Data Augmentation for Object Detection using Deep Reinforcement Learning

Axel Andersson

Nils Hallerfelt

Department of Automatic Control

MSc Thesis
TFRT-6225
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Abstract

Data augmentation is a concept which is used to improve machine learning models for computer vision tasks. It is usually done by firstly, defining a set of functions which transforms images and secondly, applying a random selection of these functions on the images. Since the quality of training data is one of the, if not the most important factor to obtain a good model, this master thesis poses the question whether an intelligent deep reinforcement learning (DRL) agent can select augmentation functions in a better way. More specifically, can the agent select augmentations such that the performance of an object detection model increases? Besides improving the performance of an object detection model, the DRL agent provides insights in what constitutes good data augmentation. The project results in an agent which augments images such that mean average precision (mAP50) increases with 2.3% compared to a baseline detector, trained with random augmentations. This is a promising result that encourages further research on this area. To our knowledge, this is the first time a deep reinforcement learning agent has been used to improve an object detection model via better data augmentation.

Key words: data augmentation, deep reinforcement learning, machine learning, computer vision, object detection

Acknowledgements

Firstly, we would like to sincerely thank our supervisors, Robin Göransson, Joel Sjöbom and Albin Heimerson for supporting us throughout the entire process of this thesis. Our weekly meetings steered us in the right direction and your feedback and insights have always been valuable to us.

We would also like to thank everyone at the Core Technologies Analytics department at Axis Communications for continuous feedback, interest in the project, helping us understand the code base and letting us use your resources to conduct our experiments. We couldn't have asked for a better environment to do this master thesis.

Contents

List of Abbreviations

AI	**A**rtificial **I**ntelligence
AP	**A**verage **P**recision
CNN	**C**onvolutional **N**eural **N**etwork
DP	**D**ynamic **P**rogramming
DRL	**D**eep **R**einforcement **L**earning
DQN	**D**eep **Q**-**N**etwork
DDQN	**D**ouble **D**eep **Q**-**N**etwork
FC	**F**ully **C**onnected
FFN	**F**eed **F**orward **N**etwork
GAN	**G**enerative **A**dversarial **N**etwork
GPI	**G**eneralized **P**olicy **I**mprovement
GPU	**G**raphics **P**rocessing **U**nit
IoU	**I**ntersection **o**ver **U**nion
MAE	**M**ean **A**bsolute **E**rror
MAP	**M**ean **A**verage **P**recision
MC	**M**onte **C**arlo
MDP	**M**arkov **D**ecision **P**rocess
ML	**M**achine **L**earning
MSE	**M**ean **S**quared **E**rror
POMDP	**P**artially **O**bservable **M**arkov **D**ecision **P**rocess
RL	**R**einforcement **L**earning
RNN	**R**ecurrent **N**eural **N**etwork
(S)GD	(**S**tochastic) **G**radient **D**escent
TD	**T**emporal **D**ifference

1 Introduction

In recent years, the field of computer vision has witnessed remarkable advancements in a variety of tasks such as classification, segmentation and object detection. The demand for robust and accurate object detection models continues to grow since it is a key component in various applications like video surveillance, medical imaging and autonomous vehicles. Due to the lucrative applications, researchers are interested in enhancing the performance of these systems. The quality of a model is heavily reliant on the quality of training data, a common expression in machine learning is "garbage in, garbage out". Furthermore, a model, in particular a deep learning model, needs vast amounts of data to learn meaningful tasks. Perfect, labeled data can be hard to find or expensive to create more of. In areas such as video surveillance or medical imaging, training data can contain sensitive information which prohibits researchers from licensing datasets as open source, making the data publicly available. A common strategy to improve models when data is scarce is to augment data. Each training point is slightly modified according to some function which provides new data to train models on.

An important question is how these augmentations are picked. One promising avenue to improve object detection systems is to leverage techniques from deep reinforcement learning. This has shown impressive results in other areas of machine learning such as natural language processing. An example of this is *Reinforcement Learning with Human Feedback* which is a technique used by OpenAI during training of models used in ChatGPT [Ouyang et al., 2022].

This thesis addresses the challenge of optimizing the performance of object detection models by incorporating a deep reinforcement learning agent. The primary focus is to empower the model by providing it with a dataset which has been dynamically augmented with pre-defined augmentations selected by an intelligent agent. This intelligent augmentation strategy aims to overcome the limitations associated with static augmentation pipelines. Some augmentations might not contribute at all, and there might be augmentations which suits better to certain images compared to others. Static augmentation pipelines does not consider these issues.

In this initial chapter, the thesis will be motivated, related work will be presented and research questions are introduced. The chapter will also cover limitations, ethical considerations and present the outline of the thesis.

1.1 Motivation

The motivation for this research stems from the realization that the effectiveness of object detection models should be coupled with the quality and relevance of the data augmentations applied to the training dataset. Traditional methods rely on fixed augmentation pipelines, neglecting the dynamic nature of diverse datasets. As a result, models may not generalize across different scenarios as well as it could. Poor generalization leads to sub-optimal performance in real-world applications.

By leveraging the power of deep reinforcement learning, we seek to improve an object detection model with intelligently selected data augmentations. This approach is motivated by the need to enhance model adaptability, ultimately enabling it to better capture and understand intricate patterns in various image contexts. In particular when data is scarce. The intelligent augmentation strategy aims to optimize feature learning, address overfitting concerns and improve the model's robustness across various domains, thereby improving the overall model capabilities.

In summary, the research seeks to understand the demand of a dynamic augmentation strategy in real-world datasets. This will be done by integrating a deep reinforcement learning agent during training of an object detection model. The envisioned outcome is a more resilient and adaptive object detection model that excels in diverse environments, laying the groundwork for advancements in computer vision applications.

1.2 Research Questions

- What does good data augmentation depend on?
  - Is there an optimal strategy for augmenting an image?
  - ... or does a good augmentation strategy depend on the dataset? Is there an optimal strategy which can be applied to all images in a dataset?
- Can our proposed Deep Reinforcement Learning Algorithm improve the data augmentation strategy? Ideally, such that the performance of an object detection model increases when using the framework.

- Investigate if previously impressive results of DRL for data augmentation extend from image classification problems to object detection.

- What kind of policy is learned by a reinforcement learning agent? Which augmentations does it use and when?

1.3 Related Work

Previous work has been done on improving AI systems by letting another AI-system do the data augmentations. In a paper by Google Brain [Cubuk et al., 2019] a deep reinforcement learning agent is trained using the Proximal Policy Optimization algorithm to improve data augmentation strategies on CIFAR-10 [Krizhevsky, 2012] and ImageNet [Russakovsky et al., 2015] which are publicly available image classification datasets. The proposed method in this paper receives state of the art results with their augmentation strategy. In this research paper, the authors try to find a policy which consists of a number of sub-policies for augmenting an image. For each image in every mini-batch, one of the sub-policies is randomly selected for augmenting the image.

[Qin et al., 2020] uses a reinforcement learning algorithm to augment data for a segmentation task. They try to improve a model trained to segment kidney tumours from medical images. In contrast to the paper by Google Brain, this paper aims to learn a specific augmentation strategy for each image in the dataset.

Other approaches to augmenting images with neural networks is to use a Generative Adversarial Network (GAN) to create new images or change the style of images in the dataset. [Perez and Wang, 2017] does this successfully. They do this for a classification task and it could be argued that GAN:s might work better in this setting compared to object detection. In object detection, both class labels and locations of objects are required in order to train on the data. It is certainly a harder task for a GAN to produce an image which also has reliable ground truth labels.

The take-away from this is that reinforcement learning agents previously have been used for other tasks than object detection to successfully augment data which improves models.

1.4 Limitations

The first limitation of this thesis regards the domains for which we expect our system to work well on. As this thesis is done in collaboration with Axis Communications, a company famous for its video surveillance cameras, the primary domain will be surveillance images. We will not evaluate reinforcement learning agents on datasets from other domains such as medical images and do not attempt to create an agent which generalizes well across vastly different image domains.

The second limitation is that the object detection model architecture which we test and evaluate the agent on will be fixed. This research will not investigate whether the developed agent can improve the performance of different model types, the agent will be optimized for one particular model architecture.

The duration of a master thesis is approximately five to six months which sounds like a long time, but training these systems can take weeks. The field of deep reinforcement learning is quite new but despite this, there exists thousands upon thousands of ideas and concepts to improve reinforcement learning systems. This means the thesis will only consider a few different algorithms which we believe have potential.

1.5 Ethical Considerations

The datasets which are used in this research contains, among other things, people. This means there is a risk of built-in representational, demographic and cultural bias in the dataset. This can in turn make the model performance vary depending on where and how it is used. The datasets which are used in this research is a public dataset which is well curated by professionals and the other datasets are collected in-house by Axis Communications and are also well-curated and continuously monitored. Despite this, the datasets are large and this makes it difficult to keep track of every image which means fairness or bias issues can still occur.

Another concern with images on people is data privacy. With the public dataset the curators again have to be trusted to include images with people's consent. People in the European Union are protected by the General Data Protection Regulation (GDPR) and this regulation is honored by Axis Communications.

Training AI systems also have an environmental impact since computational power is needed. However, we think AI will have a positive impact on most societal challenges, including the environment. This particular thesis also investigates how to train AI systems more efficiently, also implicating a reduced environmental impact.

1.6 Outline

In the next chapter, the theory behind the used methods will be presented. Concepts from machine learning, deep learning and reinforcement learning will be described extensively. The theory section will also consider data augmentation and evaluation metrics. In Chapter 3, Methodology we will describe our methods and formulate the problem in a reinforcement learning framework. After the methodology chapter we will present the results from our experiments. Finally, we will discuss and conclude our results as well as present future work which can be done within the area.

2 Theory

This chapter will cover the most important concepts to get through this thesis. The concepts are within the area of machine learning and data processing. Most important are the concepts covered in sections 2.2, 2.4 and 2.5 and if you already feel proficient in machine learning, feel free to skip the other sections.

2.1 Foundations of Machine Learning

Machine learning is a broad term for computational methods that can learn from data. This is interesting as it allows one to make predictions about new, unseen data or discover useful patterns. In this section, some methods and algorithms will be presented that can learn from data. Let us begin with an example of when techniques from machine learning can be useful.

EXAMPLE: TITANIC KAGGLE CHALLENGE

A famous machine learning competition from the website kaggle.com is called "Titanic - Machine Learning from Disaster" [Cukierski, 2012] where the objective is to predict whether or not a person survived the Titanic disaster. The competitor in this challenge is given a dataset of people who embarked on the journey and if they survived or not. A person, x is described by a set of *features*. In this particular challenge, a person is described by:

$$x = \{age, ticket\ class, number\ of\ siblings\ onboard, sex, fare, ...\}$$

Thus, a person is encoded by a set of numerical features. Each person also has a *label* associated with it. In this case, the label, denoted y, can take on one of two different values:

$$y \in \{survived, not\ survived\}$$

The task is to build a model, which maps a person's features, x, to the correct label, y. In this competition, a competitor is scored on the *accuracy* (*#correct predictions / #predictions*) of its predictions on a set of new data, a dataset where the label is unknown to the competitor. In order to do this, a model which utilizes the labeled data must be learned. This problem is called a *classification problem*. Given a set of features, the objective is to assign the correct *class* (survived or not survived) to the entity described by the features.

During the course of this section, several other problems suited for machine learning will be explored. We will cover the main concepts to enable learning and show that a learning task can be reformulated to an optimization task. This section will also introduce the two main paradigms in machine learning; supervised and unsupervised learning.

Loss Functions

A core part of a machine learning problem is to choose and design a *loss function*. This is a function that describes the loss or the error of the model. Say the aim is to predict peoples heights (in an arbitrary unit). Let a model prediction of the height of N people to be an $N \times 1$ vector, $\hat{\boldsymbol{y}}$, where the i:th element is denoted with $\hat{y}_i$. The actual height of those N people is denoted $\boldsymbol{y}$. The loss function is a measurement of how wrong the model was in its prediction and can for instance be:

$$\mathcal{L}_{MSE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{2.1}$$

where MSE is short for *mean squared error*. The squared difference between the prediction and the truth is averaged over the N samples. Another common loss function is the *mean absolute error*:

$$\mathcal{L}_{MAE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i| \tag{2.2}$$

where the squared difference is replaced by the absolute value of the difference. As mentioned earlier, each of the predictions, $\hat{y}_i$ is the result of a model, $f(\cdot; \boldsymbol{\theta})$, which is parameterized by some vector, $\boldsymbol{\theta}$. Hence, given some input data point, x_i, the prediction is:

$$\hat{y}_i = f(x_i; \boldsymbol{\theta}) \tag{2.3}$$

If relation 2.1 and 2.3 are combined it results in:

$$\mathcal{L}_{MSE}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - f(x_i; \boldsymbol{\theta}))^2 \tag{2.4}$$

Loss functions are often denoted like this, as a function of the model parameters as those are the parameters to be changed and improved to minimize the loss function. There are several approaches to minimize loss functions, which will be discussed further.

Optimization

In the previous subsection, it was seen that the *learning* problem could be reformulated as an optimization problem. To minimize a loss function, an optimization algorithm is needed. In some cases, a simple closed form solution that is not too computationally expensive exists and that can be used. An example of this is if linear regression is used. An example would be to again try to predict the height of people, $\boldsymbol{y}$ given some features, e.g. those described in example 2.1. The linear regression model is described by:

$$\hat{y}_i = f(x_i; \boldsymbol{\theta}) = x_i^T \boldsymbol{\theta} \tag{2.5}$$

Here, x_i is the $D \times 1$ feature vector describing person i. Let X be the $N \times D$ matrix which describes the features of all people in the dataset. Then it's possible to write:

$$\hat{\boldsymbol{y}} = X\boldsymbol{\theta} \tag{2.6}$$

Let 2.1 be the loss function, this can be rewritten in matrix form:

$$\mathcal{L}_{MSE}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - x_i^T\boldsymbol{\theta})^2 = \frac{1}{N}(\boldsymbol{y} - X\boldsymbol{\theta})^T(\boldsymbol{y} - X\boldsymbol{\theta}) \tag{2.7}$$

The objective is to find $\boldsymbol{\theta}$ which minimizes 2.7 which can be done by finding $\boldsymbol{\theta}$ such that the gradient of the loss is the null vector.

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}_{MSE}(\boldsymbol{\theta}) = \mathbf{0} = \frac{2}{N}X^T(\boldsymbol{y} - X\boldsymbol{\theta}) \implies \boldsymbol{\theta}^* = (X^TX)^{-1}X^T\boldsymbol{y}$$

In this case, a mathematical expression for the optimal model parameters, $\boldsymbol{\theta}$ is obtained without the need for an optimization algorithm. However, sometimes the model, $f(\cdot;\boldsymbol{\theta})$ is more complex than in this linear regression exercise and a closed form solution is out of scope. This is when iterative optimization algorithms are needed.

Gradient Descent Algorithms. A classic iterative optimization algorithm is gradient descent. In this algorithm, the model parameters, $\boldsymbol{\theta}$ are iteratively updated in the opposite direction of the gradient. The update rule is:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^{(t)}) \tag{2.8}$$

where α is called the *step size* or *learning rate*. This is called a *hyper parameter*, it is not a parameter belonging to the model but a parameter that affects the training of the model. The choice of learning rate is essential for good learning. If the learning rate is too small, the risk is that the optimum is not reached within a feasible number of steps. On the other hand, if the learning rate is too large, it is possible to overshoot and miss the optimal value. This phenomenon is visualized in fig. 2.1. A popular variant of gradient descent is *Stochastic Gradient Descent* (SGD). Instead of computing the gradient on the entire dataset, the gradient is computed on one sample from the dataset chosen at random. The update rule is:

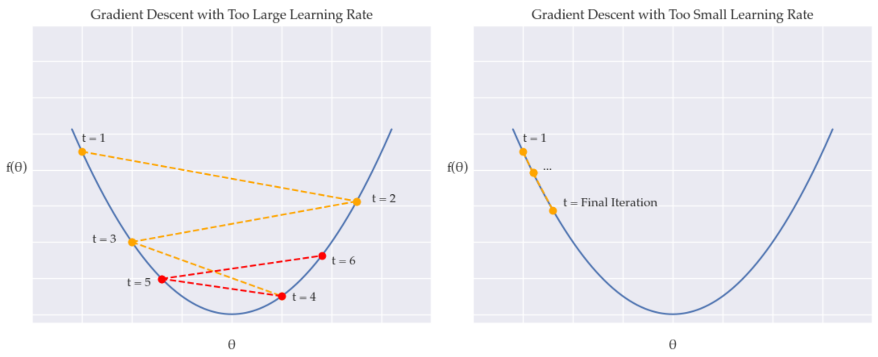
$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha\nabla_{\boldsymbol{\theta}}\mathcal{L}_i(\boldsymbol{\theta}) \tag{2.9}$$

where $\mathcal{L}_i$ is the loss for one example in the dataset such that $\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(\boldsymbol{\theta})$ (in the case of MSE or MAE loss). The motivation for this is that in expectation, the gradient taken for one sample is equal to the gradient of all samples.

$$\mathbb{E}[\nabla_{\boldsymbol{\theta}}\mathcal{L}_i(\boldsymbol{\theta})] = \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$$

Another common variant is mini-batch SGD which is an alternative between GD and SGD. Instead of updating the gradient on one sample, we update it on a mini-batch consisting of B samples. If, $\boldsymbol{g}$ is defined by:

$$\boldsymbol{g} = \frac{1}{|B|}\sum_{i\in B}\mathcal{L}_i(\boldsymbol{\theta})$$



(a) Issue with a too large learning rate. **(b)** Issue with a too small learning rate.

Figure 2.1 Potential issues with iterative optimization algorithms such as gradient descent. In the first case, the optimum is missed because the size of the steps is too large. In the second case, the size of the steps is too small and the optimum is not reached within a feasible amount of time.

then the update rule for mini-batch SGD is given by:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha \boldsymbol{g} \tag{2.10}$$

In the extreme case where $|B| = N$, mini-batch SGD is equivalent to the original (batch) GD proposed in 2.8. The motivation for using mini-batch SGD is that the computation can be easily parallelized on a GPU by running it on B threads. That is, mini-batch SGD is (about) as fast as SGD but should provide a slightly better estimate of the full gradient.

Other popular optimization algorithms are ADAM (Algorithm 1; [Kingma and Ba, 2017]) which utilizes the first and second moments of the gradient to update the parameters. The intuition for using moments in an optimization algorithm is that it may be less sensitive to ravines in the optimization landscape. A ravine is a region in the landscape where it is much more steep in some dimensions compared to others. The ADAM-algorithm introduces more parameters which must be tuned for the algorithm to function properly, β_1 and β_2. This is usually a downside as it can be difficult to know what values are suitable for a task beforehand. However, there are upsides and downsides with all optimization algorithms. It can be shown that under some circumstances, ADAM converges faster than SGD [Kim et al., 2017]. Although, this does not necessarily mean that it converges to a better local minima in a complex optimization landscape.

Supervised Learning

Several examples of supervised learning have already been seen in the previous sections. Supervised learning is the type of tasks where the learning

Algorithm 1: ADAM

```
Data:
    $\alpha$ : step size
    $\beta_i \in [0, 1)$ : exponential decay rates
    $\mathcal{L}(\theta)$ : stochastic objective function
    $\theta_0$ : initial parameters
Result:
    $\theta_t$ : resulting parameters

$m_0 \hookleftarrow 0$    // Initialize 1st moment vector;
$v_0 \hookleftarrow 0$    // Initialize 2nd moment vector;
$t \hookleftarrow 0$    // Initialize time step;
while $\theta_t$ not converged do
    $t \hookleftarrow t + 1$;
    $g_t \hookleftarrow \nabla_\theta \mathcal{L}_t(\theta_{t-1})$    // Get gradient w.r.t. objective function;
    $m_t \hookleftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$    // Update first moment;
    $v_t \hookleftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$    // Update second moment;
    $\hat{m}_t \hookleftarrow m_t / (1 - \beta_1^t)$    // Bias correction of $m_t$;
    $\hat{v}_t \hookleftarrow v_t / (1 - \beta_2^t)$    // Bias correction of $v_t$;
    $\theta_t \hookleftarrow \theta_{t-1} - \alpha \hat{m}_t / (\hat{v}_t + \epsilon)$    // Update step;
end
```

occurs because a set of supervisory signals are available. Example 2.1 in the beginning of this chapter is an example of a supervised learning task. There is a set of input data which we wish to map to a set of output data. The learning can occur because there is an explicit supervisor which can tell right from wrong to a model.

Supervised learning includes several different tasks. Example 2.1 is, as already mentioned, a *classification task*. More specifically, a *binary classification task* where the model should predict "Survived" or "Not survived". Another example of a binary classification task would be to determine whether an e-mail is spam or not. Binary classification can be generalized to include a larger number of classes, often called *multi-class classification*. An example of a multi-class classification task is found in the MNIST dataset [Lecun et al., 1998]. This is a dataset containing images of handwritten digits and their corresponding digit. The task is to predict the correct digit from the image, hence, there are 10 different classes (digit 0-9) to choose from.

Another class of tasks is *regression tasks*. In these tasks, the goal is to predict a continuous variable, such as the height of a person or the number of traffic accidents on a particular road from a set of features.

In *machine translation* the aim is to automatically translate speech or text in one language to another language. *Semantic segmentation* and *object detection* are two common tasks in computer vision and are supervised learning tasks. The objective in semantic segmentation is to correctly classify each pixel in an image to a group. As an example, this can be used to detect cancer cells in a medical image as an example.

