is 78%, recall is 76% and f1 score is 77%. This shows that the model is balanced.



```
Accuracy: 0.7656050955414013
                  precision    recall  f1-score   support

        cassette       0.81      0.76      0.78       357
       chain_saw       0.66      0.59      0.62       386
          church       0.73      0.87      0.79       409
            fish       0.95      0.70      0.81       387
     french_horn       0.72      0.77      0.74       394
   garbage truck       0.87      0.76      0.81       389
        golf_ball      0.72      0.81      0.76       399
       parachute       0.81      0.87      0.84       390
  petrol_station       0.66      0.77      0.71       419
springer_spaniel       0.82      0.76      0.79       395

        accuracy                           0.77      3925
       macro avg       0.78      0.76      0.77      3925
    weighted avg       0.77      0.77      0.77      3925
```

Reflection.

CNNs are extremely successful in image classification because they can learn the spatial hierarchies of the image's features. In many, different industries they are important for increasing productivity and accuracy.

In conclusion, the model with data augmentation performed better. The model without data augmentation had an accuracy of 70.2% and with had an accuracy of 76.6%. Both models performed relatively well, although there were some cases of overfitting and underfitting. These issues could be solved by adding further parameters for data augmentation like colour or changing the model architecture. Another way I could improve the model's performance is to use pre-trained models as the generalise better due to being trained on diverse datasets. In real life applications, this model would need to be fined tune to have a higher accuracy as the 30% could result in high risks when image classification is used for medical diagnosis.

Task 2 – Reinforcement Learning.

Introduction.

Within machine learning, reinforcement learning is the process by which an agent makes decisions through interaction with its surroundings. A Gridworld environment is used to find the optimal policy for the agent (the mouse) to reach the goal (a block of cheese) as effectively as possible while optimising benefits. Value iteration and Q-learning are the two reinforcement learning techniques used.

Value iteration is a programming technique that uses the Bellman equation to update state values.

Bellman's equation:

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s'|s,a) V(s') \right)$$

This formula represents a state **s** as the highest possible expected return on any given action **a**, where **R** (**s**, **a**) is the instant reward of taking action **a** in state **s** and **γ** represents the discount factor. **P (s'|s, a)** the probability of moving from one state **s** to another **s'** after doing a specific action **a**. We can choose the optimum course of action by looking at the value of the following state.

Unlike value iteration, Q-learning is a model free algorithm. Rather than using knowledge of the environment, the agent learns the ideal action value function by trial and error.

Q-learning equation:

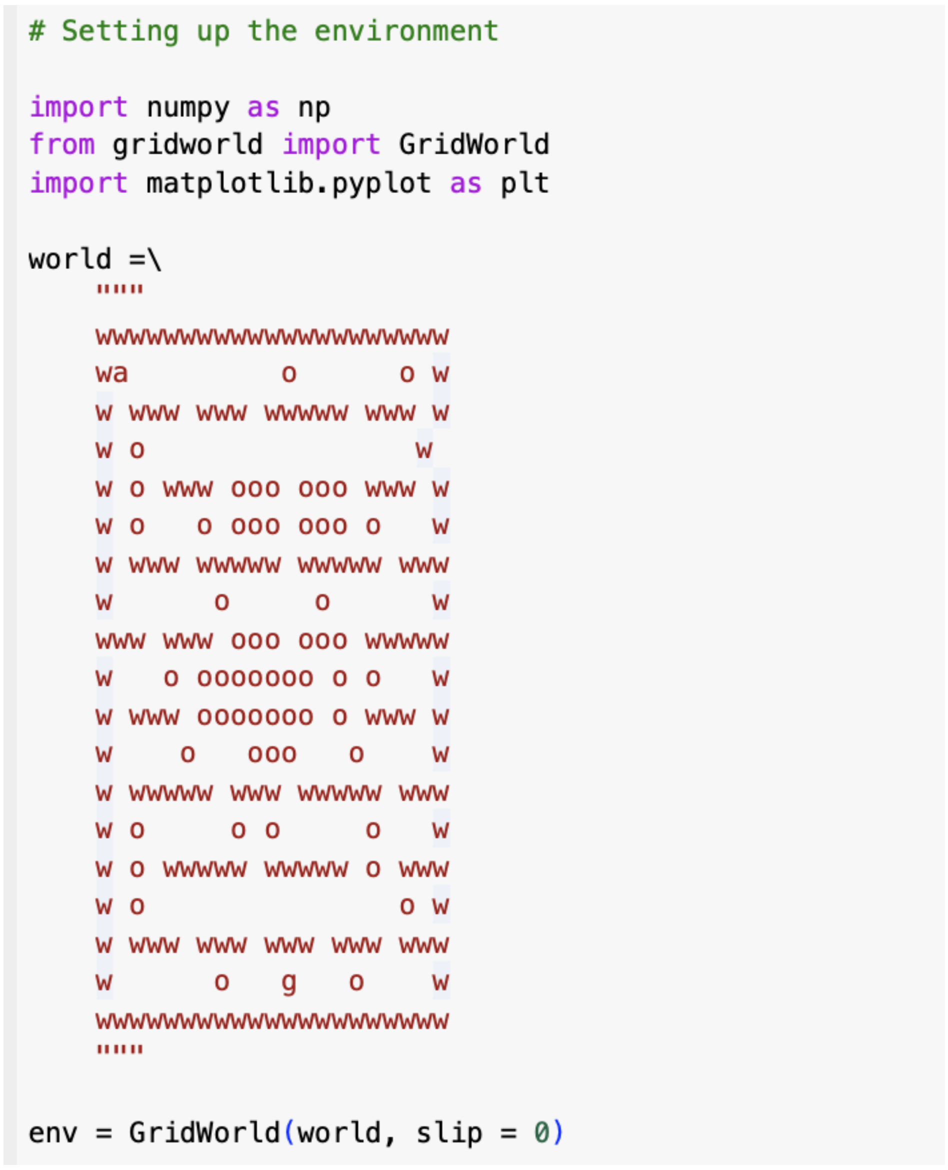
$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

