Bounding Box Improvement With Reinforcement Learning

by
Andrea Lewis Cleland

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Melanie Mitchell, Chair
Bart Massey
Wu-Chang Feng

Portland State University
2018

Abstract

In this thesis, I explore a reinforcement learning technique for improving bounding box localizations of objects in images. The model takes as input a bounding box already known to overlap an object and aims to improve the fit of the box through a series of transformations that shift the location of the box by translation, or change its size or aspect ratio. Over the course of these actions, the model adapts to new information extracted from the image. This active localization approach contrasts with existing bounding-box regression methods, which extract information from the image only once. I implement, train, and test this reinforcement learning model using data taken from the Portland State Dog-Walking image set [12].

The model balances exploration with exploitation in training using an ϵ-greedy policy. I find that the performance of the model is sensitive to the ϵ-greedy configuration used during training, performing best when the epsilon parameter is set to very low values over the course of training. With $\epsilon = 0.01$, I find the algorithm can improve bounding boxes in about 78% of test cases for the 'dog' object category, and 76% for the 'human' category.

Acknowledgements

I wish to express sincere gratitude to my advisor, Melanie Mitchell, for her guidance, patience, and support, which made this project possible.

I would also like to thank committee members Bart Massey and Wu-chang Feng for the time they took to review my work and offer productive feedback.

Special thanks also to my partner Lizzy and my dad who both have always been a loving bedrock of support.

Contents

List of Figures

List of Tables

Chapter 1

Background

1.1 Introduction

Being able to absorb and interpret visual information to identify and locate objects is an essential tool for survival used both by our species and any other that has eyes. Human beings take visual input to levels far beyond basic survival. We seek out and create objects and scenes of aesthetic beauty, we use visual symbols to represent our language, and we create pictures and visual models to convey our abstract ideas to others. It is therefore unsurprising that figuring out how to teach computers to identify and locate objects in an image would be useful to us. These algorithms are already in use in our social media programs that automatically locate human faces in our photographs, to suggest we "tag our friends." Sometimes these programs recognize your friends' faces and tag them for you. Smart phones are beginning to come with features that allow users to look up information about objects around them by pointing the phone's camera at them. These are only the beginning applications of object localization and classification by computers. Notably these tools are being developed to guide self-driving cars and robots.

However, although the task of locating objects in a scene is easy for humans, it is a non-trivial task for computer programs, which lack the millions of years of evolution that guide our visual cortex. Recent developments of *Convolutional Neural Networks*

(CNNs) have made huge strides in the task of correctly classifying objects. However locating objects is still quite computationally expensive. The most obvious approach is to apply a sliding window that exhaustively searches an image for objects. This is prohibitively slow, so much research in the last few years has focused on ways to economize this search. This is typically done by first generating a list of object proposals based on various probability models of where objects are likely to occur, and then proceeding to refine the search from there.

One commonly used way to refine the search is *bounding box regression.* In bounding box regression, CNN features are extracted from the region of an image given by the object proposal, and are used to statistically predict a more accurate bounding box. This computationally cheap method has been effective in improving bounding boxes in most cases, however there is always interest in investigating alternatives that might prove to work even better.

This thesis investigates one possible alternative that employs reinforcement learning to train an algorithm to generate a better bounding box through a sequence of transformations. In each step, new features are drawn from the image, allowing the algorithm to adapt to new information, as well as to a history of what actions it has done before. This active approach is more computationally expensive than bounding box regression, but may prove to be an effective way to harness contextual information that might be missed by a method that extracts information from the image only once.

Reinforcement learning (RL) refers to a broad range of learning techniques that emulate the way living beings learn by trying actions and learning from successes and failures. In reinforcement learning, an agent is trained to make good decisions in a given environment by receiving rewards for advantageous behavior. Agents observe the state of a given environment, and take actions that transform the environment to a new state. The agent chooses actions according to its policy, $\pi(s)$, which must be learned over the course of training. The chosen action moves the agent to a new state, s'. The action selection process is repeated until a terminal state is reached. The sequence of actions taken by the agent is referred to as an *episode*. In training, when an agent takes an action, it receives a numerical reward, r, often $+1$ or -1. If the reward is positive, the agent updates its policy so that it is more likely to take the same action in similar situations. Through trial and error, the agent learns a policy that aims to maximize the total reward over the course of the episode.