Object Detection. This particular supervised learning task is of special interest for the reader of this thesis. The objective of this thesis is to augment data such that the performance of a model for object detection increases. Given an image, the objective is to localize certain objects and predict the correct class of these objects. In practice, this is often done by letting a model predict the location, width and height of *bounding boxes*. A bounding box is

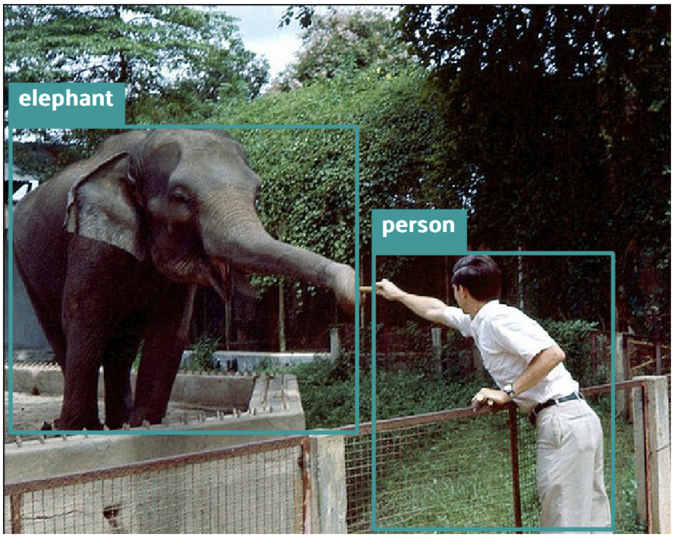


Figure 2.2 An image with its ground truth bounding boxes and classes used to train a model for object detection. The bounding boxes encapsulates the objects of interest, in this case people and elephants. Each bounding box is also associated with a class.

a rectangle where the sides are parallel with the x- and y-axes of the image and of minimal size such that it fully encapsulates the object. An image with bounding boxes (from the COCO dataset [Lin et al., 2014]) is visualized in fig. 2.2. In addition to finding the correct bounding boxes, each box should be assigned with the correct class from a set of predefined classes (multi-class classification). With fig. 2.2 as an example, the model is expected to find two bounding boxes and tell us that one of them contains an elephant and the other contains a person.

Unsupervised Learning

Unsupervised learning is a type of learning where no labels are available. It is a set of algorithms used to discover hidden structures and patterns in data. Common unsupervised tasks are clustering, anomaly detection, data generation and dimensionality reduction.

Clustering is the task of grouping data into clusters. Common algorithms for this task is k-means and DBSCAN. Learning an unsupervised task is also associated with minimizing a loss function as in supervised learning. The main difference is the absence of a label or some sort of ground truth. In k-means clustering, the objective is to minimize the objective in 2.11. Given a set of observations $(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_N})$, the aim is to divide these N points into $K(< N)$ clusters $(\boldsymbol{C} = (C_1, C_2, \ldots, C_k))$.

$$\underset{\boldsymbol{C}}{\arg\min} \sum_{k=1}^{K} \sum_{x \in \boldsymbol{C}} ||\boldsymbol{x} - \boldsymbol{\mu_k}||_p^2 \tag{2.11}$$

$\boldsymbol{\mu_k}$ is the *centroid* i.e. the mean of the points in cluster C_k if the L^2-norm is used.

Dimensionality reduction is another common task. This is used to reduce the number of dimensions for each data point which can be desirable in various situations. For instance when visualizing data of high dimension or to reduce computational cost for a down-stream task. When reducing the number of dimensions for a data point, there is obviously going to be some loss of information, so the main challenge in developing a good dimensionality reduction algorithm is to keep as much information as possible while compressing the data. One such algorithm is *Principal Component Analysis* (PCA). This algorithm finds the basis vectors which contains most of the variance in the data and transforms the data to this basis. The data can be compressed to an arbitrary number of dimensions, however fewer dimensions implicates a greater information loss.

2.2 Reinforcement Learning

Reinforcement learning (RL) aims to learn a strategy, or policy, dictating what actions to take in various situations; this is essentially learning a mapping from states to actions.

While reinforcement learning is its own unique paradigm in the machine learning landscape, it shares elements with both supervised and unsupervised learning.

In supervised learning, a model learns from labeled examples provided by a 'supervisor'. This supervisor, equipped with the knowledge of what is right and wrong, guides the model's behavior. If this guiding information is absent, the model will be unable to adjust or learn effectively. Reinforcement learning replaces this explicit guidance with a reward signal. Though the reward structure might be designed by an external entity (akin to a supervisor), it offers only implicit guidance. This means that an RL agent can explore its environment, making decisions and learning from the outcomes of those decisions, driven primarily by the rewards or penalties it receives. This style of learning, without explicit examples of right behavior, is similar with unsupervised learning's ethos.

However, there are key differences between unsupervised learning and RL. In unsupervised learning, the primary objective is to uncover hidden structures or patterns in the data, without a specific target outcome in mind. On the other hand, reinforcement learning is goal-oriented: the agent's decisions are geared towards maximizing the cumulative reward over time.

In summary, while reinforcement learning borrows elements from both supervised and unsupervised learning, it stands apart in its approach and objectives, emphasizing exploration, interaction, and reward maximization.

Main Components of Reinforcement Learning

One of the most central concepts in RL is that of an *agent*. An agent would be what in the previous section was referred to as the learner or model, it is this entity that is supposed to learn. This agent learns through interacting with what is called an *environment*. The environment in reinforcement learning represents everything that the agent interacts with and everything that is external to the agent. It responds to the agent's actions by providing a new state and reward. The environment encapsulates the dynamics, constraints, and rules that define how states transition and how rewards are determined based on the agent's actions. In addition to the agent and environment, other important elements in RL are those of *policy*, *reward signal*, *value function*, and in some cases *model*. The general RL-setup can be visualized as in figure 2.3.

The policy is a strategy that the agent employs to determine its actions. More technically, a policy in reinforcement learning is a function that defines the behavior of the agent. The policy can be understood as including the probability assignment between action for the current state, and also the action selection from that distribution. The policy directly selects an action for each state, based on the underlying probabilities. The deterministic or stochastic nature of the policy, in this case, refers to whether the same action is consistently chosen for a given state (deterministic) or whether the action is sampled based on a probabilistic process (stochastic). For example, when one says that a policy is made greedy, the policy chooses the action that gives the highest reward, which would corresponds to the action given the highest probability. Then, if the policy is made greedy, the same action (the

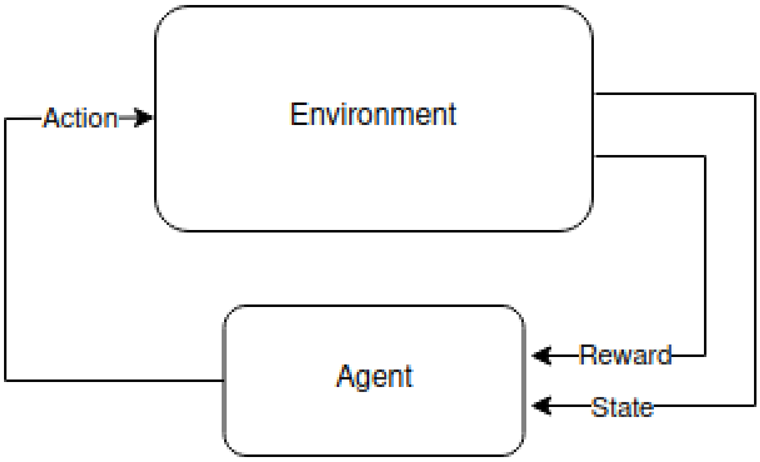


Figure 2.3 A visual representation off the interaction between the agent and the environment in a typical RL-setting. The agent receives a state and reward signal from environment, before deciding what action to take.

one given the highest probability is always chosen for each state, and the policy is deterministic.

The reward signal defines the immediate feedback the agent receives after taking an action in a specific state. This signal serves as the primary basis for the agent's learning. By gauging the quality of its actions through received rewards (or penalties), the agent can adjust its policy over time. The objective in most reinforcement learning scenarios is for the agent to maximize its expected cumulative reward over time.

The value function is an estimation of the expected cumulative reward an agent can achieve from a particular state, or state-action pair. While the reward signal gives immediate feedback, the value function provides a long-term perspective, indicating how beneficial it is for the agent to be in a specific state or to take a certain action from that state. By learning accurate value functions, the agent can make more informed decisions about its actions.

Lastly, a model represents the agent's understanding or approximation of the environment's behavior. Not all reinforcement learning approaches utilize a model; those that do, like model-based RL, use this internal representation to infer how the environment will respond to different actions, helping the agent to plan ahead.

In the grand scheme of RL, these components work in tandem, guiding the agent's learning journey. Through continuous interaction with the environment, receiving rewards, and adjusting based on its value function and policy, the agent strives to find the most effective strategy to achieve its objectives. Over time, with the right algorithms and adequate exploration, an agent can converge towards optimal or near-optimal behaviors in many tasks.

State Transitions & Episodes

An agent can interact with the environment by using a defined action set $\mathcal{A} = \{A^1, A^2, ...\}$, where the action set describes all possible actions the agent is allowed to take. At an arbitrary time step t, the agent observes the state S_t of the environment, with a corresponding reward signal R_t, as can be seen in figure 2.3. In many cases the agent is allowed to only inspect a function of the state, which sometimes is lossy. Then the environment outputs an observation O_t together with a reward signal, which in turn is corresponding to the latent state S_t. When this is the case the environment is said to be partially observable.

In order to obtain feedback from the environment a reward function R generates an immediate reward R_t. The reward function often depends on the current state of the environment only, but can also be more intricate.

One often also uses the trajectory in reinforcement learning. A trajectory is a sequence of interactions between the agent, environment, and reward function:

$$\tau = (S_0 \sim \rho_0(\cdot), A_0, R_0, S_1, A_1, R_1, \dots)$$

where the initial state S_0 is sampled from a start-state distribution ρ_0. In reinforcement learning, the course of interactions, represented by trajectories, can either span a limited time frame or continue indefinitely. This distinction leads to the concepts of finite and infinite games:

Finite Games (or Episodes): These have a definite ending point, known as a terminal state. When an agent reaches this state, the episode concludes. Finite games can be likened to playing a round of chess or completing a level in a video game. After the episode ends, the agent typically starts over in a fresh episode, often from a state sampled from the start-state distribution ρ_0. The length of each episode, which can vary from one episode to the next, is termed the "horizon." An agent's learning process might involve undergoing numerous episodes, allowing it to improve its policy from cumulative experiences across all episodes.

Infinite Games: These are perpetual, lacking a terminal state, and the agent-environment interactions continue indefinitely. In such scenarios, the agent's goal usually shifts to maximizing some form of discounted cumulative reward, given the infinite horizon. The discount factor, represented by

γ, determines the present value of future rewards, with rewards further in the future being reduced in value.

Given a state S_t and an accompanying action A_t, the state of the environment changes into S_{t+1}. Generally there are two types of transition processes, either deterministic or stochastic. If the transition process is deterministic, the next state is governed by a deterministic transition function:

$$S_{t+1} = f(S_t, A_t).$$

However, if the transition process is stochastic in nature, the process is instead determined by a random process:

$$S_{t+1} \sim p(S_t, A_t).$$

Given the dynamics of state transitions in reinforcement learning, particularly in environments with stochastic nature, a fundamental challenge arises: the determination of optimal action selection strategies. An intuitive approach is to adopt a greedy policy, wherein the agent, at each decision point, selects the action that maximizes the expected reward based on its current knowledge. This strategy emphasizes exploitation of acquired knowledge. However, the exclusive adoption of a greedy policy may preclude the agent from exploring potentially more rewarding actions that have not yet been sufficiently evaluated. This leads to a well-known conundrum in the realm of reinforcement learning termed the exploration-exploitation trade-off. The challenge lies in discerning when to rely on known strategies (exploitation) and when to investigate lesser-known actions in the hope of uncovering superior strategies (exploration).

The Markov Decision Process

A Markov decision process (MDP) is a discrete time stochastic control process, providing a mathematical framework for modeling decision-making in situations where the outcome is determined by both the state of the process, and the actions that are taken by an agent. One key property of a MDP is that the transition to a next state in a control process only depends on the current state, and not on the history of states. This is known as the Markov property, and is a property inherent to memoryless stochastic processes. By modelling a process this way, significant simplifications can be made, making the decision-problems more tractable and easier to analyse, as transitions can be calculated solely on the current state without regard to the possibly long history of previous states.

More formally, an MDP is defined as a 5-tuple, $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where $\mathcal{S}$ is the state-space, $\mathcal{A}$ the action-space, P the transition process, R the reward function, and γ a discount factor. One may omit γ, however setting $\gamma = 1$

effectively is the same, and using it allows us to define the MDP clearly in a reinforcement learning setting.

The general goal of optimizing under the framework of MDP is to find a policy π that maximizes the expected return for an agent that steers the process. More specifically the policy is a mapping from each state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$ to the probability distribution $\pi(a|s)$:

$$\pi(a|s) = p(A_t = a|S_t = s).$$

The expected return of a policy is the expected return over all possible trajectories τ, given that policy. The probability of a T-trajectory can be expressed as:

$$p(\tau|\pi) = \rho_0(S_0) \prod_{t=0}^{T-1} p(S_{t+1}|S_t A_t)\pi(A_t|S_t).$$

As such the expected return of a policy π can be defined:

$$J(\pi) = \int_{\tau} p(\tau|\pi)R(\tau) = \mathrm{E}_{\tau\sim\pi}\left[R(\tau)\right]$$

where $R(\tau)$ is the discounted return:

$$R(\tau) = \sum_{t=0}^{T} \gamma^t R_t$$

The Reinforcement Learning optimization problem can now simply be formulated: find the policy π^* that maximizes the expected return

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{2.12}$$

In the case of infinite horizon MDPs, this can be seen as finding the policy function π that maximises the expected cumulative reward:

$$\max_{\pi} \mathrm{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})\right] \quad \textit{where} \quad a_t = \pi(s_t).$$

In a similar way this can be formulated for the case of episodic MDPs where the policy intends to maximize the cumulative reward over each episode. To formalize this, a subset of the state space can be formed that includes every terminal state, $\mathcal{S}_T \subseteq \mathcal{S}$. Any time a state $s \in \mathcal{S}_T$ is reached, the current episode ends. The optimization task can then be easily formulated with trajectories, where a trajectory ends when a terminal state is reached. Let $\{\tau\}$ be a set of trajectories where each τ_i is the trajectory for episode i. The goal is to maximize the expected reward over trajectories, which can be formulated as:

$$\max_{\pi} \mathrm{E}_{\tau\sim\pi} \left[\sum_{t=0}^{T_i} \gamma^t R_{i,t} \right].$$

Partially Observable Markov Decision Processes

A Partially Observable Markov Decision Process (POMDP) extends the framework of MDP to situations where the agent cannot directly observe the underlying state of the process, as first described by [Åström, Karl Johan, 1965]. Instead the environment emits only an observation signal which does not fully represent the environments state. In POMDPs the agent can only infer the true state from perceived observations. Due to partial observability $s \in \mathcal{S}$ is hidden to the agent, which instead maintains a belief $b \in \mathcal{B}$ to estimate its state. As such, the agent can only infer the believed best action $a \in \mathcal{A}$ from b, which may or may not alter the state as expected. This is an additional layer of complexity added to the process, which often makes it more difficult to learn from. On the other hand, it also allows for a wider range of real-world applications as it is common that all information isn't available or that it is computationally intractable to analyse directly. This thesis does not fully formalize the ideas from POMDP, however they have been useful in the design of network architectures and for understanding the increased complexity which is inherited from the partial observability.

The POMDP is defined from a 7-tuple $(\mathcal{S}, \mathcal{A}, P, R, \Omega, Z, \gamma)$. This extends an underlying MDP, and in doing so introduces the new notions Ω and Z, where Ω denotes the observation space, while $Z : S \times A \to \Omega$ is the observation function. Do note that the observation space Ω and belief space $\mathcal{B}$ are not the same. The belief space $\mathcal{B}$ is a probability distribution over the state space $\mathcal{S}$, representing the agent's degree of belief in being in each possible state given its observations and actions up to the current time. Unlike the observation space Ω which consists of all possible observations that could be perceived by the agent, the belief space is typically the set of all possible probability distributions over the states. It encodes the agent's uncertainty about the current state due to partial observability and is updated over time as new observations are made and actions are taken. The notion of a state can be recovered from the notion of observation by considering the history $H_t = A_0, O_1, \ldots A_{t-1}, O_{t-1}$ where $O \in \Omega$. Any history naturally holds all the information we can know about the underlying state, and can be used to recover what is meant by a state for the agent. This can be denoted by S^H to distinguish it from the actual underlying state S, and can be described as a mapping from the history to this state: $S^{H_t} = f(H_t)$. However, the Markov property should not be forgotten and is a property that the function f has if and only if any two histories H and H' that are mapped to the same state, also have the same probabilities for their next observation.

$$f(H) = f(H') \implies Pr(O_{t+1} = o|H, A) = Pr(O_{t+1} = o|H', A) \tag{2.13}$$

When this property holds, S^H is said to be a Markov state. If f is the identity function, this naturally holds the Markov property. However it quickly becomes intractable in most situations, as the state would grow with t. To overcome this problem, the idea is to instead have some compact representation of the history, and let this be the new state to consider, that can be computed incrementally and recursively instead of doing it like f that takes whole histories as input.

$$S_{t+1} = u(S_t, A_t, O_{t+1}) \tag{2.14}$$

Here u is a so called state-update function. For the purpose of tractability u must be efficient to compute. For POMDPs, the environment is assumed to have a well defined latent state $\mathcal{S}$. The Markov (belief) state $B_t = \boldsymbol{b_t} \in \mathbb{R}^d$ that lies closest to hand is the distribution over the latent state space of the environment:

$$\boldsymbol{b_t}[i] = Pr(X_t = i|H_t) \quad \forall i \in \{1, \ldots, d\} \tag{2.15}$$

Given complete knowledge of the dynamics of the environment, the i:th component of the belief can be computed utilizing Bayes' theorem:

$$u(\boldsymbol{b}, a, o)[i] = \frac{\sum_{x=1}^{d} \boldsymbol{b}[i] p(i, o|x, a)}{\sum_{x=1}^{d} \sum_{x'=1}^{d} \boldsymbol{b}[x] p(x', o|x, a)} \tag{2.16}$$

where $p(x', o|x, a) = Pr(X_t = x', O_t = o|X_{t-1} = x, A_{t-1} = a)$. Unfortunately, this approach is often to cumbersome to use in practise, especially for high-dimensional, continuous (where the summations are swapped for integrals), cases. For a more conclusive theory on POMDPs both [Sutton and Barto, 2018] and [Zhu et al., 2018] provide interesting sources. When dealing with partial observability in a deep reinforcement learning setting, it is not so easy to explicitly define a belief state, history, and update function. These have instead been assumed to be learnt implicitly by a neural network, and so these ideas have influenced the choice of state space and neural architecture in a way that is hypothesised to increase learning efficiency and potential.

Value Functions and the Bellman Equation

As mentioned earlier, the policy and value function are fundamental concepts in reinforcement learning, and so these will now be handled with a more formal approach. The value of state s under policy π is denoted by

$v_\pi(s)$, and is measured by the expected return obtained by following trajectory τ sampled from policy π starting in state s. This is the state-value function, which with the MDP framework can be defined by:

$$\begin{aligned} v_\pi(s) &= \mathrm{E}_{\tau\sim\pi}\left[R(\tau)\middle|S_0 = s\right] \\ &= \mathrm{E}_{A_t\sim\pi(\cdot|S_t)}\left[\sum_{t=0}^{\infty}\gamma^t R(S_t, A_t)\middle|S_0 = s\right], \quad \forall s \in S \end{aligned} \tag{2.17}$$

A more action-centric function is the state-action-value function $q_\pi(s,a)$, which measures the value of taking action a in state s, and then follow the policy π. This can also be defined in a similar way to the state-value function:

$$\begin{aligned} q_\pi(s,a) &= \mathrm{E}_{\tau\sim\pi}\left[R(\tau)\middle|S_0 = s, A_0 = a\right] \\ &= \mathrm{E}_{A_t\sim\pi(\cdot|S_t)}\left[\sum_{k=0}^{\infty}\gamma^k R(S_t, A_t)\middle|S_0 = s, A_0 = a\right] \end{aligned} \tag{2.18}$$

As we can see from 2.17 and 2.18 there is a close resemblance, and the state-value function can written in terms of state-action-value function.

$$v_\pi(s) = \mathrm{E}_{a\sim\pi}\left[q_\pi(s,a)\right] \tag{2.19}$$

Optimality in the context of value functions paves the path to finding better policies that yield the best possible expected returns. For any given policy π, if there exists no other policy that achieves a higher expected return, across all states, then π is deemed an optimal policy. Interestingly, there may be several optimal policies for a particular task, all of which share the same state-value function, denoted as $v^*(s)$, and the same action-value function, denoted as $q^*(s,a)$.

The optimal state-value function, $v^*(s)$, for all $s \in \mathcal{S}$, is defined as the maximum value function over all policies:

$$v^*(s) = \max_\pi v_\pi(s)$$

Similarly, the optimal action-value function, $q^*(s,a)$, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$, is the maximum action-value function over all policies:

$$q^*(s,a) = \max_\pi q_\pi(s,a)$$

These definitions imply that if we know $v^*(s)$ or $q^*(s,a)$, we can consequently determine an optimal policy by choosing, at each state, the action that maximizes the expected return. Specifically, the optimal policy can be found from the optimal action-value function as follows:

$$\pi^*(s) = \arg\max_a q_\pi(s,a)$$

However, the central challenge is that we rarely know v or q to begin with. Reinforcement learning algorithms are designed to estimate these optimal functions and thereby derive optimal or near-optimal policies. Many of these algorithms make use of the *Bellman Equation* and the *Bellman Optimality Equation*, which are to be discussed next.