This formula changes the predicted future reward (the Q-value) for a state action pair by considering the immediate reward as well as the best probable future reward from the next state. It modifies the existing Q-value by adding a fraction of the difference between the new and old estimates, using a learning rate to determine how much the update should affect the value.

Both techniques are assessed on their convergence rate, learning behaviour and final policy. The impact of various discount factors and exploration rates on Q-learning's overall performance is also explored. To comprehend the advantages and disadvantages of each method for handling Gridworld situations, they will be compared.

Methods.

To find out which method offers the best policy and how they differ in decision making, it was important to establish the environment. The figure below shows the code for setting up the environment.



```
# Setting up the environment

import numpy as np
from gridworld import GridWorld
import matplotlib.pyplot as plt

world =\
    """

    wwwwwwwwwwwwwwwwwwwww
    wa         o      o w
    w www www wwwww www w
    w o                w
    w o www ooo ooo www w
    w o   o ooo ooo o   w
    w www wwwww wwwww www
    w      o     o      w
    www www ooo ooo wwwww
    w   o ooooooo o o   w
    w www ooooooo o www w
    w    o   ooo   o    w
    w wwwww www wwwww www
    w o     o o     o   w
    w o wwwww wwwww o www
    w o               o w
    w www www www www www
    w      o   g   o    w
    wwwwwwwwwwwwwwwwwwwww
    """

env = GridWorld(world, slip = 0)
```

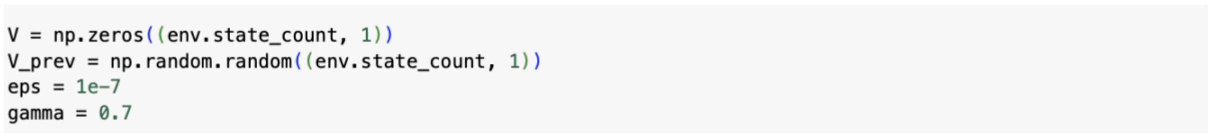
Three main libraries were imported for both value iteration and q-learning (an additional library was imported specifically for q-learning). Those libraries included import numpy as np, from gridworld import import Gridworld and import matplotlib.pyplot as plt.

NumPy is often used in reinforcement learning as it can store and process the state and reward matrices and is know for executing operations like calculating state transitions. While Gridworld is the environment where the agent operates. This library allows for the agent to move based on the actions and also displays the grid visually. Lastly, matplotlib is used to visually display the environment.

The grid environment is defined by different values, where **w** represents the wall (obstacles) , **a** represents the agent's starting position, **o** represents spaces where the agent can freely move, and **g** represents the goal. The agent's goal is to navigate to **g** while avoiding the obstacles and to find the most effective path.