The Bellman Equation. The Bellman Equation is a recursive decomposition of the state-value function or state-action-value function which provides great insight to the structure of the problem as well as on how to calculate optimal policies. The idea stems all the way back to the 1950:s and the groundbreaking work done by Richard Bellman [Bellman et al., 1957]. His theories in the then new field of dynamic programming (DP) has been influential in many parts of science, especially in reinforcement learning where "Reinforcement Learning: An Introduction" by Richard S. Sutton and Andrew G. Barto popularized the approach.

To better see the recursive structure in 2.17 and 2.18, note the following relation:

$$\begin{aligned} R(\tau_{t:T}) &= R_t + \gamma R_{t+1} + \cdots + \gamma^T R_T \\ &= R_t + \gamma(R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T) \\ &= R_t + \gamma R(\tau_{t+1:T}) \end{aligned} \tag{2.20}$$

Note that the right hand sides appear in the expectation of the definitions of the state-value function and state-action-value function. One should also keep in mind that if the task is continuing, $T \to \infty$, the sum must be discounted ($\gamma < 1$), or else it will diverge. Using 2.20 the state-value function can be expanded:

$$\begin{aligned} v_\pi(s) &= \mathrm{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}\left[R(\tau_{t:T})\middle|S_t = s\right] \\ &= \mathrm{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}\left[R_t + \gamma R(\tau_{t+1:T})\middle|S_t = s\right] \\ &= \mathrm{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}\left[R_t + \gamma \mathrm{E}_{a'\sim\pi(\cdot|s'),s''\sim p(\cdot|s',a')}\left[R(\tau_{t+1:T})\right]\middle|S_t = s\right] \\ &= \mathrm{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}\left[R_t + \gamma v_\pi(s')\middle|S_t = s\right] \\ &= \mathrm{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}\left[r + \gamma v_\pi(s')\right] \end{aligned} \tag{2.21}$$

This equation 2.21 is the Bellman Equation for v_π. The equation states that the value of a state s under policy π is the expected return from that state. The expected return from that state is the sum of the immediate reward and the discounted value of the subsequent state s', averaged over all possible actions a in state s'. The policy $\pi(a|s)$ weights the sum with the likelihood of taking action a in state s, while $p(s', r|s, a)$ is the transition probability given the current state and the chosen action. With minor modifications the Bellman Equation can also be derived for the state-action-value function:

$$\begin{aligned}
q_\pi(s,a) &= \mathrm{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}\left[R(\tau_{t:T})\middle|S_t = s, A_t = a\right] \\
&= \mathrm{E}_{a\sim\pi(\cdot|s),s'\sim p(\cdot|s,a)}\left[R_t + \gamma R(\tau_{t+1:T})\middle|S_t = s, A_t = a\right] \\
&= \mathrm{E}_{s'\sim p(\cdot|s,a)}\left[R_t + \gamma \mathrm{E}_{a'\sim\pi(\cdot|s'),s''\sim p(\cdot|s',a')}\left[R(\tau_{t+1:T})\right]\middle|S_t = s, A_t = a\right] \\
&= \mathrm{E}_{s'\sim p(\cdot|s,a)}\left[R_{t+1} + \gamma \mathrm{E}_{a'\sim\pi(\cdot|s')}\left[q_\pi(s',a')\right]\middle|S_t = s\right] \\
&= \mathrm{E}_{s'\sim p(\cdot|s,a)}\left[r + \gamma \mathrm{E}_{a'\sim\pi(\cdot|s')}\left[q_\pi(s',a')\right]\right] \qquad (2.22)
\end{aligned}$$

The value of taking action a in state s under policy π is the expected immediate reward plus the discounted value of the action values in the next state, averaged over all possible next states s', and subsequent actions a', as described by [Sutton and Barto, 2018].

From these equations the Bellman optimality equation can be derived for value functions, which is a way of formalizing when a value function is optimal. Intuitively this is an equation that says that if we recursively maximize the value function over each state, we get the optimal value function:

$$v^*(s) = \max_a \mathrm{E}_{s'\sim p(\cdot|s,a)}\left[R(s,a) + \gamma v^*(s')\right] \qquad (2.23)$$

and for the state-action-value function:

$$q^*(s,a) = \mathrm{E}_{s'\sim p(\cdot|s,a)}\left[R(s,a) + \gamma \max_{a'} q^*(s',a')\right]. \qquad (2.24)$$

For a full derivation of these and a more in depth theory on the Bellman Equation and the Bellman Optimality Equation, both [Dong et al., 2020] and [Sutton and Barto, 2018] provide excellent resources. A natural question now is how to learn an optimal or close to optimal policy that solves the RL optimization problem from 2.12.

Learning and RL Algorithms

Having established the Bellman Equation as a foundational concept in understanding the dynamics of value functions in reinforcement learning, this section shifts focus towards the learning aspect in RL. Here, learning is defined as an algorithmic process enabling an agent to refine its decision-making capabilities through accumulated experience. These algorithms are the mechanisms through which an agent improves its policy based on interactions with the environment.

The goal of finding an optimal policy, as formalized by the Bellman Optimality Equation, is at the core of reinforcement learning. However, the practical challenge lies in how an agent can learn such policies efficiently, especially when the state space, action space, and dynamics are complex or vast. When the complexity of the task is not too great, the concept of dynamic programming and straight forward, often fully deterministic, algorithms can be deployed and function well. One of the most important such algorithms is policy iteration. Even though policy iteration and its closest relatives cannot be applied directly in some tasks, many of the more advanced algorithms build upon the overall idea, these more advanced methods are called *Generalized Policy Iteration*.

Policy Iteration and Generalized Policy Iteration. Policy iteration, a DP algorithm, is a two-step process involving policy evaluation and policy improvement. Initially, one starts with an arbitrary policy and evaluate it to determine the value of each state under that policy. As the dynamics of the environment is assumed to be completely known here, and as the problem is assumed to be modelled as an MDP, the Bellman equation for state-value functions from 2.21 can be applied recursively. This is the policy evaluation step. Once the policy evaluation is finished, the next step is to improve that policy. For some state, s its desirable to take another action $a \neq \pi(s)$ such that the returns are improved. This action can be evaluated by the state-action-value function for that state action pair, as given by 2.18. Then if

$$q(s,a) = q(s,\pi'(s)) \geq v(s)$$

the new policy π' must be at least as good as π. This shows the process for a single, arbitrary, state. However it can be extended to all states as each value function is built up recursively from the other value functions as seen by the Bellman Equations. Consider the new greedy policy π' given by:

$$\pi'(s) = \arg\max_a q_\pi(s,a) = \arg\max_a \mathrm{E}_{\tau\sim\pi}\left[R(\tau)\middle|S_t = s, A_t = a\right] \tag{2.25}$$

The greedy policy fulfills that $q_\pi(s,\pi'(s)) \geq v_\pi(s) \forall s$. It can be shown that this necessarily improves the policy if it is not optimal, and that the policy does not change if it is optimal. Suppose that this happens, that policy π' is as good as but not better than policy π. Then $v_\pi = v_{\pi'}$, and the policy is thus optimal by the Bellman optimality equation from 2.23.

Generalized Policy Improvement (GPI) represents a foundational concept in reinforcement learning, encapsulating the cyclical process of policy evaluation and improvement. This iterative approach hinges on two core steps:

1. **Policy Evaluation**: Assessing the current policy's effectiveness, typically by estimating the value function, which represents the expected return from each state under the current policy.

2. **Policy Improvement**: Refining the policy to make more optimal decisions, often based on the updated value function.

The interplay between these steps is nuanced. On one hand, they work in opposition: a policy made greedy based on current evaluations may render those evaluations less accurate, necessitating further evaluation. On the other hand, they are complementary, as each cycle of evaluation and improvement incrementally nudges the policy towards optimality. GPI is more general than classic policy iteration. It does not mandate a complete or precise evaluation of the policy at every step. Instead, it allows for partial updates to the value function, followed by adjustments to the policy. This flexibility is what makes GPI more of a meta-algorithm, a framework underpinning many specific reinforcement learning algorithms.

Monte Carlo Methods. One of the primary limitations with DP algorithms such as policy iteration is the need of a complete model of the environment. These algorithms need knowledge of the state transition dynamics conditioned on each action. Monte Carlo (MC) methods are a family of sampling based learning algorithms in reinforcement learning which do not rely on any model of the environment, instead it can learn solely from experience.

Monte Carlo methods represent a model-free approach in reinforcement learning, where learning is achieved directly from episodes of experience without any assumption of knowledge about the environment's dynamics. This method is particularly useful in scenarios where the environment's model is unknown or too complex to be formulated. The key idea behind Monte Carlo methods involves learning from complete sequences of states, actions, and rewards, and using these to estimate value functions. A simple MC update rule, where V under policy π is the estimate of v_π is:

$$V(S_t) \hookleftarrow V(S_t) + \alpha \left[R_\tau - V(S_t) \right], \tag{2.26}$$

where α is a step-size parameter like the learning rate in traditional machine learning. Here R_τ is the *target* for the update rule.

The process in Monte Carlo methods involves:

- Generating episodes based on the current policy.
- Calculating returns (cumulative rewards) for each state visited in the episode.
- Averaging these returns over multiple episodes to estimate the value function.

This approach is distinct from DP methods, which require a complete model to predict state transitions. Monte Carlo methods only need the ability to generate episodes, making them applicable to a broader range of problems. GPI can be effectively used together with Monte Carlo methods. In this context, GPI becomes a framework for iteratively improving the policy based on the value function estimates derived from the Monte Carlo approach. The integration of GPI with Monte Carlo methods can be described as follows:

1. Use Monte Carlo methods for policy evaluation: Estimate the value function of the current policy by averaging the returns from complete episodes.
2. Apply GPI for policy improvement: Based on the estimated value function, refine the policy to make it more optimal. This can be achieved by making the policy greedier with respect to the estimated value function.

By combining Monte Carlo methods with GPI, we obtain a powerful algorithmic framework that benefits from the model-free nature of Monte Carlo methods while leveraging the systematic policy improvement mechanism of GPI. This results in a flexible and robust approach to learning in environments where a model is either unavailable or impractical to use. While Monte Carlo methods provide a valuable model-free approach in reinforcement learning, particularly in environments where a complete model is unavailable or impractical, they have several inherent limitations:

- **High Variance**: Monte Carlo methods can exhibit high variance in their value estimates, especially in stochastic environments. This variance can lead to slow convergence and unstable learning.
- **Exploration Dependence**: These methods require thorough exploration of the state and action space to generate reliable estimates. In large or complex environments, achieving sufficient exploration can be challenging, leading to poor performance in under-explored regions.

- **Episode Length Sensitivity**: From 2.26 it is apparent that the full trajectory for the episode must be computed before any update can be made. The reliance on complete episodes for updates makes Monte Carlo methods less efficient in scenarios with very long or continuous episodes.
- **Lack of Bootstrapping**: In reinforcement learning, bootstrapping refers to the process of using current estimates to update future estimates. Monte Carlo approaches do not utilize bootstrapping, relying solely on actual returns for updates. This absence of bootstrapping can limit the speed of learning compared to methods that use existing estimates to inform updates.
- **Suitability for Episodic Tasks**: These methods are primarily designed for episodic tasks with clear terminal states. They are not ideally suited for continuous, non-terminating tasks.
- **Delayed Credit Assignment**: In environments with sparse or delayed rewards, Monte Carlo methods can struggle with the credit assignment problem, as it can be difficult to determine which actions were responsible for obtaining the reward.

These limitations often make Monte Carlo methods less preferable in certain scenarios, leading to the adoption of alternative approaches like Temporal Difference learning.

Temporal Difference Learning. Temporal Difference (TD) learning is another learning method in RL that combine the ideas from both DP and Monte Carlo methods. Just like DP, TD learning uses information from other subsequent states to update the value for the current state (bootstrapping), and just like MC it is a sample-based and model-free technique.

From 2.26 it was noted that the target for the MC update was R_τ, which can only be known after a full episode is completed. The TD method does not require this, as the target for the update is changed. The simplest TD update rule can be formulated as:

$$V(S_t) \hookleftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{2.27}$$

This means that the update can be made as soon as the transition from S_t to S_{t+1} is complete, with no need to wait for the full trajectory τ for the current episode. This means that the problem of delayed credit assignment found with the MC method is made less prominent, as well as the problem of episode length sensitivity. It is also apparent from the TD update rule that it uses bootstrapping and sampling, as it updates the current value of $V(S_t)$ based on the estimated value of the next state $V(S_{t+1})$ (bootstrapping), and the actual reward received R_{t+1} (sampling). However, although bootstrapping is used as in DP methods, there is no need for a complete model of the dynamics, as the bootstrapping in itself is sampled based.

The method from 2.27 is also called the TD(0) (one-step TD), as it looks one step forward. The TD(0) method can be used for policy evaluation, as shown in algorithm 2.

Algorithm 2: TD(0) Policy Evaluation Algorithm

```
Data:
  $\pi$ : The policy to be evaluated
Result:
  $V_\pi(s)$ : The evaluated state-value function under policy $\pi$

initialize $V(s)$    // An arbitrarily initialized state-value function;
for each episode do
    $S \sim \rho_0$;
    for each step in current episode do
        $A \hookleftarrow \pi(S)$;
        $S', R \hookleftarrow Env(A)$;
        $V(S) \hookleftarrow V(S) + \alpha\,[R + \gamma V(S') - V(S)]$;
        $S \hookleftarrow S'$;
    end
end
$V_\pi(s) \hookleftarrow V(s)$;
return $V_\pi(s)$
```

It is also important to note that the terms inside the brackets of 2.27 forms an error term:

$$\delta = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{2.28}$$

This error, the TD-error, measures the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$ of the same quantity (better in the sense that it is estimated from more samples).

Q-learning Algorithm. Q-learning is a simple TD-learning algorithm, developed by [Watkins, 1989]. The algorithm was initially developed to overcome some of the most obvious problems with MC methods, not least the delayed credit assignment issue. In Q-learning the goal is to find the Q-function that best approximate the true optimal state-action-value function $q^*(s,a)$, and in doing so finding the optimal policy implicitly. In order to do this the Q-learning algorithm uses an iterative TD approach for the Q-function:

$$Q(S_t, A_t) \hookleftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{2.29}$$

The learned state-action-value function directly tries to approximate the optimal policy q^*, independently of the current policy. However, the policy is still used to decide which state-action pairs are visited. This means that the Q-learning algorithm has convergence proofs as long as all state-action pairs continues to be visited. A consequence of this is that one usually defines two policies, a *collect policy* and an *evaluation policy*. The collect policy is only used to collect experiences and updating the Q-function, while the evaluation is the resulting policy when the algorithm finishes. This is why the Q-learning algorithm is an *off-policy algorithm*, it learns from a different policy than the current policy, allowing it to collect more diverse experiences in contrast to the policy iteration and MC methods described, where the learning happens from the current policy. That the Q-learning algorithm is off policy can be seen from that the target in 2.29 has no connection to the current policy, it is instead a maximization over the current Q-function.

It is common to have an ϵ-greedy collect policy, but there are many more potential options here. By letting π_C stand for an arbitrary collect policy, the full (one-step) Q-learning algorithm can be presented in procedural form, as seen in Algorithm 3.

Algorithm 3: Q-learning algorithm

```
Data:
    $\pi_C$ : The collect policy.
Result:
    $Q_{\pi*}(s,a)$ : The evaluated state-action-value function.

initialize $Q(s,a)$    // An arbitrarily initialized state-value function;
for each episode do
    $S \sim \rho_0$;
    for each step in current episode do
        $A \hookleftarrow \pi_C(S)$    // Get action according to policy;
        $S', R \hookleftarrow Env(A)$    // Observe reward + next state;
        $Q(S,A) \hookleftarrow Q(S,A) + \alpha\,[R + \gamma \max_a Q(S',a) - Q(S,A)]$;
        $S \hookleftarrow S'$;
    end
end
$Q_{\pi^*}(s,a) \hookleftarrow Q(s,a)$;
return $Q_{\pi^*}(s,a)$
```

This algorithm can be extended to multi-step Q-learning, where the bootstrapping is done over multiple steps forward, however the ideas are essentially the same. An interesting, and problematic, property of the Q-learning algorithm is the *maximization bias* inherently introduced by the update rule from 2.29. Here the policy is derived as a maximization from the target policy, given the current Q-function. The maximization is taken over the estimated values $Q(S', a)\ \forall\ a$, which may lead to significant positive bias. Assume the expected value of some fixed state s, in which any of the actions $\{a_1, a_2, \dots a_n\}$ can be taken, each of which has an expected value b. As the maximization is done over the estimated state-action-values $\{Q(s, a_1), Q(s, a_2), \dots, Q(s, a_n)\}$, its obvious that there is a positive bias to a value $c > b$, although the true value is b.

To mitigate the maximization bias problem in Q-learning, the *Double Q-learning* algorithm has been developed. Double Q-learning addresses the maximization bias by using two separate Q-value estimators, Q_A and Q_B, and updates them alternately. The key idea is to decouple the action that is chosen from the action whose value is updated. This is achieved as follows:

1. During each update, one of the two Q-functions is selected for updating the action value. Suppose Q_A is selected.
2. The action A^* for the state S' is chosen based on the Q-function not being updated (in this case, Q_B).
3. The target value for updating Q_A is computed using $Q_B(S', A^*)$ along with the received reward.

By using two Q-function and updating them alternately in this manner, Double Q-learning effectively reduces the overestimation bias. The overestimation from one table is not immediately used to reinforce the other table's estimates. This separation allows for more accurate estimation of the true value of actions. The algorithm can be described as follows:

This approach allows Double Q-learning to provide a more stable and less biased estimation of the state-action values, for more accurate policy learning. The obvious drawback of this approach is the increased memory requirements needed to hold two Q-estimators in memory.

Function Approximation

Function approximation is an important concept in reinforcement learning, particularly crucial in context where state and or action spaces are too vast for tabular methods. It involves approximating value functions, such as the state-value function or the state-action-value function, with parameterized functions. This enables extrapolation from seen data to unseen data. Assuming no computer has infinite memory and instant look-up time, this is necessary in many real world applications where problems often are large or continuous. Using function approximation can also enable very efficient learning, as information can be shared across similar states.

Algorithm 4: Double Q-learning algorithm

```
Data:
    $\pi_C$ : The collect policy.
Result:
    $Q_{\pi*}(s,a)$ : The evaluated state-action-value function.

initialize $Q_A(s,a)$ and $Q_B(s,a)$    // Arbitrarily initialized tables;
for each episode do
    $S \sim \rho_0$;
    for each step in current episode do
        $A \hookleftarrow \pi_C(S)$;
        $S', R \hookleftarrow Env(A)$;
        if randomly select $Q_A$ for update then
            $A^* \hookleftarrow \arg\max_a Q_B(S',a)$;
            $Q_A(S,A) \hookleftarrow Q_A(S,A) + \alpha\,[R + \gamma Q_B(S',A^*) - Q_A(S,A)]$;
        else
            $A^* \hookleftarrow \arg\max_a Q_A(S',a)$;
            $Q_B(S,A) \hookleftarrow Q_B(S,A) + \alpha\,[R + \gamma Q_A(S',A^*) - Q_B(S,A)]$;
        end
        $S \hookleftarrow S'$;
    end
end
$Q_{\pi^*}(s,a) \hookleftarrow \frac{1}{2}(Q_A(s,a) + Q_B(s,a))$;
return $Q_{\pi^*}(s,a)$
```

Basis of Function Approximation. The function approximation approach involve representing the value functions (such as the Q-function in Q-learning) not as discrete tables but as continuous functions. This representation is achieved through various approximates, such as:

- **Linear functions**: Simple and effective for certain problems, linear functions offer ease of computation and interpretation.
- **Polynomials**: Here the value function is represented by a polynomial of the state variables. For example, a quadratic function in with two state variables would be $v(x,y) = ax + bxy + cy + dx + ey + f$ where the coefficients are the learnable parameters.
- **Basis Functions**: The value functions can be expressed as a linear combination of basis functions, where each basis function is a transformation of the input state, and the input is weighted by learned parameters. Common choices of basis functions include Fourier bases and radial basis function, among others. If we exemplify with a Fourier

basis, and assume a state represented by the singleton state x, the feature representation of x might be:
$u(x) = [1, \sin \pi x, \cos \pi x, \sin 2\pi x, \cos 2\pi x]^T$, and the learnable weights $\vec{\omega} = [\omega_1, \omega_2, \omega_3, \omega_4, \omega_5]^T$. Then the estimated value for state x would be $v(x) = \vec{w}^T u(x)$.

Although the above have the advantage of being able to handle large or continuous problems, there are a few drawbacks. In order for the approximation to be effective, the value functions true form should be compatible with the choice of function approximator. This makes the choice of function approximator problem specific, as the user must assume a domain a priori. If this is possible, the above can be very effective. But for complicated, non-linear functions where the value function landscape is not known beforehand, more expressive function approximators can be necessary.

Function Approximation in Q-learning. In Q-learning, function approximation can be used to estimate the state-action-value function, $Q(s, a)$. Traditionally, Q-learning employs a tabular approach, which becomes impractical in large-scale problems. When adopting function approximation the standard Q-learning changes somewhat:

- **Generalized Q-function**: The Q-function is represented as $Q(s, a; \theta)$, where θ are the parameters of the approximator.
- **Update Rule Adaptation**: The update rule in Q-learning is modified to adjust the parameters θ based on the temporal difference error δ (see eq2.28), aiming to minimize the difference between predicted Q-values and observed rewards.

Integrating function approximation into Q-learning transforms it into a powerful tool capable of handling complex, high-dimensional environments. This approach retains the core principles of Q-learning – learning from the temporal difference error and updating estimates towards a policy that maximizes future rewards – while making it applicable to a broader range of real-world problems where less information (but more data) is available. One very expressive and powerful function class that is often utilized to express the generalized Q-function are artificial neural networks together with learning approaches from deep learning, which will be covered next.

2.3 Deep Learning

This section will primarily discuss deep learning in the context of artificial neural networks. However, deep learning and artificial neural networks are actually separate terms and cannot be used interchangeably. Deep learning is the concept of stacking entities such that each layer can learn some level of abstraction. The most widely used form of deep learning today is artificial neural networks which consists of layers of *neurons*. Other forms of deep learning exists, a variant could be stacking Boltzmann machines on top of each other.

Feed-Forward Neural Networks

A feed-forward network (FFN) is a type of neural network in which the information flows in one direction. Each layer receives information from the previous layer and sends it to the next layer. The layers consists of neurons or *perceptrons* (these terms will be used interchangeably), these are represented by the circles in fig. 2.4. A neuron takes several inputs, does some

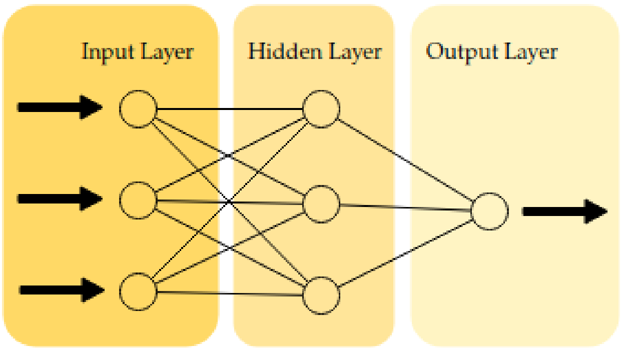


Figure 2.4 Schematic sketch of a feed-forward network. Data flows in through the input layer, then to a hidden layer (often multiple such layers). The hidden layers passes on information to the output layer.

mathematical operations on them which includes a non-linear operation, and then outputs the result. The output for neuron j can be described by

$$o_j = f\Big(\sum_{k=1}^{K} w_{kj}x_k + b_j\Big)$$

where x is the input, f is a non-linear *activation function*, $w_{..}$ is conventionally referred to as the *weights* and $b_{.}$ as the *bias*. Popular choices for activation functions are the sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.30}$$

the rectified linear unit (ReLU):

$$\text{ReLU}(x) = \max(0, x) \tag{2.31}$$

and the hyperbolic tangent:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.32}$$

This pattern of calculating a linear combination of the outputs from the previous layer and then applying an element-wise non-linear function is what constitutes the representational power of the neural network. They are in fact universal function approximators under certain circumstances. A famous result is Barron's Theorem [Barron, 1993] which states that for any function $f : \mathbb{R}^d \to \mathbb{R}$ that has a Fourier transform:

$$\hat{f}(\omega) = \int_{\mathbb{R}^d} f(x) e^{-i\omega^T x} dx$$

and is sufficiently smooth such that:

$$\int_{\mathbb{R}^d} |\omega||\hat{f}(\omega)| d\omega < C$$

where $|\omega| = (\omega \cdot \omega)^{1/2}$. Then there exists a function, f_n for $n \geq 1$ and $r > 0$ on the form:

$$f_n(x) = \sum_{j=1}^{n} c_j \phi(x^T w_j + b_j) + c_0$$

such that

$$\int_{|x|<r} (f(x) - f_n(x))^2 dx \leq \frac{(2Cr)^2}{n}$$

This is quite a remarkable result as it states that any sufficiently smooth function can be approximated by a one-hidden layer neural network. The approximation error is bounded by the smoothness of the function (C), the size of the region (r) and the number of neurons (n). It is possible to shrink this bound by using more neurons, n, i.e using a larger neural network. Many similar theorems could've been picked to demonstrate the representational power of neural networks and new mathematical results are still found today.

Neural networks are trained by using the back-propagation algorithm which was first proposed in this context by [Rumelhart et al., 1986]. The back-propagation algorithm is a smart way of computing the partial derivatives of the weights and biases with respect to the loss function in a neural network. It utilizes a forward pass and a backward pass through the network, resulting in a time complexity of $\mathcal{O}(K^2 L)$ for a neural network with L layers, each containing K neurons.

Convolutional Neural Networks

So far, the examples of neural networks have been fully connected. In fig. 2.4 for instance, it is possible to see that each neuron in layer l is connected to every neuron in layer $l-1$ and every neuron in layer $l+1$. Convolutional networks consists of convolutional layers and work slightly differently. In this thesis, images are often the input to the network and this is a good application for convolutional neural networks. If a fully connected network were to process an image, the first step would be to flatten the image as described in fig. 2.5. This might remove the spatial properties of the image. A convolutional layer consists of a convolution which is made up of one or

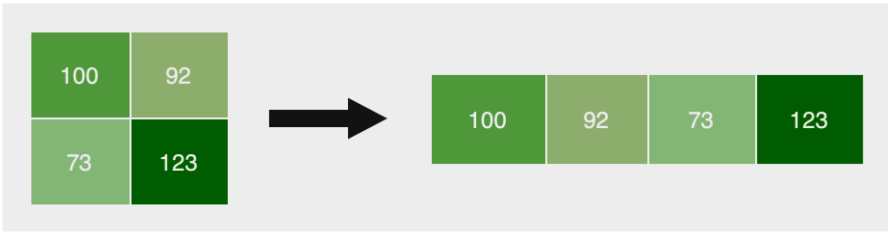


Figure 2.5 A 2×2 pixel image and its shape after flattening. The flattened version is what a fully connected network sees.

more filters. The filters are local, in the sense that the convoluted feature is affected only by the pixels closest to it. Weights of the filters make up the trainable parameters in a convolutional neural network. Figure 2.6 shows a 2×2 filter applied to a 3×3 pixel image. An image often have multiple

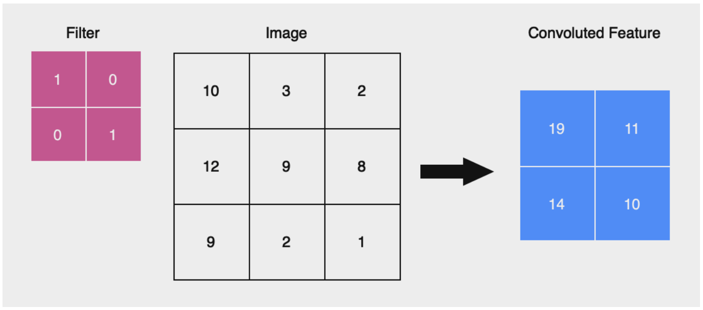
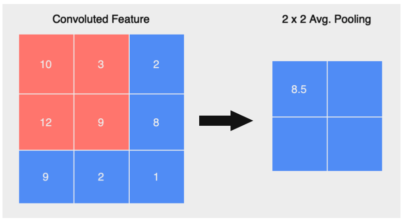
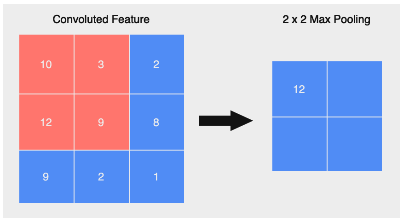


Figure 2.6 A 2×2 filter applied to an image produces a convoluted feature.

channels, such as an RGB-image which has three channels, one for each of the base colors; red, green and blue. It is a common strategy to use at least one filter for every channel and often more than one filter per channel. This results in data that grows in depth during inference through the network.



(a) 2×2 Average Pooling operation.



(b) 2×2 Max Pooling operation.

Figure 2.7 Example of two pooling operations, average and max pooling. The pooling operations down-samples the input, this can be seen in both examples where the input size is 3×3 while the output is 2×2.

An image of input size (H, W, C) which passes though a convolutional layer resulting in shape (H', W', C') have a smaller or equal-sized height $(H' \leq H)$ and width $(W' \leq W)$ while the number of channels grows of stays the same $(C' \geq C)$. An image passing through a convolutional neural network can therefore often look like a pyramid. The number of output channels can vary between layers and is an architectual choice in a convolutional neural net. Filters in the convolutional layer are followed by a non-linear function such as 2.30, 2.31 or 2.32 and the non-linearity is usually followed by a *pooling* function. This is a down-sampling operation and the most common way to do this is through either average pooling (take average of nearby pixels) or max pooling (take the maximum of nearby pixels. Examples of max and average pooling can be found in fig. 2.7. Choosing a pooling operation can also be seen as a kind of hyper-parameter. It is often desirable to down-sample images, especially if the resolution is high since the input space is very large. An image in full HD resolution is $1920 \times 1080 = 2,073,600$ pixels per channel.

Other common layers in convolutional neural networks are skip-connections in which a few layers are skipped. If $\boldsymbol{F}$ is a series of layers and $\boldsymbol{X}$ is the input to those layers, then

$$\boldsymbol{Y} = \boldsymbol{F}(\boldsymbol{X})$$

but with a skip connection the input to $\boldsymbol{F}$ is added to the output such that:

$$\boldsymbol{Y} = \boldsymbol{R}(\boldsymbol{X}) + \boldsymbol{X} \tag{2.33}$$

where $\boldsymbol{R}$ is called a residual branch. This addresses the *vanishing gradient problem* where the gradient goes to zero during training of a network, which results in very small or no updates in the network. The skip-connections forces the gradient in 2.33 to be at least 1. The vanishing gradient problem can arise during training of very deep neural networks. The paper introducing the ResNet architecture [He et al., 2015] showed that a deeper network does not always perform better despite having more trainable parameters, they proposed skip-connections as a solution to make deeper networks function better.

Another common operation to do in a deep neural network is batch-normalization. This keeps track of a running average and running variance in order to normalize each batch seen by the network. Often a batch-normalization layer also contains two trainable parameters which are multiplied respectively added to the normalized batch.

When designing a convolutional neural network, it is common to use a combination of all the mentioned layers. A common design is to construct a block of layers where a block might consist of convolutional filters, an activation function, a pooling function and batch normalization. A few of these blocks can be repeated to obtain a deeper neural network. After the series of convolutional blocks, it is common to flatten the output and process it through a fully connected network to complete the task, which can be classification or regression for instance.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) is a class of neural networks that are well-suited for sequential data and tasks where the order of information is crucial. Unlike feed-forward networks, which process input data in a single pass, and convolutional networks, which operate on fixed-size input, RNNs can handle sequential data of varying lengths. This makes them particularly useful for tasks like natural language processing, time series analysis, and speech recognition.

RNNs have recurrent connections that allow information to persist across different time steps. This is achieved through loops in the network, enabling the hidden state to capture information from previous time steps. The hidden state in an RNN serves as a memory of the network, capturing information about the sequence processed so far. It is updated at each time step based on the current input and the previous hidden state. Fig. 2.8 shows a schema of an RNN in rolled and unrolled form. The unrolled form shows the computational flow of an RNN.

Each hidden state is computed by using the previous hidden state, the following equations describe how data from previous time-steps is incorporated:

$$\begin{aligned} y_t &= \sigma(W_{yh}h_t + b_y) \\ h_t &= \sigma(W_{hx}x_t + W_{hh}h_{t-1} + b_h) \end{aligned}$$

The hidden state is calculated from the previous hidden state and the current input.

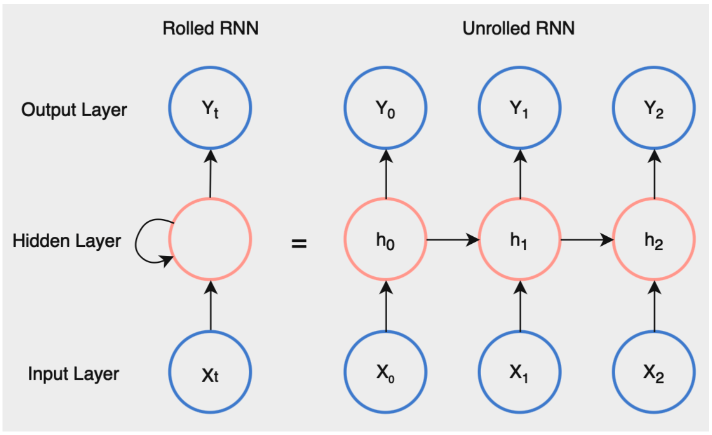


Figure 2.8 Schema of a Recurrent Neural Network drawn rolled and unrolled which shows the computational flow of an RNN.

Training RNNs can be challenging due to the vanishing problem and exploding gradient problem where the gradient grows out of control as they are propagated back through time, making it difficult for the network to learn long-range dependencies. While RNNs are designed to capture sequential dependencies, they can struggle to learn long-term dependencies. This has led to the development of more advanced RNN architectures, such as Long Short-Term Memory (LSTM) networks [Hochreiter and Schmidhuber, 1997] and Gated Recurrent Unit (GRU) networks [Cho et al., 2014].

LSTMs and GRUs are specialized RNN architectures that address the vanishing gradient problem by incorporating gating mechanisms. These architectures enable the network to selectively update and forget information in the hidden state. A useful resource that provides a good introduction to recurrent neural nets is [Schmidt, 2019].

2.4 Deep Reinforcement Learning

In deep reinforcement learning, the concepts of deep learning and reinforcement learning are combined as the name indicates. This is often done by estimating different relevant quantities in RL, such as q-values with neural networks. The motivation for doing this is that keeping a large table of state-action values is not always feasible. Say that the state space, $\mathcal{S}$, consists of images of high resolution, then the size of this state-space is simply too large to keep track of a q-value for each image and each action. A method to approximate the q-values is needed.

Deep Q-Networks

The deep Q-network (DQN) is one of the first deep reinforcement algorithms developed and the intuition behind it has already been briefly outlined. The idea is to pass the state through a neural network to obtain the q-values for each action. An example of this is found in fig. 2.9 where a neural network is used to approximate the q-values of three different actions, $a_{1:3}$ to a state, s. DQN builds upon the off-policy Q-learning algorithm de-

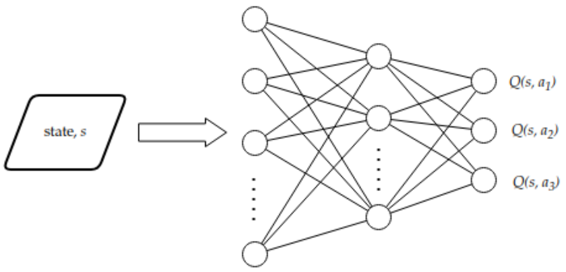


Figure 2.9 When using a deep Q-network the state, s is passed through a neural network with equal number of output neurons as there are possible actions. In this case the problem allows for 3 actions, a_1, a_2 and a_3. Hence, the network outputs three q-values; $Q(s, a_1)$, $Q(s, a_2)$ and $Q(s, a_3)$.

scribed in section 2.2. The update rule for the q-values in (1-step) Q-learning is:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big(r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\Big) \tag{2.34}$$

where the temporal difference error is defined by the expression in the large parenthesis:

$$\delta = r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{2.35}$$

To optimize the deep Q-network, a convex function (such as MSE (2.7) or MAE (2.2)) of δ in 2.35 should be minimized. The objective function for DQN can then be described by 2.36

$$\mathcal{L}_{DQN}(\boldsymbol{\theta}_p) = \Big(r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \boldsymbol{\theta}_t) - Q(s_t, a_t; \boldsymbol{\theta}_p)\Big)^2 \tag{2.36}$$

where $\boldsymbol{\theta}_\cdot$ are the parameters for the deep Q-network. Note that there are two different subscripts on the network parameters, $\boldsymbol{\theta}_p$ and $\boldsymbol{\theta}_t$. These two subscripts stand for *target* and *policy*. Two different neural networks, the target net and the policy net both approximates the q-value. The policy net is updated by optimizing the loss function in 2.36 with some optimization algorithm (e.g one from 2.1) while the target net is set equal (hard update) to the policy net every C steps. The idea of doing this was proposed by [Mnih et al., 2015] and is an idea to mitigate divergence during training. A further refinement of this method, proposed by [Kobayashi and Ilboudo, 2021] is to not do a hard update but a soft update, meaning; every C steps the target net is updated according to the following rule:

$$\boldsymbol{\theta}_t \leftarrow (1-\tau)\boldsymbol{\theta}_t + \tau\boldsymbol{\theta}_p \tag{2.37}$$

i.e a weighted average is taken between the two networks and this can further improve training stability. Both τ (soft update-parameter) and C (how often to update the target network) are hyper-parameters which can have a great impact on learning.

It is known that reinforcement learning with non-linear function approximation can be unstable and even diverge, this was shown by [Tsitsiklis and Van Roy, 1997]. The cause for this is in part the fact that observations are often obtained sequentially which implicates temporally dependent datapoints, and in part because of the correlation between the estimate ($Q(s_t, a_t)$) and the target ($r_t + \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$). Using a target network addresses the second problem. The previously mentioned paper [Mnih et al., 2015] does however propose a solution to the first problem as well. By using an *experience replay*, which is common practice today when training DQN:s, it is possible in theory to avoid training on temporally correlated data. The experience replay is a buffer where (*state*, *action*, *reward*, *next state*)-tuples (also called *experiences*) are stored. During training, when exploration of the environment occurs, the new experiences are stored in the replay buffer and when the network parameters are optimized, a batch is sampled uniform at random from the experience replay. Ideally, this replay buffer should contain as much data as possible but due to memory constrains, this might not be possible. Since the experiences from early episodes in the training session are in the replay buffer longer, there is a slight bias towards those observations. This effect can be corrected for by pre-filling the replay buffer with experiences. The full DQN-algorithm can be seen in algorithm 5.

Double DQN

Double DQN is a variant of DQN which aims to assist in some issues that can arise with the standard-DQN algorithm. Double DQN was proposed by [Hasselt et al., 2015] and is an extension of the Double Q-learning algorithm (tabular method) [Van Hasselt, 2010] which used linear function approximators for the Q-values. Standard Q-learning have a tendency to overestimate the Q-values. [**overestimation**] showed that every Q-value is overestimated up to a certain upper bound and that the overestimation can lead to suboptimal policies. Their assumption for their mathematical results are that

Algorithm 5: DQN Training Algorithm

```
Data:
  $Q(\cdot,\cdot;\boldsymbol{\theta}_p)$ : Initialized policy network
  $R$ : Empty or pre-filled replay buffer
Result:
  $Q(\cdot,\cdot;\boldsymbol{\theta}_p)$ : Trained policy network

$\boldsymbol{\theta}_t \hookleftarrow \boldsymbol{\theta}_p$    // Initialize target network parameters;
$t_{global} \hookleftarrow 0$    // Initialize time step;
for episode = 0, 1, 2, ... do
    Initialize environment and observe $s_0$;
    while episode not finished do
        // Using $\epsilon$-greedy policy;
        if Random(0, 1) < $\epsilon$ then
            Select random action $a_t$;
        end
        else
            $a_t = \arg\max_{a_t} Q(s_t, a_t; \boldsymbol{\theta}_p)$;
        end
        Take action $a_t$ and observe reward $r_t$ and next state $s_{t+1}$;
        Store $(s_t, a_t, r_t, s_{t+1})$ in $R$;
        Sample $(s_j, a_j, r_j, s_{j+1})$ uniform at random from $R$;
        Update $\boldsymbol{\theta}_p$ by taking gradient of $\mathcal{L}_{DQN}$ in 2.36;
    end
    // Update target net every $C$ episodes;
    if $t_{global}\%C = 0$ then
        $\boldsymbol{\theta}_t \hookleftarrow (1-\tau)\boldsymbol{\theta}_t + \tau\boldsymbol{\theta}_p$;
    end
    $t_{global} \hookleftarrow t_{global} + 1$;
end
```

the Q-values have a random, uniformly distributed error. [Hasselt et al., 2015] expands on this result and show that any kind of estimation error can implicate an upward bias. The error may come from the environment, function approximation or any other source. This in an interesting result for this thesis as various estimates are used in many parts of the system, e.g the environment.

To mitigate overestimation [Van Hasselt, 2010] suggested to use a different set of parameters to approximate Q-values during action selection and action evaluation. This means that the first term in eq. 2.36, which often is called the *target* is modified from:

$$Y_t^{DQN} = r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \boldsymbol{\theta}_t)$$

to:

$$Y_t^{DoubleDQN} = r_t + \gamma Q(s_{t+1}, \arg\max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \boldsymbol{\theta}_t'); \boldsymbol{\theta}_t) \tag{2.38}$$

It can be seen in 2.38 that one set of parameters ($\boldsymbol{\theta}_t'$) is used for action selection while another set of parameters ($\boldsymbol{\theta}_t$) is used for evaluating the action. In the original double Q-learning paper [Van Hasselt, 2010] one out of the two sets of parameters are selected at random to be updated at each optimization step. The roles of $\boldsymbol{\theta}_t$ and $\boldsymbol{\theta}_t'$ can also be switched at random time steps. [Hasselt et al., 2015] shows that the benefits of double Q-learning over standard Q-learning translates to the non-linear function approximator-case with double DQN. They test their methods on six different Atari-games and see improved policies.

2.5 Data Augmentation

Data augmentation is the concept of transforming or perturbing data such that new training examples are created. It can be applied to different types of data such as speech, text or mechanical signals but in this section it will be discussed in the context of images.