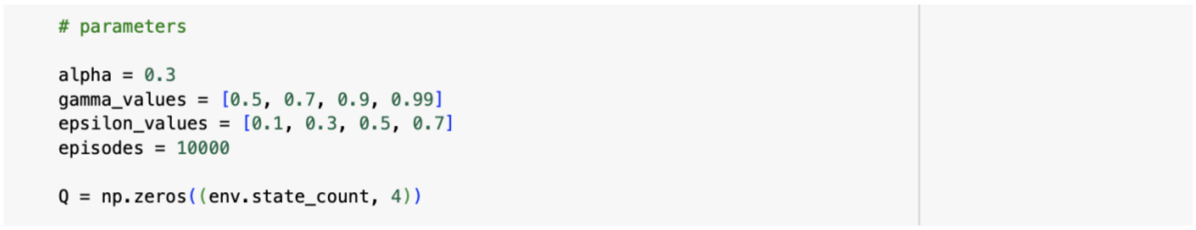
env = GridWorld(world, slip = 0) is used to set the environment as deterministic. Slip = 0 means that every action taken by the agent results in expected movements. By doing this, it is ensured that the agent can learn the best course of action without being interrupted by unpredictability.

As mentioned earlier the two methods being used are Value Iteration and Q-learning. In value iteration **V(s)** is initialised as np.zeros((env.state_count, 1)), this makes the states equal, starting the agent with no bias. Additionally, by using the Bellman's equation, this enables rewards to raise the values of states while preserving convergence stability by propagating backwards by iteration. This parameter can be seen in the figure below.



```
V = np.zeros((env.state_count, 1))
V_prev = np.random.random((env.state_count, 1))
eps = 1e-7
gamma = 0.7
```

The other important parameters of value iteration are the discount factor and the convergence threshold. The discount factor is set to gamma = 0.7. This determines how much future rewards will influence a state's current value. A greater discount factor (>0.5) promotes long term planning by placing a higher value on future rewards. A lower discount factor (<0.5) prioritises benefits immediately. A discount of 0.7 guarantees that the agent will weigh future rewards equally with immediate rewards, while considering short term rewards.



```
# parameters

alpha = 0.3
gamma_values = [0.5, 0.7, 0.9, 0.99]
epsilon_values = [0.1, 0.3, 0.5, 0.7]
episodes = 10000

Q = np.zeros((env.state_count, 4))
```

The figure above shows the parameters used for Q-learning. Unlike Value iteration where **V(s)** is initialised as zero, the Q-values are initialised as 4, following an optimistic assumption. This is when the values are set higher than the expected rewards. Since Q-learning updates depend on received incentives, the high values will move the agent to explore further in the beginning as it attempts to validate whether those values are correct. The starting values will be adjusted over time as the agent explores the environment and gains rewards. Alpha represents the learning rate and is used to calculate how much of the new information supersedes prior knowledge. The Q-values are updated by the agent moderately as the value of alpha is 0.3. This value allows the new information to be balanced with the prior knowledge. The gamma values also known as the discount factors, vary from 0.5 to 0.99, this is to compare the different discount factors and see which value presents the best policy. The epsilon values represent the exploration rates and vary from 0.1 to 0.7. The exploration rate will determine how frequent the agent will investigate new actions rather than using known actions. Higher exploration rates indicate more random exploration, where lower exploration rates favour exploitation of learnt policies. The number of episodes represents how often the agent is trained to find the optimal policy.

Experiments.