Deep neural networks require a large amount of training examples in order to perform well. This can be problematic when labeled data in the desired domain is scarce. Training on a small dataset during many epochs is generally a poor solution as the model is likely to overfit and not generalize well to new, unseen data. This is a good context to employ data augmentation. A (too small) dataset of size N can by augmenting each training example once be increased to a size of $2N$. Augmenting data by adding noise can also be a good strategy to make a model more robust to adversarial examples and noisy data. Applying data augmentation has been a known technique to reduce overfitting as long as convolutional neural networks have been around. [Lecun et al., 1998] is an early of example where data augmentation is employed to reduce overfitting on the MNIST dataset.

Common augmentations for images are rotating, flipping, cropping, adding Gaussian noise or slightly moving the bounding boxes (specific to object detection) in the images. A more detailed explanation of the different augmentations used in the experiments in this thesis can be found in the next chapter.

There exists other types of data augmentation besides the previously mentioned affine transformations. [Perez and Wang, 2017] use Generative Adversarial Networks (GANs) to do something called style transfers. This means a neural network modifies the images such that a different style is applied to the image. Different styles can mean almost anything, the weather can for instance be changed in the image. The GAN can also be trained to alter images such that they look like the work of a certain artist. The techniques involving generative models are however easier to use for image classification where one can assume that the label of the augmented example stays the same. For object detection however, there can of course be multiple objects, with different labels and different locations. The task of generating augmented images while still preserving quality of bounding boxes is a more difficult task. This thesis will not consider augmentations involving a GAN or any other types of generative models. It will focus on how to chose between the traditional augmentations described earlier.

2.6 Evaluation Metrics

Evaluation metrics are metrics that are used to understand the performance of a model for a certain task. In this section, some evaluation metrics for object detection will be explained but also metrics that will be used to evaluate reinforcement learning agents.

Object Detection & Classification Metrics

Two initial, important concepts to understand are *precision* and *recall*. These two metrics are used in classification to understand the general performance of a model as well as what kind of errors the model tends to do. Precision and recall are defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.39}$$

and

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.40}$$

a table explaining the abbreviations in 2.39 and 2.40 can be found in table 2.1. Depending on the task, it can be more important that the model is optimized for one of these metrics compared to the other. It is common to optimize for precision if false positives are expensive. Say a hypothetical task is to predict if a patient needs surgery or not and that the procedure can be dangerous and expensive. In this case it is important that when the model predicts that the patient needs surgery, then this is actually the case. When optimizing for recall instead, the false negatives are more expensive than the false positives. An example of this is fraud detection, where it might be devastating to miss a fraud.

Table 2.1 Table explaining true/false positives/negatives. If the task is to classify whether an image contains a cat or not, a positive classification would be to predict "Cat" while a negative would be to predict "Not Cat".

Abbreviation	Full Name	Explanation
TP	True Positive	Labels correctly classified as positive
FP	False Positive	Labels incorrectly classified as positive
FN	False Negative	Labels incorrectly classified as negative
TN	True Negative	Labels correctly classified as negative

One could also construct the precision-recall curve in which recall is plotted on the x-axis and precision is plotted on the y-axis. The points on the curve are determined by varying the prediction threshold. A model outputs a probability between zero and one, and a common threshold to use is 0.5, meaning; if the prediction is above 0.5, then a positive prediction is made. This curve can help to understand the model performance as well as picking a suitable threshold. The area under the curve is another important metric called *average precision* (AP).

Remember, object detection is not solemnly about classification, but also localization. The model is expected to output bounding boxes, indicating where the items are. A metric to measure the quality of bounding boxes is *Intersection over Union* (IoU). This is the ratio between the area of the intersection and the area of the union of the predicted and the true bounding box. The IoU is often combined with AP, precision and recall by requiring a threshold of the IoU. An IoU-threshold can for instance be set to 0.5 and then positive examples are only counted when the IoU is at least 0.5. If AP is averaged over all IoU thresholds and all classes, the metric *mean Average Precision* (mAP) is obtained. If a number between 0 and 100 is added after mAP, such as in mAP50, then this is the average precision (over all classes) where the IoU is at least 0.5. This is hence a metric of both bounding box quality and classification performance which is suitable for object detection.

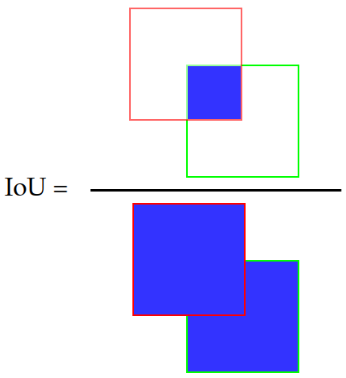


Figure 2.10 Intersection over Union is a metric for the quality of a bounding box. The area of the intersection of the two boxes is divided with the area of the union. IoU is a value between 0 and 1 where 1 corresponds to a perfect bounding box.

Reinforcement Learning Metrics

A common evaluation metric for reinforcement learning agents is to look at the average discounted return over many episodes. The discounted return is the discounted sum of all rewards collected during an episode and was introduced in equation 2.20. The return and reward can be more or less interpretable depending on the task (although it should always be as high as possible) and in case of a less interpretable reward signal, the introduction of a baseline can be of benefit. The reward signal from the environment can be noisy and in that case a comparison between agents can be preferable. Other interesting metrics to monitor are the average episode length and histograms over selected actions. These metrics can reveal the behaviour which the agent exhibits in the environment.

3

Methodology

This chapter will describe how the deep reinforcement learning framework is developed, how it is trained and tested as well as how the behavior of an agent is investigated. Reinforcement learning problems can, as mentioned in section 2.2, be framed as an interaction between an agent and an environment (see fig. 2.3). Hence, the chapter will be structured according to the main components of reinforcement learning. Firstly, an overview of the work flow will be presented. Secondly, the design of the environment will be discussed. Thirdly, the design of the agent and how the agent and environment are interacting with each other is presented. With this in mind, a mathematical model inspired by the framework of POMDP is set up. A thorough description of methods for evaluating the agent will also be provided. Table 3.1 describes some important terminology which is used throughout this and oncoming chapters.

Table 3.1 A table of used notation describing various components of the reinforcement learning framework. This notation will be used in the following chapters.

Concept	Explanation
Detector	The object detection model we are trying to optimize with good augmentations.
Childnet	An object detection model used to calculate the reward signal in the environment which has been trained for several epochs.
Backbone	A pre-trained convolutional neural network used to extract features from an image. It is pre-trained on ImageNet which is a classification dataset.

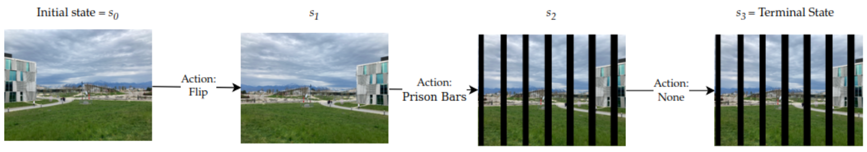


Figure 3.1 An example of an episode. The agent receives an image from a dataset and decides how to augment the image. Initially, the agent chooses to augment the image with a Flip augmentation. The agent then has to decide whether it wants to further augment the image. It chooses to do the Prison Bar augmentation so that the image now is both flipped and decorated with black stripes. After that, the agent chooses to not do anything and this action terminates the episode.

3.1 Overview of Methodology

An agent will be trained to pick good augmentations for images. This will be done with a reinforcement learning algorithm (deep Q-learning, Algorithm 5) and for this reason an environment will have to be designed.

A crucial part in the problem set-up is deciding how an episode is defined. An episode is the event of augmenting a single image. A step in an episode is equivalent to applying one augmentation to that image. Termination of an episode occurs either when the agent has taken a pre-determined number of actions or when it decides for itself that the best action is to not augment the image anymore. A visualization of a potential episode can be seen in fig. 3.1. The environment will see actions (augmentations) selected by the agent and provide a new state and a reward. The reward will be calculated with the help of a neural network called the "Childnet" (see Table 3.1). This network has the same architecture as the object detection model that we are trying to improve (the "Detector", see Table 3.1). The role of the childnet is to simulate how a detector behaves during training.

When an agent has finished training, it can be used to augment a dataset which is used to train a detector. The quality of an agent can be determined by comparing performance metrics (e.g. mAP50) for a detector trained with augmentations done by the agent and a detector trained with some other augmentation strategy.

In the next sections, we will describe how the different component have been developed. There are many different variations of environment and agent designs that are possible to pick.

3.2 Designing the Environment

Given an action provided by the agent, the environment is supposed to output a new state (the next state) and a reward signal. When designing an environment, three different entities needs to be considered; the state space, action space and reward signal. All three of these entities can be varied with different combinations of design choices and it is difficult to know what provides the most efficient learning in the end. Reinforcement learning algorithms are often benchmarked on computer games (see e.g. [Bellemare et al., 2013]) since there often exists a score which can be used as a reward function. Games also often define a clear set of allowed actions. The performance of the agent is simply defined by its ability to achieve high scores in the game. However, in a real world application such an obvious reward signal rarely exists. In this thesis, the objective is to maximize the performance (e.g. mAP, see section 2.6) of an object detection model through improved data augmentation. Unfortunately, designing a reward function which promotes good augmentation is not straight-forward.

Reward Function

In the computer games found in the Arcade Learning Environment [Bellemare et al., 2013] a common strategy for reward function design is to let the agent play the game and when it is finished, the agent receives a reward proportional to the accumulated score (although scores can certainly also be received during game-play). A naive approach to reward function-design for our augmentation task, analogous to this would be to:

1. Let the agent augment the training dataset.
2. Train detector on the training dataset.
3. Evaluate detector on the test dataset.
4. Use an evaluation metric for the detector as reward signal to the agent.

The issue with this strategy is that a good detector can take several days to train and this full detector-training would only account for one training example for the agent. The agent will also contain a neural network which will require thousands of training examples to learn anything useful. Evidently, the required training time with this approach would extend over hundreds of master thesis-projects. There are several other issues as well with this approach, this naive example should demonstrate that the main metric which should be improved (mAP for a detector) will be difficult to use explicitly (compared to many computer games). Instead, good heuristics of what defines good data augmentation have to be found.

During the course of this project, three distinctively different ways of calculating the reward signal have been used. These three algorithms will be called TrainLoss, GradientNorm and EvalLoss.

TrainLoss Reward Function. In the TrainLoss reward function we use the heuristic that if the training loss of the Childnet (see 3.1) on an image is high then the neural network learned a lot from that training example. The hypothesis was that, as the images are still in the same domain, this would mean the detector would also learn a lot from that example. Thus, the larger the training loss, the higher reward signal yielded to the agent. As will be seen in the Results chapter, this reward function did not end up working particularly well. The TrainLoss algorithm is found in algorithm 6. The al-

Algorithm 6: TrainLoss Reward Function

```
Data:
    $x_t$ : Image after the previous augmentation
    $y$ : Ground truth
    $L_{t-1}$ : Loss on the image before the previous augmentation
    $C(\cdot)$ : Childnet
    $\mathcal{L}(\cdot)$ : Loss Function for object detection.
    $f$ : Monotone and increasing function
Result:
    $r$ : Reward

$\hat{y}_t \hookleftarrow C(x_t)$    // Make prediction;
$L_t \hookleftarrow \mathcal{L}(y, \hat{y}_t)$    // Calculate loss;
$r \hookleftarrow f(L_t - L_{t-1})$    // Compare with previous loss;
```

gorithm takes an augmented image as input, makes a prediction with the childnet and calculates the loss of this prediction. This loss is compared to the loss of the image before the most recent augmentation was applied. Since $f(\cdot)$ is an increasing function (for example a simple linear function $x = y$ or the sigmoid function as in eq 2.30), the reward is high if the childnet received a higher loss after the augmentation was made, indicating it was a more difficult (and perhaps more useful) training example. An apparent risk with this reward function is that it might promote the agent to destroy all training examples by applying as many and as difficult augmentations as possible. This is a risk since training examples of very poor quality probably yields a high loss from the childnet.

GradientNorm Reward Function. The GradientNorm reward function does not utilize the training loss of the childnet, instead it regards the gradient of the loss function with respect the childnet's parameters. The idea behind this is that if the norm of the gradient is large. This means the rate of change in the childnet's network parameters would be large and this might indicate that the childnet would learn a lot from that training example. The algorithm for GradientNorm can be found in algorithm 7. It takes an aug-

Algorithm 7: GradientNorm Reward Function

```
Data:
    $x_t$ : Image after the previous augmentation
    $y$ : Ground truth
    $N_{t-1}$ : Norm of image gradient before the previous augmentation
    $C(\cdot;\boldsymbol{\theta})$ : Childnet
    $\mathcal{L}(\cdot)$ : Loss Function for object detection.
    $f$ : Monotone and increasing function
Result:
    $r$ : Reward

$\hat{y}_t \hookleftarrow C(x_t;\boldsymbol{\theta})$    // Make prediction;
$N_t \hookleftarrow ||\nabla_{\boldsymbol{\theta}}\mathcal{L}(y,\hat{y}_t)||$    // Calculate gradient norm;
$r \hookleftarrow f(N_t - N_{t-1})$    // Compare with previous gradient norm;
```

mented image as input and the previous norm of the gradient of the loss function w.r.t. the childnet's parameters, $\boldsymbol{\theta}$. Then it calculates the current gradient norm and compares this with the previous norm. Again, since $f(\cdot)$ is an increasing function, the reward function yields a high value if the size of the norm increases after the augmentation. A similar risk exists with this reward function as with TrainLoss, that the reward function will promote the agent to destroy images. This function is slightly more computationally expensive compared to TrainLoss since it requires computing the gradient of the loss, which is done with back-propagation. TrainLoss only does a forward pass through the network while GradientNorm requires the backwards pass too.

EvalLoss Reward Function. The final proposed heuristic for good data augmentation is EvalLoss. This is by far the most computationally expensive and is quite similar to the reward function proposed by [Qin et al., 2020]. In this algorithm, the childnet is trained on one image and then evaluated on a separate dataset to determine the test loss. If the test loss is low, then it is a good indication that the training example was useful. Similarly to TrainLoss and GradientNorm, a quantity is compared before and after an augmentation is applied. EvalLoss can be seen in algorithm 8. The training time of an agent is heavily dependent on the size of the dataset $\mathcal{D}_{reward}$ and it can be seen as a hyper parameter to the agent. There is a trade-off here between creating a less noisy reward signal, which intuitively would be the case with a large $\mathcal{D}_{reward}$ and the training time of the agent. $\mathcal{D}_{reward}$ is a separate dataset to the agent's training dataset (see section 3.6). The parameters for the childnet, $\boldsymbol{\theta}$ are always reset after the training step. This is to accurately measure the effect of every augmentation, which would be less

Algorithm 8: EvalLoss Reward Function

```
Data:
    $x_t$ : Image after the previous augmentation
    $y$ : Ground truth
    $L_{t-1}$ :
Validation loss on the image before the previous augmentation
    $C(\cdot;\boldsymbol{\theta})$ : Childnet
    $\mathcal{L}(\cdot)$ : Loss Function for object detection.
    $f$ : Monotone and increasing function
    $\mathcal{D}_{reward}$ : Dataset
Result:
$r$ : Reward

$\hat{y}_t \leftarrow C(x_t;\boldsymbol{\theta})$    // Make prediction;
$\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$    // Save Childnet parameters;
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\nabla_{\boldsymbol{\theta}}\mathcal{L}(y, \hat{y}_t)$    // Update childnet parameters;
$L_t \leftarrow 0$    // Initialize total validation loss;
for $x \in \mathcal{D}_{reward}$ do
    $\hat{y} \leftarrow C(x;\boldsymbol{\theta})$;
    $L_t \leftarrow L_t + \mathcal{L}(y, \hat{y})$;
end
$L_t \leftarrow \frac{L_t}{|\mathcal{D}_{reward}|}$    // Get average validation loss;
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}'$    // Restore childnet parameters;
$r \leftarrow f(L_{t-1} - L_t)$    // Compare with previous loss;
```

clear if the childnet progressively got better (or worse) during training of an agent.

State Space

The state space can be configured in multiple ways. It is desired to provide the agent with an accurate representation of the state while keeping the state space small since this affects the size of the neural networks which will be used. The neural networks will be kept as small as possible because smaller networks generally require less data. This will be necessary since the training algorithm will be significantly slower compared to algorithms used in supervised learning. The implication being, far fewer training examples can be seen per time unit.

There are two pieces of information that will be included in a state for this task.

- The image itself.

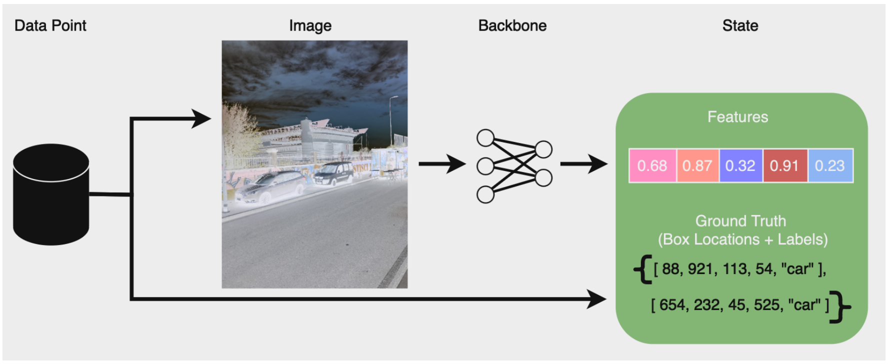


Figure 3.2 Representation of a state to the agent. The datasets used contains images and the ground truth, which is a list of vectors containing the bounding box locations and the labels. The image is passed through a pre-trained neural network (the backbone) and the extracted feature, together with the ground truth constitutes the state presented to an agent.

- The ground truth, which is a list of pixel coordinates for the bounding boxes, along with the label.

Most of the images in the used datasets are of high resolution which will require quite a large neural network to process. In the spirit of keeping the neural networks small, we decided to use an observer model in the environment which extracts features from the image. This is then treated as the observation of the state. The observer model will be referred to as the "Backbone" (see table 3.1) and is a quite large convolutional neural network which is pre-trained on the ImageNet dataset [Russakovsky et al., 2015]. The backbone is not a part of the agent as it is never trained. It is also possible to select how many layers of the pre-trained network should be used, using more layers generates a feature space of smaller dimension. A schema of how a state is created can be seen in fig. 3.2. In none of the experiments which will be done is the agent being presented with the full image, it's always features extracted from an image. However for the sake of clarity in this text, the features extracted from images may also be referred to as images, since "features" is a quite abstract notion and it may not always be clear what is meant with this.

During some experiments, the state space have been defined to include an additional piece of information: The previous actions during the episode. This means the agent is also shown what it has done previously. This information can be useful due to some of the augmentations which are used (see table 3.2). Some augmentations, such as "Flip", does not alter the colors in the images, it just flips the image on the horizontal axis. Since flipping an image two times results in the same image, it would be more clever to not do an augmentation to begin with. Without also providing the agent with its previous actions, one cannot expect the agent to learn this relationship.

Action Space

The action space of the agent is the set of actions it can choose to take. These actions will be different kinds of augmentations that can be applied to images. A table of augmentations and their explanation is provided in table 3.2. Visual examples of what the augmentations do to an image can be found in Appendix A. Most of these augmentations have one or several parameters

Table 3.2 Table of augmentations which can be used by an agent. This defines the action space.

Augmentation	Explanation	Parameters
Rotate	Rotates an image by a random number of degrees	Max Rotation, Min Rotation
Flip	Flips the image along the horizontal axis	—
Gaussian Noise	Adds Gaussian Noise	Mean and Std. Dev.
Arithmetic	Changes the hue, saturation and lightness	Max Hue Difference, Max Contrast Factor, Min Contrast Factor, Max Brightness Difference, Max Saturation Factor, Min Saturation Factor
Prison Bars	Adds vertical black bars add regular intervals	Bar width
Box Jitter	Slightly changes the bounding box locations	Max Difference
Motion Blur	Gives the image the appearance of being captured at speed	Kernel Size
None	Does nothing to the image	—

which defines the severity of the augmentation. With Gaussian noise for instance, one have to define the mean and standard deviation of the added noise. Setting a large mean will lead to an image where the motive is harder to recognize. Almost every augmentation have a random component to it, and this is common as the point of augmenting images is to introduce variations to the dataset. This property of the augmentations will be kept and this means that when the agent decides to take action "Arithmetic" for instance, then the agent will not know exactly how much this will change the image.

Since the augmentations take different parameters which changes the severity of it, many different actions spaces can be configured. One action can for instance be Gaussian noise with a high mean and variance while another action is noise with a much lower mean. In our experiments, the action space will consist of 20 actions, each augmentation in table 3.2 is used three times (except for Flip and None which does not have any severity parameters). Each augmentation with a low, medium and high severity setting. A hypothesis is that the random nature of the augmentations can make learning harder for the agent and specifying the action space into more detail might mitigate this effect.

The "None" action is treated a bit differently compared to the others. If the agent chooses "None", then the episode is terminated, the agent will not be able to augment that image anymore. The agent is allowed to make several augmentations to an image and the number of allowed augmentations per image can be seen as a hyper parameter. Thus, taking the "None" action means that nothing more should be done to the image.

3.3 Mathematical Model

Given the above explanations and assumptions made on the environment, agent, and their interaction, the problem can be modelled with a POMDP in mind. As described from section 2.2, a POMDP can be defined as a 7-tuple $(\mathcal{S}, \mathcal{A}, P, R, \Omega, Z, \gamma)$. From 3.2, the latent state space $\mathcal{S}$ includes any arbitrary high resolution RGB image, which is considered as a continuous space. As discussed there, the dimensionality of this space is large, and in order to reduce this problem an observer in the form of a pre-trained neural network (the backbone, see 3.1) is used. In addition to this NN, a function that takes the finite history of the actions in the current episode as input and outputs a one-hot encoded representation of this action history is used. A third function that encodes the images ground truths (class labels and bounding boxes) is also used. Together these functions make up the observation function $Z : \mathcal{S} \times \mathcal{A} \to \Omega$, where Ω denotes the latent observation space that the agent can observe. The latent observation space is then defined by combining the outputs of these three functions - the backbone neural network, action history encoding function, and ground truth encoding function. By letting the 3-channel input image be a tensor with shapes (d_1, d_2, d_3), the backbone down-samples such an image into an (d'_1, d'_2, d_3)

tensor, where $d'_1 < d_1$ and $d'_2 < d_3$. Denote $f_{\text{NN}} : \mathcal{S} \rightarrow \mathbb{R}^{d'_1 \times d'_2 \times d_3}$ to be the backbone neural network that processes the high-resolution RGB images, $f_{\text{action}} : \mathcal{H} \rightarrow \{0, 1\}^{d_4 \times d_5}$ be the action history encoding function, where d_4 is the cardinality of the discrete action space and d_5 the maximum number of allowed actions during a single episode. The ground truth encoding function is $f_{\text{gt}} : \mathcal{G} \rightarrow \mathbb{R}^{d_5 \times d_6}$, where d_6 are the maximum allowed ground truths we allow to encode. The latent observation space Ω can then be described as follows. Let F be the unary operator which when operates on a tensor of dimensions $(t_1, t_2, ..., t_n)$ produces a new tensor of size $(1, \prod_i t_i)$, and con be the binary operator concatenating two vectors of size $(1, t_1)$ and $(1, t_2)$ into a single $(1, t_1 + t_2)$ vector. Then the shape of the observation space follows as:

$$\Omega = \{f_{\text{NN}}(s), con(F(f_{\text{action}}(h)), F(f_{\text{gt}}(g))) \mid s \in \mathcal{S}, h \in \mathcal{H}, g \in \mathcal{G}\} \tag{3.1}$$

An arbitrary tuple in the latent observation space is then:

$$\vec{o} = (f_{\text{NN}}(s), con(F(f_{\text{action}}(h)), F(f_{\text{gt}}(g))) \in (\mathbb{R}^{d'_1 \times d'_2 \times d_3}, \mathbb{R}^{(d_4 \cdot d_5) + (d_5 \cdot d_6)}) \tag{3.2}$$

Here, $\vec{o}$ represents the final observation tuple (I_f, Tr) in Ω, which the agent uses to make decisions, where I_f are image features, and Tr are truths (encoded ground truths and encoded action history).

The action space is defined to be discrete, with $\mathcal{A} = \{A_0, A_1, \ldots, A_N\}$. Most actions are probabilistic functions on images, i.e. on the latent space off the environment. Thus these actions can be defined as mappings from a state $s \in \mathcal{S}$ to a probability distribution $\Delta(\mathcal{S})$:

$$A_n : \mathcal{S} \rightarrow \Delta(\mathcal{S}) \tag{3.3}$$

In this setting, where the framework of POMDP are used together with a model free DRL algorithm, the transition function P is learned implicitly by the agent. This makes it important to provide the agent with enough information and learning potential to extract the dynamics of the environment through experience, while balancing the dimensionality problem. An example of this is how the observation space and observation function has been constructed. Take for example the Box Jitter augmentation. It only acts on the bounding boxes of an image, so after applying this augmentation the agent cannot observe the change from image data alone, and it would not be plausible to learn the transition dynamics without the extra data from the ground truths. Another important take away is that, for the agent to be able to learn temporal dependencies, such as how an image transforms when applying an action A_n, some kind of memory of the previous states should help.

3.4 Designing the Agent

Designing the agent involves choosing a DRL algorithm and designing suitable neural network architectures. This section will also cover how the agent is used.

Selection of Deep Reinforcement Learning Algorithm

The agent will be a DDQN-Agent, the theoretical aspects of this was covered in section 2.4 and 2.4. This means that for a given image, the expected discounted return (Q-values) for every possible action will be returned by the agent. The Q-values can be used in different ways, a natural choice is to always pick the action associated with the highest Q-value. An alternative approach, which will be used in this project is to instead sample an action according to the discrete probability distribution generated by the Q-values. This approach is selected because the we believe that it is advantageous to have some degree of randomness when augmenting images. The task of the agent becomes to nudge this probability distribution towards the better augmentations.

Neural Network Architectures

The neural network in a DQN-agent is responsible for estimating the Q-values of each possible action given an observation. This means the input to the neural net will be the state which consists of extracted features from an image and the ground truth, this could be seen in fig. 3.2. Essentially there are two pieces of information which needs to be processed by the network, the image features and the ground truth and these are quite different. The image features, which are extracted from the backbone, still has over 100.000 dimensions which are organized into hundreds of channels. In order to keep the neural net small, we decide to add a few convolutional blocks which consists of convolutional filters, a non-linear function, batch-normalization layers and zero-padding operations. After letting the image features pass through a few convolutional blocks, the feature space is further down-sampled. We use enough convolutional blocks to shrink the feature space from above 100.000 dimensions to 12.800. Shrinking the feature-space further is done since we later in the network want to combine the features with the ground truth and pass all information through a common head. This head consists of a sequence of LSTM and FC layers. A graphical representation of the architecture is seen in fig. 3.3. The image features which have passed through the convolutional blocks are then flattened and concatenated with a flattened representation of the ground truth. Then this concatenated information enters the common head. The head starts with a single RNN layer. The reason for using the LSTM comes from the theory

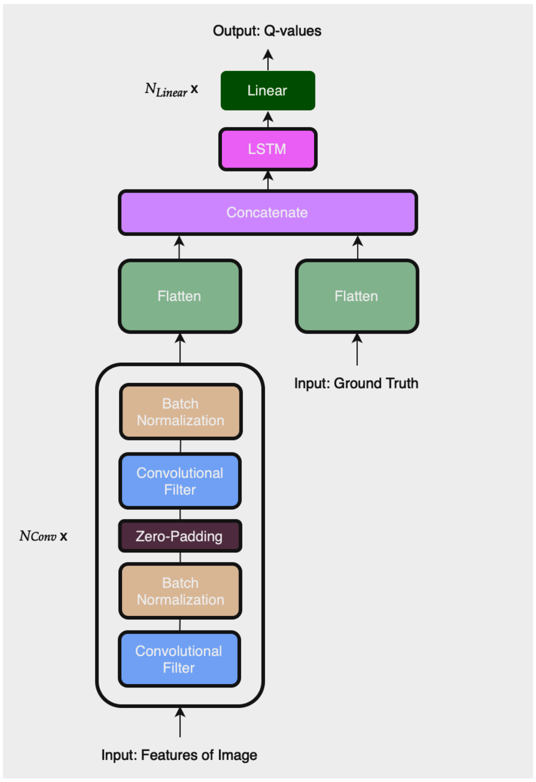


Figure 3.3 Neural network architecture for estimating Q-values used by the agent.

of histories from POMDPs in section 2.2. For the agent to to be able to infer as much information as possible from a sequence of observations, the history of actions and observations is useful. Although this is not directly implemented, as the DRL framework tries to learn this implicitly, the motivation for using recurrent units at this stage comes from equation 2.14. This should allow the agent to remember the last observation and compare it to the current one, leading to a more effective learning of the transition process. Another reason for employing the RNN is that the decision making in a single episode is sequential, as the first augmentation during an episode influences the second one, and so on. This is a domain where RNNs usually excels.

3.5 Agent Training Algorithm

In this section, the relationship and interaction between the environment, which was described in section 3.2, and the agent, described in section 3.4 is going to be discussed. The agent will be trained with the DQN Training Algorithm (Algorithm 5) and this section will explain how the components designed according to section 3.2 and 3.4 relates to this algorithm.

An episode begins with observing the initial state, s_0, in this problem setting this means to fetch an image and its ground truth from a dataset. Given this data point, an action is selected with the ϵ-greedy policy. This means that with a probability of ϵ $(\in [0,1])$ a random action is selected, otherwise the agent outlined in 3.4 selects the action. The selected action is communicated to the environment so that it can calculate the reward according to Algorithm 6, 7 or 8 and prepare the next state by passing the image through the backbone to extract features. When the reward and next state have been calculated, a transition tuple (s_t, a_t, r_t, s_{t+1}) can be constructed, this tuple is stored in the replay buffer. The parameters of the DQN is then optimized by sampling a batch of transitions uniformly at random from the replay buffer, calculating the loss (eq. 2.36) and taking the gradient of it. These steps are done until the episode ends which happens after a fixed and pre-determined number of augmentations or when the agent chooses the "None"-augmentation. After the episode, the target net is updated with a soft update (eq. 2.37) to more closely match the policy net. However, this is not done after every episode, we do this update every thousandth episode. When this procedure is done, a new episode is initiated by sampling a new image from the dataset and the loop restarts.

The hyperparameters and other parameters used during agent trainings can be found in Appendix B.

3.6 Data Usage

An essential part of training machine learning models is the data. In this thesis, two datasets are used, a public dataset which has been modified and curated by Axis Communications and a dataset which has been collected and curated in-house by Axis Communications. For competitive reasons, we cannot disclose that much information about these datasets but the dataset which will be referred to as the "In-House Dataset" is more definitely within the domain of surveillance images while the "Public Dataset" also contains surveillance images, although the domain is broader.

Let's denote all the data which is used during a training session, where a training session includes training the DRL agent, evaluating the agent (e.g. calculate average reward), training a detector and evaluating the detector, with $\mathcal{D}$. The dataset $\mathcal{D}$ can be a mix of the Public Dataset and the In-House Dataset or a subset of one of them. This dataset, $\mathcal{D}$ and is divided into a few, mutually exclusive datasets such that:

$$\mathcal{D} = \mathcal{D}_{train} \cup \mathcal{D}_{validation} \cup \mathcal{D}_{test}, \quad \mathcal{D}_i \cap \mathcal{D}_j = \emptyset \quad \forall i, j \in \{train, validation, test\}$$

and

$$\mathcal{D}_{reward} \subset \mathcal{D}_{validation}$$

- $\mathcal{D}_{train}$ which is used for training the DRL agent, the detector and the childnet.
- $\mathcal{D}_{reward}$ which is used in algorithm 8 for calculating the reward.
- $\mathcal{D}_{validation}$ which is for continuously evaluating the agent during training. This evaluation includes calculating statistics such as average episode return or average episode length. It is also used as a validation set during training of detectors and childnets to monitor trainings.
- $\mathcal{D}_{test}$ which is used to test the final performance of the detector which has been trained with agent augmentations.

It can be seen from the list above that the agent is trained on the same data as the detector and the childnet which is reasonable since it is this data, the data that the detector is eventually trained on, that the agent should be able to augment well. The role of the childnet is to simulate the behavior of a detector during the training of an agent, so it is natural that the childnet is trained on the same data as well.

The reward dataset, $\mathcal{D}_{reward}$ is separate compared to both the test set, $\mathcal{D}_{test}$ and training set, $\mathcal{D}_{train}$ and is much smaller in size compared to the other datasets.

3.7 Evaluating the Agent

The ultimate goal of the agent is to provide augmentations which improves the final performance of the detector. To evaluate if this is the case, a detector is trained on a dataset which have been augmented by the agent and then tested on a separate test set to reveal its true performance metrics. This detector is trained for 20 epochs [1] and each time an image is drawn from the

[1] An epoch is defined as 100.000 images in this thesis, even if the size of the dataset is different than that. This means that the number of backpropagations during a training is constant regardless of dataset size.

dataset, it has a 50% chance of being augmented or not. This means that half of the images seen by the detector during a training is augmented with the agent, and half is not augmented at all. This is a common setup when using data augmentation, there is no point in not using the original data as well. The performance metrics of the detector trained on an agent-augmented dataset will be compared to a baseline detector which is trained in a similar fashion. Instead of using augmentations proposed by an agent, the baseline detector is trained on a dataset where half of the images are augmented randomly. The random augmentation scheme applies a probability for each of the used augmentations (see table 3.2). Each of the augmentations have an equal probability of being applied, and multiple augmentations can be applied to the same image, also decided by chance. Comparing a detector trained with agent-augmentations and the baseline detector will reveal how good the agent is at augmenting images.

Establishing a Baseline

To establish a good baseline detector, several detectors will be trained. In order to detect if the agent augmentations makes any difference, we first have to establish that standard, random augmentation actually improves object detection performance. A hypothesis is that the augmentations will have a decreasing impact on performance as the size of the training dataset increases. To find a suitable dataset size, we train several detectors with different dataset sizes. For each dataset size, one detector with random augmentations is trained and one detector with no augmentations. When the results between these two detectors are compared, it will become apparent if augmenting the dataset has any impact at all. If there is no measurable difference in performance between a detector trained without augmentations and one with random augmentations, we do not think we can expect an agent to improve that much. So, finding a suitable baseline detector is really about finding a dataset size for which augmentations can have an effect. The training dataset with which a good baseline detector was found will be used as the training dataset, $\mathcal{D}_{train}$ described in section 3.6.

Intermediate Evaluation Metrics

Besides the final evaluation described above, other intermediate evaluation metrics are used to compare different agents with each other. If the settings for the environment is equal, then two agents can be compared by the average episode return for the respective agents. To examine the behavior of an agent, action histograms and heat maps will be used.

3.8 Implementation Details

To implement the suggested solution for the POMDP we have used TensorFlow (TF) [Martín Abadi et al., 2015] and their framework TensorFlow Agents (TF-Agents) [Guadarrama et al., 2018] which is specialized for reinforcement learning tasks. This proved effective as it presents solutions to many common difficulties appearing in RL implementations.

- **Flexibility and Modularity**: TF-Agents provides a highly modular and flexible architecture, allowing for easy code iteration with different algorithms and advanced agent architectures.
- **Support for Advanced Techniques**: TF-Agents provides built-in support for advanced techniques crucial for POMDPs, such as handling partial observability and integrating memory components like RNNs. This support simplified the implementation of strategies required for our problem.
- **Robust Simulation and Evaluation Tools**: The framework includes tools for simulating environments, handling the huge amount of data that is produced, and evaluating agent performance, which are essential for rigorously testing and fine-tuning our models.
- **Robust Algorithms**: The algorithms implemented by TF-Agents are well tested, meaning one less source of error in the development process.
- **Accessible Source Code**: The framework has a clearly structured and well documented source code.

Although there was a steep leaning curve to get comfortable with TF in general and the concepts of TF-Agents in particular, it proved to be worthwhile the effort.

4

Results

4.1 Establishing a Baseline

To establish a reasonable baseline, many detectors are trained with variable sizes of the datasets. For each dataset size, one detector is trained without any augmentations and one is trained where half the dataset is augmented with random augmentations. Two line plots displays the results from this

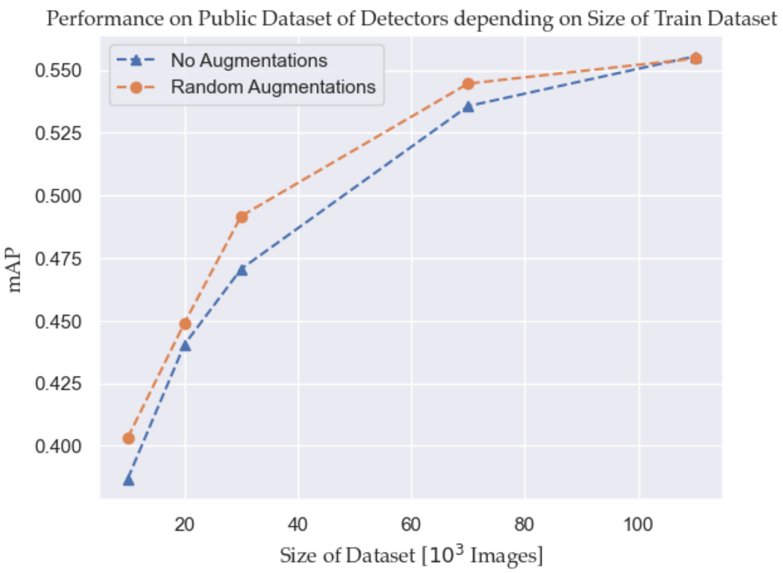


Figure 4.1 Performance on the Public Dataset of detectors which have been trained on datasets of varying size. The blue triangles are detectors trained without any augmentations. The orange circles are detectors which are trained on a dataset where half of the images were randomly augmented.

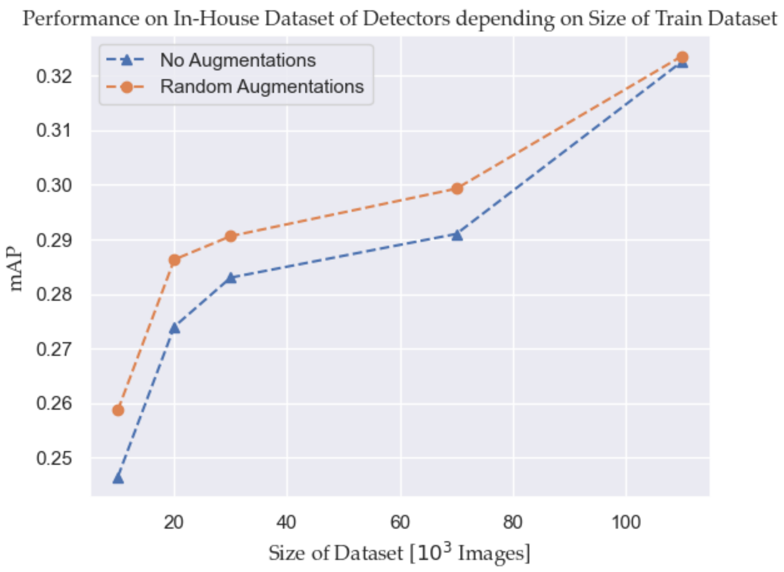


Figure 4.2 Performance on the In-House Dataset of detectors which have been trained on datasets of varying size. The blue triangles are detectors trained without any augmentations. The orange circles are detectors which are trained on a dataset where half of the images were randomly augmented.

experiment. Fig. 4.1 shows the test-mAP on the Public Dataset and 4.2 shows the test-mAP on the In-House Dataset. From this result we decide to from now on use a dataset size of 10.000 images as the difference in performance between the detectors trained with and without augmentations is large for both the Public and In-House Datasets. A smaller dataset size is favorable as it shortens the feedback loop during development as models can be trained a shorter time.

4.2 Selecting a Reward Function

To select a reward function, three agents are trained for 30.000 episodes, all conditions are equal, except that each agent is trained with a different reward function, Algorithm 6, 7 or 8. In the section 3.2, a potential issue was discussed, with 6 and 7 there could be a risk that the agent simply tries to destroy the images as much as possible. To investigate this risk, an extra action was added to the action space. In this experiment, we added a "Ruin"-augmentation which the agent could select, the augmentation turns the image completely black, making it a useless training example for a detector. After the agents are trained for 30.000 episodes, they are evaluated on

100 episodes while the actions the agents take are recorded. Heat maps over the agents' action selections are seen in fig. 4.3. This result made us choose the EvalLoss reward function for all succeeding experiments.

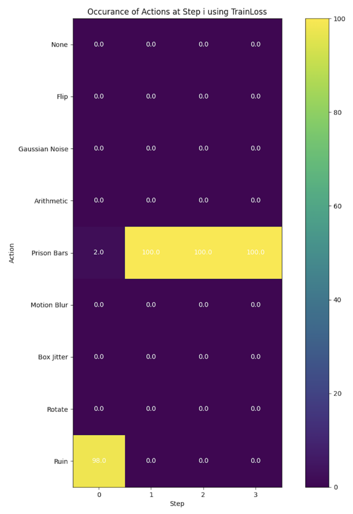
4.3 Detector Performance Results

This section discusses the performance results of object detectors using different data augmentation strategies. Table 4.1 presents a comparison between detectors trained with different augmentation and dataset configurations. Detectors trained with agent-selected augmentations/random augmentations (baseline) and on the Public/In-House Dataset. The performance is measured in terms of mAP50 on different combinations of training and testing datasets, specifically the Public Dataset (PD) and In-House Dataset (IHD). Each row in the table represents a different training-testing scenario, comparing the mAP50 results (see 2.6) of the two augmentation strategies, with 95% confidence intervals. For instance, one scenario involves training on the IHD and testing on the PD.

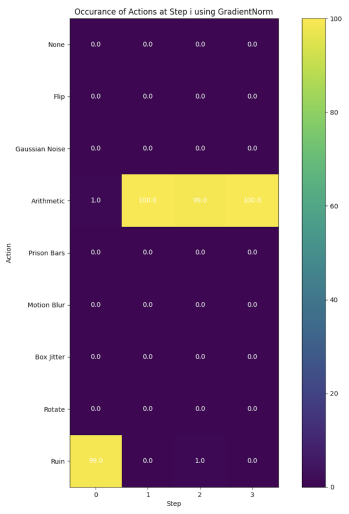
Table 4.1 Performance of Detectors with different dataset combinations. Both Public Dataset (PD) and and In-House Dataset (IHD) training sets have 10K examples. The third columns are the mAP50 results for detectors trained with augmentations made by the agent. The last column are the mAP50 results for detectors trained with random augmentations (Baseline). To exemplify, the second row are results when a Detector is trained on the IHD, but tested on the PD, with two different augmentation strategies.