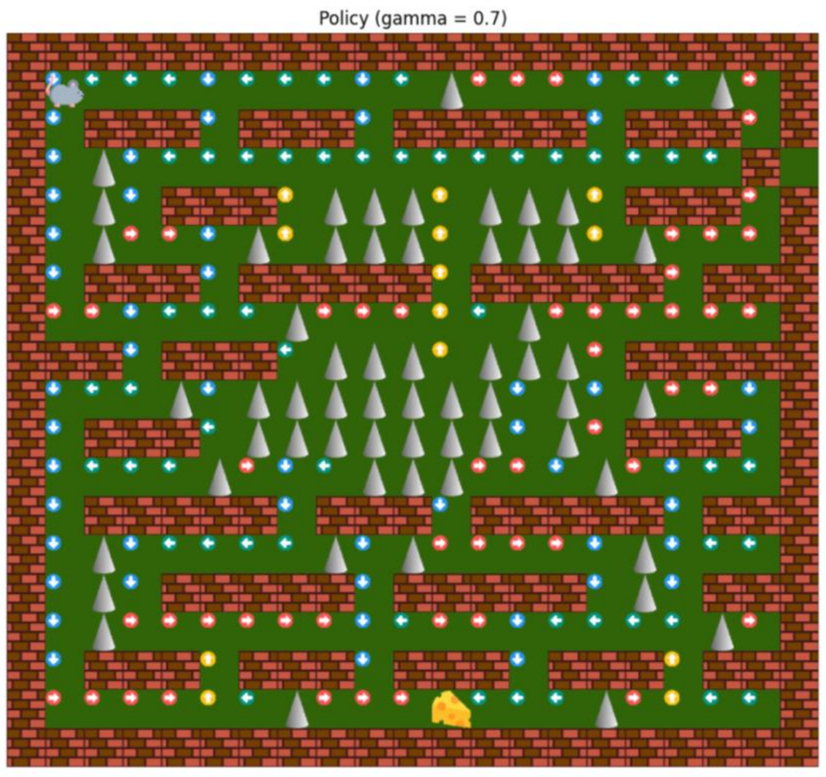
The mapping for positions is:

0 = right
1 = down
2 = left
3 = up

For Value Iteration, the Optimal Policy is as follows:

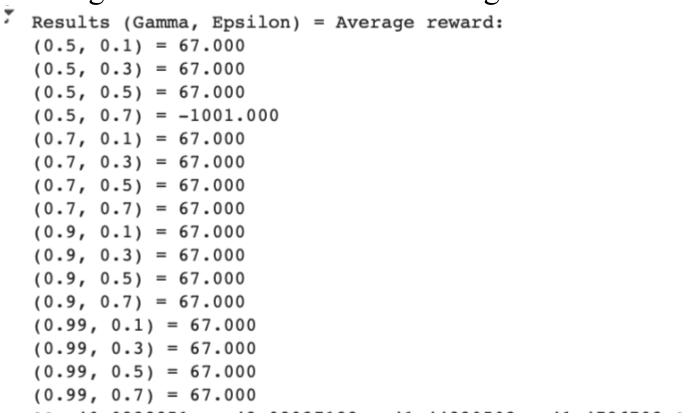
```
Opitimal policy: 0.7 = [1 2 2 2 1 2 2 2 1 2 2 0 0 0 1 2 2 2 0 1 1 1 1 0 1 2 1 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 1 2 1 3 2 3 3 3 2 3 3 3 0 1 2 0 0 1 2 3 2 2 0 3 2 2 0 3 2 0 0 0
 1 1 3 0 0 0 1 2 2 2 2 0 0 0 3 2 2 0 0 0 0 0 0 1 2 3 3 0 3 2 1 0 0 1 2 2 2
 1 2 3 1 3 0 3 0 1 0 1 0 0 0 1 1 2 1 1 1 1 0 1 0 1 0 0 1 1 2 2 2 0 0 1 2 2
 0 1 0 0 1 0 0 1 2 2 1 1 1 1 1 0 1 2 2 2 2 0 1 2 0 0 0 0 1 2 1 2 2 1 0 1 1
 1 1 1 1 0 0 0 0 0 0 0 1 2 0 0 1 2 2 2 2 2 0 1 3 1 1 3 0 0 0 0 3 2 0 0 0 0
 1 2 2 2 2 0 3 2 2]
```

For state 0 the results show that the optimal policy is 1, which would be the action right and for sate 1 the optimal policy is 2, making the action left. This continues until the end of the array. To summarise, every number shows a state in the environment and the action that will maximise the agent's long-term reward. The figure below shows a visual representation of the actions of the agent. The red arrows represent areas that will have a negative outcome for the agent. The blue arrows show areas that are considered neutral, having no immediate consequences like a reward or penalty. The yellow arrows warn the agent of paths that have a higher risk. Finally, the green arrows indicate a positive outcome often leading to the rewards.



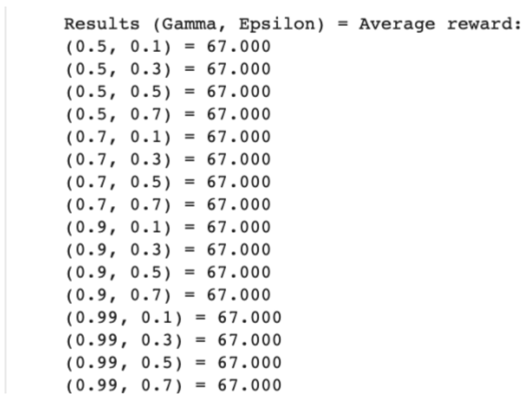
For Q-learning, I investigated how different discount factors and exploration rates impact the final performance. This was done by running through the values and comparing the average reward over 100 episodes to gain an understanding of early performance, 500 episodes to get a representation of what is happening halfway through the training and 1000 episodes to get the overall performance.

This figure shows the results of average reward for 100 episodes.



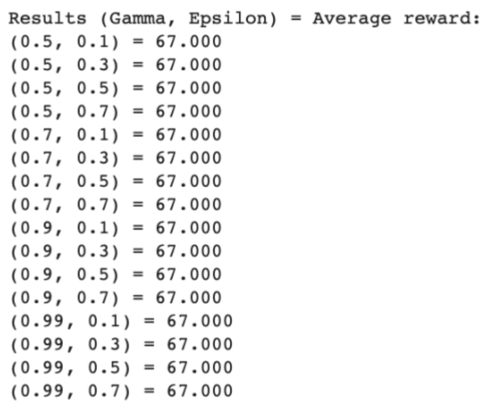
```
Results (Gamma, Epsilon) = Average reward:
(0.5, 0.1) = 67.000
(0.5, 0.3) = 67.000
(0.5, 0.5) = 67.000
(0.5, 0.7) = -1001.000
(0.7, 0.1) = 67.000
(0.7, 0.3) = 67.000
(0.7, 0.5) = 67.000
(0.7, 0.7) = 67.000
(0.9, 0.1) = 67.000
(0.9, 0.3) = 67.000
(0.9, 0.5) = 67.000
(0.9, 0.7) = 67.000
(0.99, 0.1) = 67.000
(0.99, 0.3) = 67.000
(0.99, 0.5) = 67.000
(0.99, 0.7) = 67.000
```

This figure shows the results of the average reward for 500 episodes.



```
Results (Gamma, Epsilon) = Average reward:
(0.5, 0.1) = 67.000
(0.5, 0.3) = 67.000
(0.5, 0.5) = 67.000
(0.5, 0.7) = 67.000
(0.7, 0.1) = 67.000
(0.7, 0.3) = 67.000
(0.7, 0.5) = 67.000
(0.7, 0.7) = 67.000
(0.9, 0.1) = 67.000
(0.9, 0.3) = 67.000
(0.9, 0.5) = 67.000
(0.9, 0.7) = 67.000
(0.99, 0.1) = 67.000
(0.99, 0.3) = 67.000
(0.99, 0.5) = 67.000
(0.99, 0.7) = 67.000
```

This figure shows the results of the average reward for 1000 episodes.



```
Results (Gamma, Epsilon) = Average reward:
(0.5, 0.1) = 67.000
(0.5, 0.3) = 67.000
(0.5, 0.5) = 67.000
(0.5, 0.7) = 67.000
(0.7, 0.1) = 67.000
(0.7, 0.3) = 67.000
(0.7, 0.5) = 67.000
(0.7, 0.7) = 67.000
(0.9, 0.1) = 67.000
(0.9, 0.3) = 67.000
(0.9, 0.5) = 67.000
(0.9, 0.7) = 67.000
(0.99, 0.1) = 67.000
(0.99, 0.3) = 67.000
(0.99, 0.5) = 67.000
(0.99, 0.7) = 67.000
```