Performance of Detectors			
Train Set	Test Set	mAP50 (Agent)	mAP50 (Baseline)
IHD	IHD	**0.2662** $\pm$ 0.0018	0.2602 $\pm$ 0.0033
IHD	PD	**0.1629** $\pm$ 0.0015	0.1613 $\pm$ 0.0015
PD	PD	0.4118 $\pm$ 0.0021	**0.4149** $\pm$ 0.0009
PD	IHD	**0.1629** $\pm$ 0.0015	0.1613 $\pm$ 0.0015

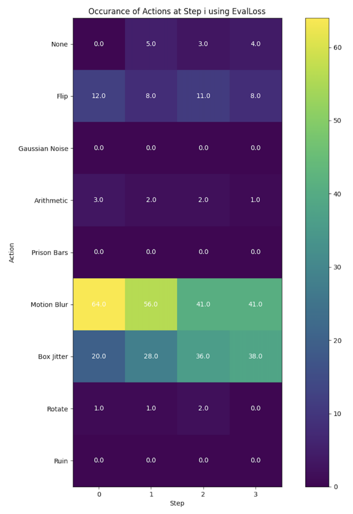
The data in table 4.1 is generated by first training two agents, one on the PD and one on the IHD. The agent trained on the PD is then used to train detectors on the PD. This generates the combination (Agent PD, Detector PD) This process is then repeated for the IHD for the combination (Agent IHD, Detector IHD). Each such combination yields two test results, one "on domain" and one "off domain". This means that a total of 2 unique combinations exist. As each such combination yields two results, giving four data points. This process was repeated 5 times for a total of 20 data points. A pairwise right-sided t-test was then performed, defining the baseline training sessions as group "Before", and the agent-augmented training sessions as



(a) TrainLoss (Algorithm 6)



(b) GradientNorm (Algorithm 7)



(c) EvalLoss (Algorithm 8)

Figure 4.3 Action heat maps for agents trained with three different reward functions (Algorithm 6, 7, 8). The heat map visualizes the occurrences of each action at each step in the episode selected by the agent. In total 100 episodes was run.

group "After". This tests if the group After is significantly better than group "Before". The result of the test can be seen in table 4.2. As the data points are paired the sample size is half the total number of data points, i.e. 20 samples (remember that we also have 20 baseline trainings). The test was performed with a significance level of $\alpha = 0.05$, and effect type Cohen's d. The result of the paired t-test indicates a significant medium difference between group "Before" and group "After", with $p = 0.0113$, and H_0 ("After" less or equal to "Before" i.e the agent augmentations does not improve the test-mAP50), the null-hypothesis is rejected in favour of the the H_1 ("After" greater than "Before" i.e. agent augmentations improve the test-mAP50). The t-value was 2.4803, which along with the significant p-value indicates a meaningful effect size. The sample size of 20 is relatively small, which could affect the reliability of the results. The normality p-value (Shapiro) of 0.5796 suggests that the data likely does not significantly deviate from a normal distribution, which validates the use of a t-test. In summary, these results indicate a statistically significant improvement with a moderate effect size.

Table 4.2 Statistical analysis results when comparing the baseline with the agent-augmentation in a right-sided paired t-test.

Metric	**Value**
Significance level (α)	0.05
P-value	0.0113
t-statistic	2.4803
Sample size (n)	20
Average of differences ($\bar{x}_d$)	0.00396
SD of differences (S_d)	0.00714
Normality p-value	0.5796

The validation loss was recorded after each epoch when training each detector, with the validation dataset as a combination of PD and IHD. The trainings were then grouped by their conditions (i.e. augmentation strategy and dataset). For each group the expected value and 95% confidence intervals were calculated via bootstrapping. The result is presented in figure 4.4. It should be noted that there is a possibility of bias in this data, as although neither the agent or detector ever gets to see this data, the environment sends a reward signal based on the validation data (but never on the test data). This potential issue is discussed in the discussion.

4.4 Policy Evaluation

The behavior of the agents on different datasets are shown in this section. To examine the behavior of the agents, we've let the agent augment images

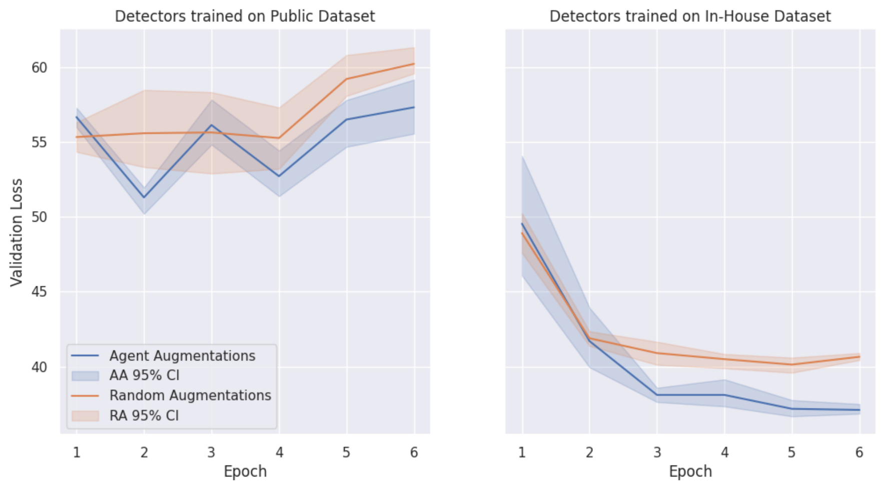


Figure 4.4 Validation Losses for Detectors trained on the Public and In-House Datasets. The blue lines are detectors trained with augmentations determined by the agent and the orange lines are detectors trained with random augmentations. The shaded areas are bootstrapped, 95% confidence intervals.

from the different datasets (the Public and In-House Dataset) and images containing different labels. With this information it is possible to determine the frequency with which the agent tends to pick augmentations in different situations. In fig. 4.5 and 4.6 the empirical action probabilities are shown in bar plots for an agent trained on 10.000 images from the Public Dataset. Fig. 4.6 also shows the severity the agent tended to pick. The figures after, fig. 4.7 and 4.8 show the same quantity for a different agent, trained on 10.000 images from the In-House Dataset. Finally, fig. 4.9 shows how the agent trained on the Public Dataset augments images which contains a specific set of labels. Each of these bar plots are constructed by running the agent for 500 episodes and counting the actions it selects. The maximum number of augmentations the agent is allowed to make on the same image is three [1].

The graphs in fig. 4.10 considers the temporal aspect, i.e when in an episode augmentations occur. Fig. 4.10 shows the transitions that never occurs when an agent trained on the In-House Dataset augments the two datasets. Consequently, this also means that all missing edges in the graphs are transitions which do occur [2].

[1] I.e an episode is at most 3 steps but can be shorter if the agent chooses the None/No Augmentation action.

[2] Apart from edges going from "No Augmentation" as this always terminates the episode.

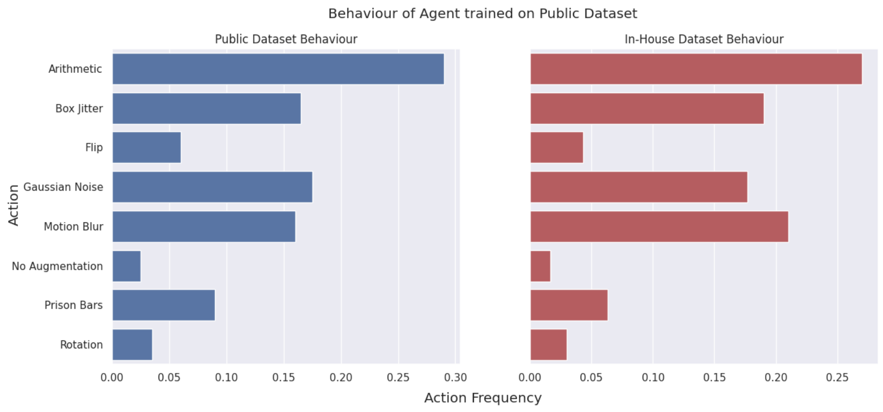


Figure 4.5 Frequency of actions selected for the Public Dataset and the In-House Dataset by an agent trained on the Public Dataset.

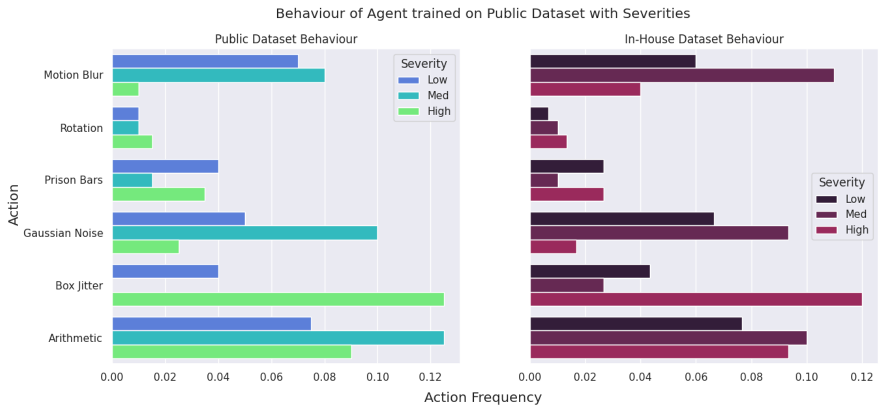


Figure 4.6 Frequency of actions selected for the Public Dataset and the In-House Dataset by an agent trained on the Public Dataset. The plot also shows which severity of the augmentation the agent tended to choose. The "Flip" and "No Augmentation" actions are not shown here as they lack a severity setting.

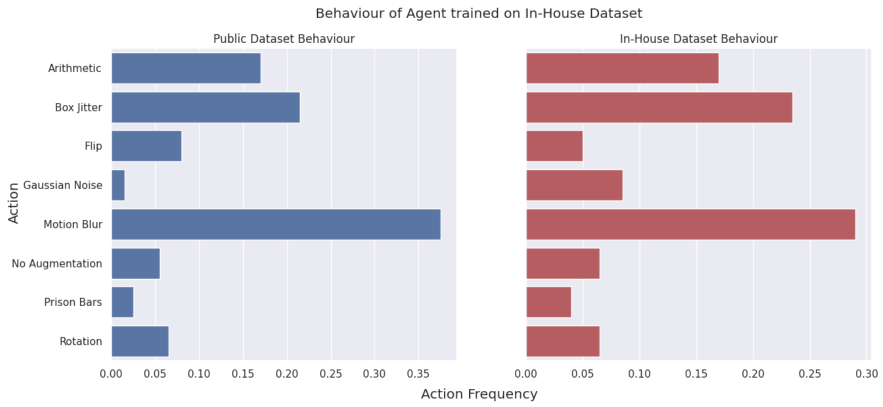


Figure 4.7 Frequency of actions selected for the Public Dataset and the In-House Dataset by an agent trained on the In-House Dataset.

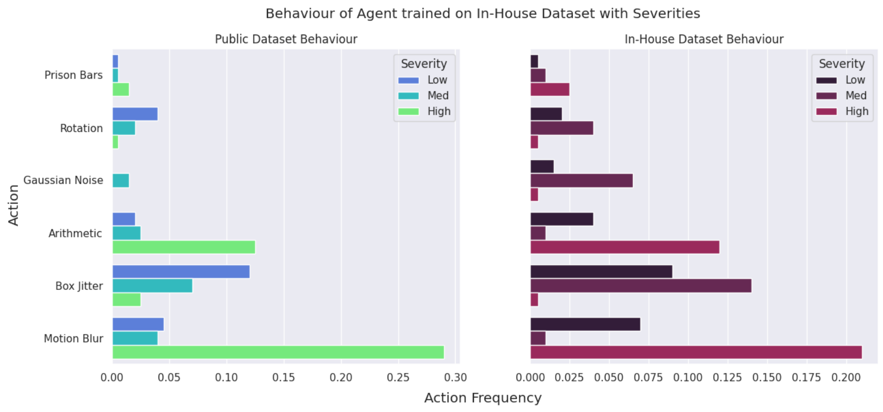


Figure 4.8 Frequency of actions selected for the Public Dataset and the In-House Dataset by an agent trained on the In-House Dataset. The plot also shows which severity of the augmentation the agent tended to choose. The "Flip" and "No Augmentation" actions are not shown here as they lack a severity setting.

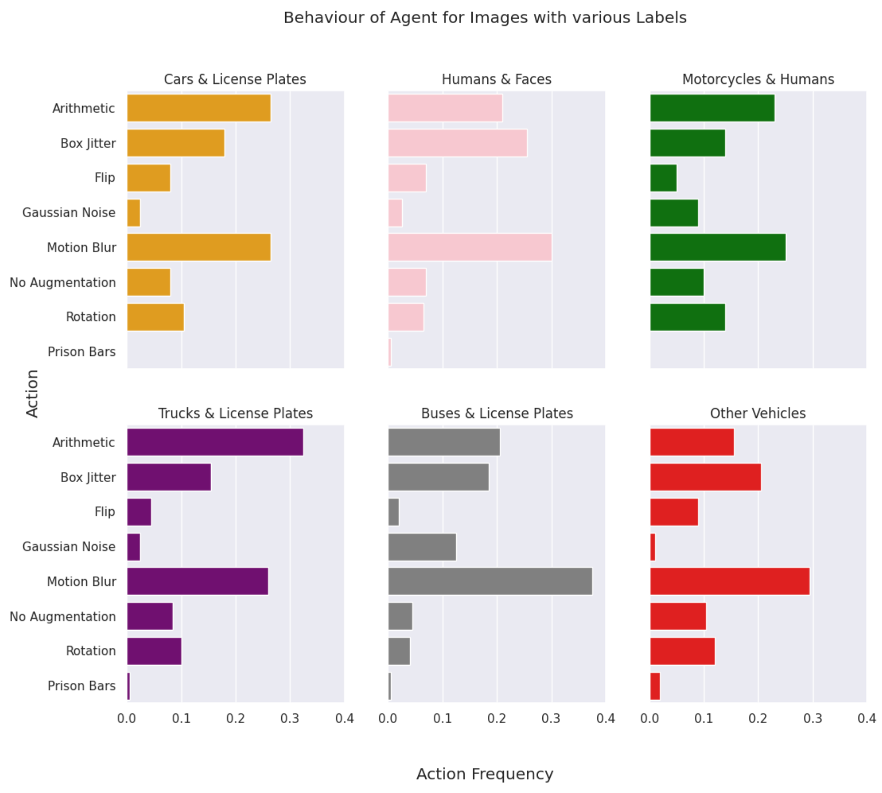


Figure 4.9 Frequency of actions selected by the agent for images containing different labels. For instance, the frequencies for "Cars & License Plates" have been calculated by letting the agent augment images that **only** contains these labels. Images are taken from both the Public and In-House Dataset. The agent which is used here is trained on the In-House Dataset.

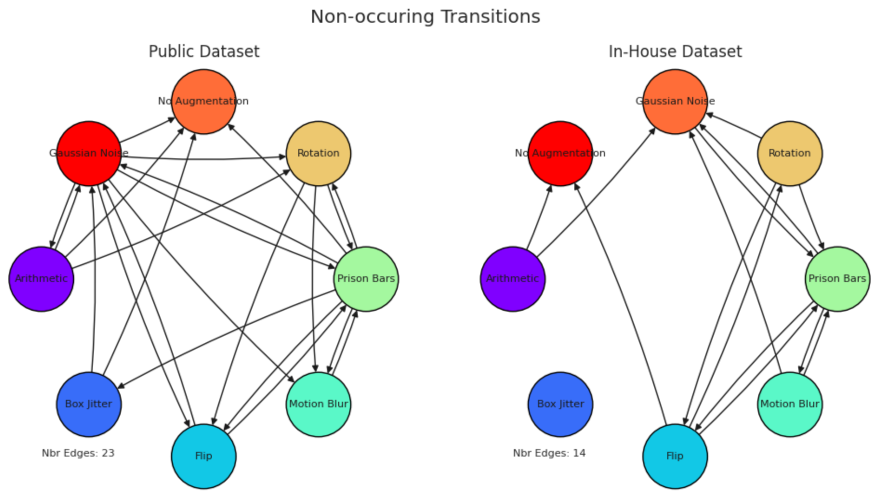


Figure 4.10 Graph of action transitions that never occur. An edge from node u to node v means that augmentation u is never followed by augmentation v. This agent is trained on the In-House Dataset.

5

Discussion

Having explored the theoretical foundations, methodologies, and empirical results of the study, we now transition to a discussion of the results. This section aims to interpret the findings, and to draw meaningful conclusions from those. Here, the choice of baseline and reward function will be discussed shortly, before moving on to interpreting the presented results.

5.1 Baseline and Reward Function

We begin by briefly motivating the choice of baseline for the empirical results, as well as why the choice of reward signal fell on the EvalLoss algorithm.

On Selecting a Baseline

The choice of baselines is crucial in contextualizing and benchmarking the performance of our proposed method. The choice of baselines is mainly motivated by two factors. To start with, the effect of augmenting the dataset should be empirically measurable. We found that when using very large datasets this was difficult, as the impact of augmenting the dataset (although randomly) had no real impact on the performance of the trained model (see figure 4.1 and 4.2). This is supported by the findings of [Yang et al., 2023], that conclude that the increase of data is not directly proportional with the performance of the model. We can only speculate about the reasons for this, but a saturation of the effective capacity of the model is reached, and that a large dataset is already diverse enough seems like plausible explanations. The second motivating factor for the use of a smaller dataset as baseline is applicability. Augmentation is often more important when the access to data within the target domain is sparse, aligning with what was previously discussed. Then comparing and using smaller datasets should be more interesting to the research community as well as to the industry.

The choice of baseline should thus both increase measurability and interest in the results. This is why it was decided to use datasets with only $10K$ examples.

Another question is about what augmentation strategy should employed for the baseline. One could approach this by using "expert made" augmentation strategies, where a human with experience has set the best known probabilities for each augmentation action. We refrained from doing so because of a few reasons. One reason is that such knowledge is not generally known, and a core reason for this thesis is to find out if different domains require different augmentation policies, and when the image domain of interest is not represented among the common benchmark datasets in the research community, such experience is hard to come by. Furthermore it can be argued that a uniformly random policy is the most general policy, as it has maximum entropy and should be less sensitive to different domains, hopefully allowing for easier reproducibility of the results we present.

Choice of Reward Function

The design of the reward function is a fundamental aspect of any reinforcement learning system, directly influencing the learning process and outcome. This is generally a difficult design process, as the reward signal needs to be transparent to the agent to infer information from it, provide the desired guidance for the agent and be computationally efficient. A badly designed reward signal can have unsuspected consequences, such as the agent learning unwanted shortcuts to a high reward. This complication was evident from our initial experiments. The constructed reward signals from TrainLoss (algorithm 6) and GradLoss (algorithm 7) were specifically designed with Hard-Example-Mining as described by [Shrivastava et al., 2016], as inspiration. These were considerably less compute heavy, with about half the training time needed compared to agents using the EvalLoss reward design (algorithm 8). However, this essentially taught the agent the wrong thing, as it gained reward from ruining the images, evident from the experiments presented in figure 4.3 where it is shown that the agent always chooses the action that destroys the image. Both signals taught the agent an unwanted shortcut to gain high rewards. In these early experiments the EvalLoss shoved much more promise for correct, although compute heavy, learning. The conclusion from this was to leave the ideas of Hard-Example-Mining influenced learning strategies, and instead focus on the more direct EvalLoss strategy. The hypothesis prior to developing the TrainLoss and GradientNorm reward functions was that both would yield poor augmentation policies. This was also the case. It was decided to try them anyway as they would be slightly easier to implement compared to EvalLoss. This facilitated the development of other infrastructure in the project.

5.2 Detector Results: Agent Augmentation or Random Augmentation

A critical assessment of the proposed method is of course to evaluate and compare detectors trained on agent-driven augmentations with detectors trained with uniformly random-driven augmentations. The random-driven augmentation strategy forms the baseline, as discussed above. One of the main results from this thesis is presented in table 4.1 , together with the analysis that showed a statistically significant improvement in mAP50 when using our proposed method over the baseline. It can also be seen that our method outperforms the baseline in three out of the four categories. One should however be cautious in generalizing these results too optimistically just yet, as the sample size is small. There are mainly two things needed to strengthen the significance of the results. The first is simply to collect more samples. The other is to train more agents, as there could potentially be a large variation in the results of the training of the agents. As time and compute were limited, we have simply picked the first agent we trained on each dataset, which of course certainly introduces an element of randomness that might skew the results. To mitigate this, future research should focus on training a larger number of agents on each dataset. This will allow for a more robust evaluation of the agent-driven augmentation approach, as it reduces the likelihood that the observed performance is due to the peculiarities of a single agent rather than the method itself. As the performance is similar for the two agents we have trained, we are optimistic that future investigation would be successful.

Another result is presented in figure 4.4. The two plots show how the validation loss evolves over epochs in different training scenarios, with included confidence intervals. Both plots clearly show that the validation loss is lower with our proposed method over essentially all epochs, implicating that the proposed method performs better than the baseline during essentially all stages of the training. This is however potentially subject to bias, as although neither the detector or agent gets to see the validation examples, the environment sends a reward signal based on the validation dataset, as can be seen in algorithm 8. As the detectors using our model also perform better on the test data, this bias, if any, should not be too large. However, if this implicitly creates bias to the validation should be thoroughly investigated, especially if the validation data is used in the training process in some way (for early stopping or similar). This is also the reason to why we have refrained from using the test data in any stage of the agents training.

5.3 Analysis of Learned Policy

This section will cover the results in 4.4 which are results regarding the action pattern of agents. In fig. 4.5, 4.6, 4.7 and 4.8 the action frequencies which also can be interpreted as the empirical action probabilities are visualized. The behaviour of two different agents are shown in the mentioned figures, an agent which is trained on 10.000 images from the Public Dataset and an agent trained on 10.000 images from the In-House Dataset. However, both agents have been evaluated on both the Public and the In-House Dataset. By doing this, it is possible to analyze how a given agent behaves in a familiar domain and an unfamiliar domain [1]. It is also possible to analyze how the agents training dataset impacts the behaviour.