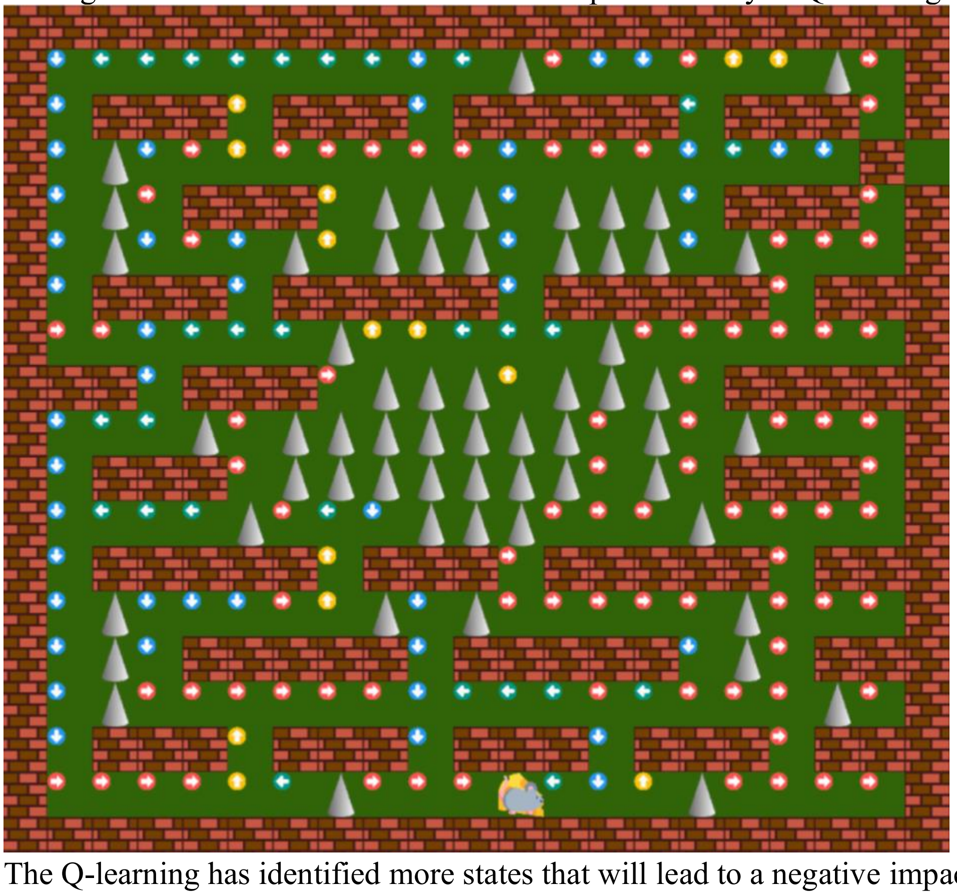
When comparing the results between 100, 500 and 1000 episodes, it is evident that the agent has potential hit a learning plateau. As from the first 100 episodes, most of the results have already reached 67% and by midway all have achieved 67%. This indicates that the agent (the mouse) has learnt a relatively optimal policy but has stop improving. As it is no longer improving in value this also suggests that the environment had been fully learned by the agent. Other reasons could be the complexity of the environment.

The Optimal Policy is as follows:

```
Pi: [1 2 2 2 2 2 2 2 1 2 0 0 1 1 0 3 3 0 0 1 3 1 2 0 1 0 1 0 3 0 0 0 0 0 1 0 0
 0 1 2 1 1 1 0 0 3 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0 3 0 0 0 1 0 0 0 1 0 0 0 0
 1 1 1 0 0 0 1 2 2 2 0 3 3 2 2 2 0 0 0 0 0 0 0 1 0 0 0 0 3 0 0 0 0 1 2 2 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 2 2 2 0 0 2 1 0
 0 0 0 0 0 0 0 0 0 0 1 3 0 0 1 0 1 1 1 0 3 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 1
 1 0 0 1 0 0 0 0 0 0 0 1 2 2 2 0 2 0 0 0 0 0 1 3 1 1 0 0 0 0 0 3 2 0 0 0 0
 0 2 1 3 0 0 0 0 0]
```

There are some shared actions between Value Iteration and Q-learning, the first 4 states share the same actions, this suggest that q-learning is converging to similar strategies.

The figure below shows a screenshot of the Optimal Policy in Q-learning.



The Q-learning has identified more states that will lead to a negative impact compared to the Value Iteration. This is because of the different type of algorithm the two methods have. Q-learning learns through interacting with the environment by trial and error following an exploration algorithm. The agent may initially believe an action is advantageous in these danger zones (represented by the red arrows) but after investigating, the agent will find that there are negative consequences. In contrast, Value Iteration is based on models and assumes that the full environment is understood. It will determine the ideal value function for every reward and state resulting in a policy that avoids the danger zones.

Reflection.

To conclude, when the dynamics of the environment were known, Value Iteration offered a deterministic and effective solution. In contrast, Q-learning was effective in unfamiliar situations, improving with each interaction. The decision between the two, is determined by whether prior information of the environment exists and how adaptable the learning process must be. Repeating the experiment with a reward and penalty map, could enhance the outcomes, particularly for Q-learning. For example, if the agent makes a positive decision +10 points will be rewarded and if the agent made a negative decision -5 points will be deducted. Unnecessary moves could also be penalised to ensure the agent finds the optimal policy.