Let's begin with a comparison between the two agents by considering fig. 4.5 and 4.7. The first observation is that both agents have an augmentation that it applies often on both datasets. The most common augmentation applied by the agent trained on the Public Dataset is the Arithmetic augmentation, both for the Public and In-House Dataset. The other agent, trained on the In-House Dataset, applies the Motion Blur augmentation most often, regardless of dataset. For this agent, this pattern also holds for the second and third most common augmentations. This suggests that some augmentations are preferable to others but that this ranking is dataset-specific. This is further supported by studying the frequency of the Gaussian Noise augmentation in fig. 4.5 and 4.7 which is a common action for the agent trained on the Public Dataset (chosen 17% of the time) while its a rare augmentation for the other agent (chosen 2%/8% of the time).

Both of the agents act quite similarly in their familiar and unfamiliar domain. For the agent in fig. 4.5, the four most popular actions (Arithmetic, Box Jitter, Gaussian Noise & Motion Blur) are common but the internal order of them is different. The same observation can be made in fig. 4.7, here the top three most common actions are shared between the dataset-evaluations. Besides the three most common actions, the distribution is not that similar. Again, this suggests that the agent finds a few augmentations which it deems to be best.

Fig. 4.6 and 4.8 shows the same data as 4.5 and 4.7 but the actions have been split into the three severities. Remember, the agent could choose between three severities for every augmentation (except Flip and No Augmentation where it is not really applicable). If the distribution of severities is considered by augmentation, there are some differences in severity-preference between the two agents. Motion Blur was a common action for both of the agents for instance, but in fig. 4.6 it is possible to see that the agent trained

[1] For example, the Public Dataset is the familiar domain for the agent trained on the Public Dataset, while the In-House Dataset is a more unfamiliar domain for this agent.

on the Public Dataset preferred the medium severity while the high setting was selected more often by the agent trained on the In-House Dataset. The relationship between severity-frequencies within an augmentation is quite consistent when the agent augments its familiar and unfamiliar domain.

While the two datasets are from different domains, both of them includes diverse images. In fig. 4.9 the action frequencies are shown when the agent augments images which contains different labels. The plots show the actions of the agent trained on the In-House Dataset. This can be seen as all subplots look quite similar to the plots in fig. 4.7. The three most common actions are generally Motion Blur, Box Jitter and Arithmetic across all subplots in fig. 4.9 but the internal order of these augmentations are however not shared between the labels. For images containing Trucks & License Plates the most common augmentation is Arithmetic while its Motion Blur for the others. For the labels Humans & Faces and Other Vehicles, Box Jitter is more common than Arithmetic. This suggests that the agent actually augments images differently depending on what the image looks like and what is in it. The resulting action distribution is not just an average of what has been good in previous images.

A future experiment that could be conducted is to train a detector with a random augmentation policy but with probabilities according to those seen in fig. 4.5 or 4.7. This experiment would bring more clarity on whether the agent has found a set of action probabilities that works well, regardless of the seen image or if it adjusts the probabilities accordingly. A hypothesis is that a detector trained with random augmentations but modified probabilities (according to fig. 4.5 or 4.7) would perform somewhere in between the detector trained with uniformly random augmentations and the agent augmentations. This is because the agent seems to have found augmentations which it prefers over others (fig. 4.5 & 4.7) while it's possible to see in fig. 4.9 that the agent tends to augment different images slightly different. This property is obviously lost with a random augmentation policy.

A closing remark which applies to every bar plot discussed so far is that a random augmentation strategy would yield a uniform distribution among actions. All distributions shown in the bar plots are very different from a uniform distribution. It is obvious that the agent has learned an augmentation strategy which considers the dataset and images it augments.

Finally, an analysis of how the agent chooses actions within an episode will be conducted. This will be done by studying the graphs in fig. 4.10 which shows the transitions that never occur. To begin with, the agent appears to use more action combinations on the In-House Dataset (which is this agent's familiar domain) compared to when augmenting the Public Dataset. This is because the graph to the right in fig. 4.10 has fewer edges compared to the right graph.

A particularly interesting aspect is that its possible to see that the agent avoids some potentially malicious action combinations. For instance, there are edges between the Rotation and Prison Bars augmentations which is a combination which can ruin an image. If Prison Bars is applied, followed by a rotation, followed by another Prison Bars augmentation, then there is a significant risk that the image is completely black. It is also possible to see that the agent tends to avoid using both Motion Blur and Prison Bars on the same image, this could also be seen as a malicious action combination as those augmentations decreases the image quality, especially with the high severity setting. Gaussian Noise also decreases the image quality and is not used in combination with Prison Bars either.

To end this section, we would like to point out that the results in this section comes from two agents. A more rigorous approach to analyze the learned policies would be to train several agents with the same configuration to determine if the policies are persistent. It could be that an agent trained with the exact same configuration as those analyzed in this section converges to a different local minima. Due to time constraints, this has not been done. When training a new agent, which takes several days, a priority has been to test new ideas. Thus, two agents were rarely trained with the exact same configuration. However, despite slight differences in configuration, the observed patterns are recognized from other trainings. The agent tends to find one or a few favorable augmentations and these have been different between the two datasets, aligning with the results seen in this section.

6 Conclusion

In this thesis we have explored the application of deep reinforcement learning for data augmentation in the context of object detection in images. Guided by a set of research questions, this thesis has aimed to investigate practical and theoretical aspects of this approach. The investigation was motivated by the potential of DRL to optimize data augmentation strategies, a critical factor in the performance of computer vision when data is scarce. This section will answer the research questions in a streamlined fashion, motivated by empirical results and insight gained during the work.

6.1 Does the proposed DRL framework increase performance?

In this thesis, we aimed to assess whether a deep reinforcement learning (DRL) framework could enhance the performance of object detectors through strategic data augmentation. Our investigation, culminating in the analysis of detectors trained with agent-driven and random augmentations, yields promising insights.

Significant Performance Improvement

The statistical analysis, as presented in Table 4.1 and further validated by the paired t-test in Table 4.2, indicates a significant but small improvement in the mean Average Precision (mAP50) for detectors trained with our DRL-based augmentation strategy. Specifically, the results show that in three out of four scenarios, our method outperforms the baseline, demonstrating the potential of DRL in refining data augmentation techniques for object detection.

Cautious Interpretation and Future Directions

While these results are encouraging, we acknowledge the limitations posed by the relatively small sample size. To strengthen the validity and reliability of our findings, further research should focus on:

- Increasing Sample Size: More extensive experimentation will provide a more robust statistical foundation, reducing the likelihood of results being skewed by random variance.

- Training Multiple Agents: Given the inherent randomness in training DRL agents, training and evaluating multiple agents will help ensure that the observed improvements are genuinely attributable to the method, not just the peculiarities of a single agents.

- Mitigating Potential Bias: As our method involves using validation loss to guide the agent's learning, it is crucial to investigate any potential bias towards the validation dataset, as the reward encourages augmentations that enhances learning targeted on the validation dataset. It is thus important for validation dataset and test dataset to have the same domain, i.e. the validation dataset should be representative for the test dataset. Ensuring that the improvement also translates to unseen test data and not only validation data, as observed in our results, is vital for confirming the method's effectiveness.

In summary, the evidence from our research suggests that the proposed DRL framework can indeed enhance the performance of object detectors through strategic data augmentation. This improvement, however, should be further validated with expanded datasets and multiple agent training to solidify these initial promising results. The potential impact of DRL in optimizing data augmentation strategies opens new avenues in object detection and other areas of machine learning where data quality and diversity are critical to model performance.

6.2 What kind of policy is learned by an RL Agent?

The policy learned by the agent uses all augmentations it could choose from. It seems to learn that one or a few augmentations always are good, regardless of the image. We have however seen that the agent learns different action probabilities for images containing different labels, this suggests that the agent also considers the individual image when applying an augmentation rather than learning a distribution that it uses for all images. We have also seen that the agent learns that certain combinations of augmentations are preferable over others, some combinations are never used.

6.3 What does good data augmentation depend on?

One of the hopes for this thesis was to investigate whether there exists something like an optimal augmentation strategy for specific images, if a specific image has a sort of global maxima in how much it can help training the Detector. Our results suggests that a good data augmentation strategy depends on what dataset should be augmented. For our two datasets, belonging to slightly different domains, the resulting agent augmentation strategies are quite different. We also have support for the fact that individual images should be augmented differently, so not only does it matter on a dataset level, but also on image level. A question that haven't been investigated is whether the state of the training affect the augmentation strategy, but more on this in the next section on future work.

6.4 Future Work

In this project we have been able to conclude that a DRL Agent improve the data augmentation strategy such that the performance of an object detection model increases. However, we've only been able to show that it works for relatively small dataset sizes. Our hypothesis is that an object detection model with a given size becomes saturated after being trained on some amount of data. In other words, there comes a point when a model of a given size cannot get any better by increasing number of training samples. A future project could be to develop a framework like the one proposed in this project which works well even when models begin to get saturated.

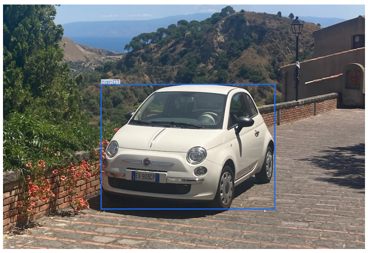
Throughout the course of this project, we've had several ideas that we, unfortunately, haven't yet found the time to implement. For instance, we've assumed in this project that there is an optimal way of augmenting an image. This is probably valid given our current set-up. However, something we haven't considered is that the effect an augmentation has probably varies depending on when during training the augmentation is used. A hypothesis is that the importance of augmentation increases later in the training. The state of the training could be encoded in the state that is presented to the agent. This could be done by modifying the state space and train new agents.

The agent training algorithm is quite slow since one reward function calculation includes backpropagating a training example through the childnet and then evaluating the childnet. If the speed of this training algorithm could be increased, then it would be possible to run more training episodes per time unit which probably would yield a better agent. A future project could be to take inspiration from the GradientNorm reward function (Algorithm 7), but instead of just using the norm of the gradient one could investigate other properties of the gradient. We believe there should be information in the gradient that could be used to determine whether a training example was good or not.

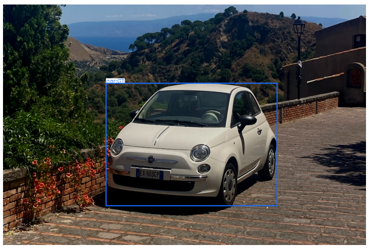
Lastly, the current framework could probably be improved by investigating other deep reinforcement learning algorithms, testing new exciting exploration algorithms and general tuning of hyper parameters.

A

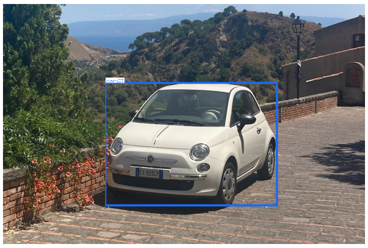
Augmentations



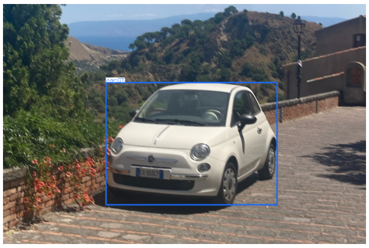
(a) Original Image.



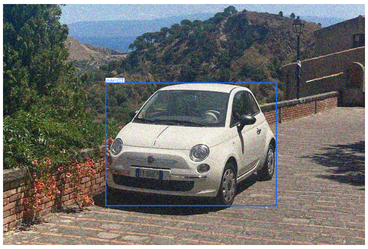
(b) Arithmetic.



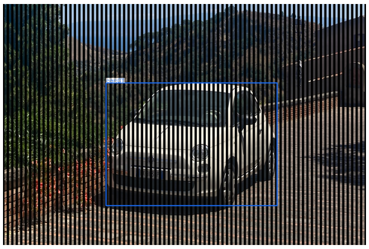
(c) Box Jitter.



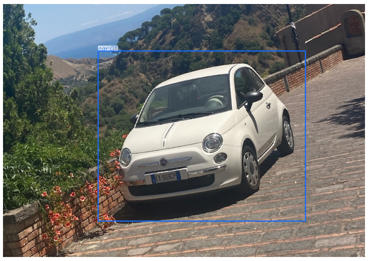
(d) Motion Blur.



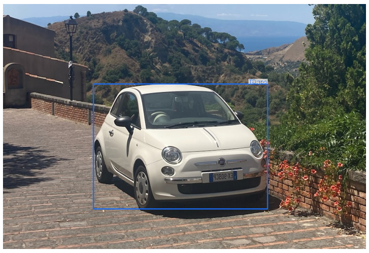
(e) Gaussian Noise.



(f) Prison Bars.



(g) Rotation.



(h) Flip.

Figure A.1 Augmentations which have been used in this project.

B

Training Details

Table B.1 Hyperparameters and settings for agent trainings. These settings are used to produce the results in table 4.1.

Parameter	**Value**	**Note**
Number of Train Episodes	40.000	-
Batch Size	16	Transitions fetched from replay buffer at each step
Max Number of Actions per Episode	3	-
γ	0.99	Discount factor, eq. 2.36
Size of $\mathcal{D}_{reward}$	160	See Algorithm 8
Replay Buffer Capacity	10000	Max Number of transitions stored in replay buffer
N_{Linear}	5	See fig. 3.3, number of FC layers.
N_{Conv}	3	See fig. 3.3, number of Conv. Blocks
Agent Learning Rate	0.0001	–
Childnet Learning Rate	0.001	–
ϵ	0.6	For ϵ-greedy policy, see Algorithm 5
τ	0.6	Target net soft update, see eq. 2.37
C	1000	Target net hard update period, see C in Algorithm 5

Bibliography

Åström, Karl Johan (1965). "Optimal Control of Markov Processes with Incomplete State Information I". eng. Journal of Mathematical Analysis and Applications **10**, 174–205. ISSN: 0022-247X. DOI: {10.1016/0022-247X(65)90154-X}. URL: %7Bhttps://lup.lub.lu.se/search/files/5323668/8867085.pdf%7D.

Barron, A. (1993). "Barron, a.e.: universal approximation bounds for superpositions of a sigmoidal function. ieee trans. on information theory 39, 930-945". *Information Theory, IEEE Transactions on* **39**, pp. 930–945. DOI: 10.1109/18.256500.

Bellemare, M. G., Y. Naddaf, J. Veness, and M. Bowling (2013). "The arcade learning environment: an evaluation platform for general agents". *Journal of Artificial Intelligence Research* **47**, pp. 253–279. ISSN: 1076-9757. DOI: 10.1613/jair.3912. URL: http://dx.doi.org/10.1613/jair.3912.

Bellman, R., R. Bellman, and R. Corporation (1957). *Dynamic Programming*. Rand Corporation research study. Princeton University Press. URL: https://books.google.se/books?id=rZW4ugAACAAJ.

Cho, K., B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014). *Learning phrase representations using rnn encoder-decoder for statistical machine translation*. arXiv: 1406.1078 [cs.CL].

Cubuk, E. D., B. Zoph, D. Mané, V. Vasudevan, and Q. V. Le (2019). "Autoaugment: learning augmentation strategies from data". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 113–123. DOI: 10.1109/CVPR.2019.00020.

Cukierski, W. (2012). *Titanic - machine learning from disaster*. URL: https://kaggle.com/competitions/titanic.

Dong, H., Z. Ding, and S. Zhang, (Eds.) (2020). *Deep Reinforcement Learning: Fundamentals, Research, and Applications*. http://www.deepreinforcementlearningbook.org. Springer Nature.

Guadarrama, S., A. Korattikara, O. Ramirez, P. Castro, E. Holly, S. Fishman, K. Wang, E. Gonina, N. Wu, E. Kokiopoulou, L. Sbaiz, J. Smith, G. Bartók, J. Berent, C. Harris, V. Vanhoucke, and E. Brevdo (2018). *TF-Agents: a library for reinforcement learning in tensorflow*. `https://github.com/tensorflow/agents`. [Online; accessed 25-June-2019]. URL: `https://github.com/tensorflow/agents`.

Hasselt, H. van, A. Guez, and D. Silver (2015). *Deep reinforcement learning with double q-learning*. arXiv: `1509.06461 [cs.LG]`.

He, K., X. Zhang, S. Ren, and J. Sun (2015). *Deep residual learning for image recognition*. arXiv: `1512.03385 [cs.CV]`.

Hochreiter, S. and J. Schmidhuber (1997). "Long short-term memory". *Neural computation* **9**:8, pp. 1735–1780.

Kim, H., J. Kang, W. Park, S. Ko, Y. Cho, D. Yu, Y. Song, and J. Choi (2017). *Convergence analysis of optimization algorithms*. arXiv: `1707.01647 [stat.ML]`.

Kingma, D. P. and J. Ba (2017). *Adam: a method for stochastic optimization*. arXiv: `1412.6980 [cs.LG]`.

Kobayashi, T. and W. E. L. Ilboudo (2021). "T-soft update of target network for deep reinforcement learning". *Neural Networks* **136**, pp. 63–71. ISSN: 0893-6080. DOI: `10.1016/j.neunet.2020.12.023`. URL: `http://dx.doi.org/10.1016/j.neunet.2020.12.023`.

Krizhevsky, A. (2012). "Learning multiple layers of features from tiny images". *University of Toronto*.

Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998). "Gradient-based learning applied to document recognition". *Proceedings of the IEEE* **86**, pp. 2278–2324. DOI: `10.1109/5.726791`.

Lin, T., M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Doll'a r, and C. L. Zitnick (2014). "Microsoft COCO: common objects in context". *CoRR* **abs/1405.0312**. arXiv: `1405.0312`. URL: `http://arxiv.org/abs/1405.0312`.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng (2015). *TensorFlow: large-scale machine learning on heterogeneous systems*. Software available from tensorflow.org. URL: `https://www.tensorflow.org/`.

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis (2015). "Human-level control through deep reinforcement learning". *Nature* **518**:7540, pp. 529–533. ISSN: 00280836. URL: `http://dx.doi.org/10.1038/nature14236`.

Ouyang, L., J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe (2022). *Training language models to follow instructions with human feedback*. arXiv: `2203.02155 [cs.CL]`.

Perez, L. and J. Wang (2017). *The effectiveness of data augmentation in image classification using deep learning*. arXiv: `1712.04621 [cs.CV]`.

Qin, T., Z. Wang, K. He, Y. Shi, Y. Gao, and D. Shen (2020). *Automatic data augmentation via deep reinforcement learning for effective kidney tumor segmentation*. arXiv: `2002.09703 [eess.IV]`.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). "Learning representations by back-propagating errors". *nature* **323**:6088, pp. 533–536.

Russakovsky, O., J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei (2015). "ImageNet Large Scale Visual Recognition Challenge". *International Journal of Computer Vision (IJCV)* **115**:3, pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

Schmidt, R. M. (2019). *Recurrent neural networks (rnns): a gentle introduction and overview*. arXiv: `1912.05911 [cs.LG]`.

Shrivastava, A., A. Gupta, and R. Girshick (2016). *Training region-based object detectors with online hard example mining*. arXiv: `1604.03540 [cs.CV]`.

Sutton, R. S. and A. G. Barto (2018). *Reinforcement Learning: An Introduction*. Second. The MIT Press. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

Tsitsiklis, J. and B. Van Roy (1997). "An analysis of temporal-difference learning with function approximation". *IEEE Transactions on Automatic Control* **42**:5, pp. 674–690. DOI: `10.1109/9.580874`.

Van Hasselt, H. (2010). "Double q-learning." In: pp. 2613–2621.

Watkins, C. (1989). "Learning from delayed rewards".

Yang, S., W. Xiao, M. Zhang, S. Guo, J. Zhao, and F. Shen (2023). *Image data augmentation for deep learning: a survey*. arXiv: `2204.08610 [cs.CV]`.

Zhu, P., X. Li, P. Poupart, and G. Miao (2018). *On improving deep reinforcement learning for pomdps*. arXiv: `1704.07978 [cs.LG]`.

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER'S THESIS *Date of issue* January 2024 *Document Number* TFRT-6225
Author(s) Axel Andersson Nils Hallerfelt	*Supervisor* Robin Göransson, Axis Communications, Sweden Joel Sjöbom, Axis Communications, Sweden Albin Heimerson, Dept. of Automatic Control, Lund University, Sweden Johan Eker, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle

Data Augmentation for Object Detection using Deep Reinforcement Learning

Abstract

Data augmentation is a concept which is used to improve machine learning models for computer vision tasks. It is usually done by firstly, defining a set of functions which transforms images and secondly, applying a random selection of these functions on the images. Since the quality of training data is one of the, if not the most important factor to obtain a good model, this master thesis poses the question whether an intelligent deep reinforcement learning (DRL) agent can select augmentation functions in a better way. More specifically, can the agent select augmentations such that the performance of an object detection model increases? Besides improving the performance of an object detection model, the DRL agent provides insights in what constitutes good data augmentation. The project results in an agent which augments images such that mean average precision (mAP50) increases with 2.3% compared to a baseline detector, trained with random augmentations. This is a promising result that encourages further research on this area. To our knowledge, this is the first time a deep reinforcement learning agent has been used to improve an object detection model via better data augmentation.

Keywords
data augmentation, deep reinforcement learning, machine learning, computer vision, object detection

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 1-97	*Recipient's notes*
Security classification		

http://www.control.lth.se/publications/