A key concept in reinforcement learning is the trade-off between *exploration* and *exploitation*. An agent exploits what it has learned when it takes actions that have earned it rewards in the past. Exploitation is the ultimate goal of learning, but it can be short sighted if used too much early in training. To learn, an agent must take risks and try new actions. Additionally the agent must learn how one action affects the next. Sometimes one must take a step backward to get around a barrier. A successful reinforcement learning technique needs a way to balance these short-term/long-term trade-offs [15].

One of the simplest and most common methods of balancing exploration and exploitation is an algorithm called *epsilon-greedy* (ϵ-greedy). The parameter $0 < \epsilon < 1$ represents the probability that an agent in an a reinforcement learning system will choose exploration. When choosing exploration, the agent chooses an action to take with uniform random probability. With probability $(1 - \epsilon)$, the agent selects the action it believes to be best based on current information.

The parameter, ϵ, may be held constant over the course of training, or it can vary. One common approach is to start ϵ at a high value close to 1, and then gradually reduce the value over the course of training. This approach is known as *annealing*. The idea behind it is that early in training, the agent has not learned enough to make de-

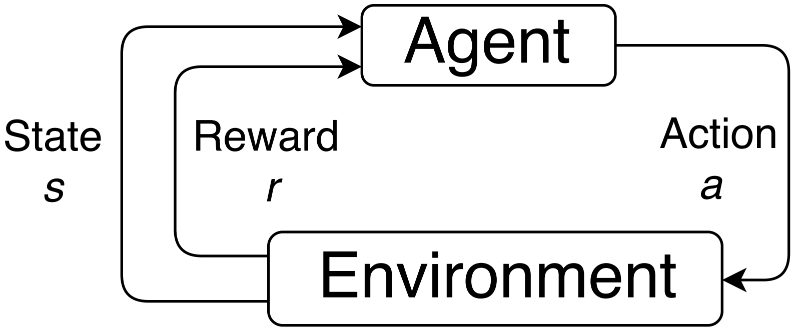


Figure 1.1: A schematic of the flow of information in reinforcement learning system. The agent observes a state given to it by the environment, and chooses an action to take according to its policy. The environment responds by sending the agent a new state, as well as a reward. The agent uses the reward to update its policy and the process begins

cisions, so there is greater advantage to exploration. However, later in training, when the agent has gathered more information, there is greater advantage in exploitation.

The precise ϵ-greedy scheme is up to the researcher to decide. In this thesis, I hypothesize that the ϵ-greedy scheme is an important determinant of the performance of the algorithm. To test this hypothesis, I train the algorithm under seven different ϵ-greedy policies, including six constant-value configurations, and an annealing policy, and compare the performance of each.

1.2 Reinforcement Learning

Reinforcement learning problems are usually imagined the following way. An agent observes a state given by an environment, and chooses an action to take according to a policy the agent is learning. The environment responds by sending the agent a new state and a reward. The agent uses the reward to update its policy and then takes another action. The process repeats until a terminal state is reached, concluding the episode. Figure 1.1 shows a schematic of the flow of information in a reinforcement learning system.

A *Markov Decision Process* (MDP) is a formal mathematical representation of how the agent interacts with the environment to learn its policy. MDPs have the following 5 components:

1. S, the set of states, which includes an initial state s_0 and a terminal state s_T,
2. A, the set of actions,
3. Transition rules that determine the next state s', given a state s and an action a,
4. $r(s, a)$, the function that determines the reward an agent receives when it takes action a in state s,
5. $\gamma \in [0, 1]$, the discount factor, which weights the value of future rewards to against the present reward.

The agent's goal is to learn a policy $\pi(s)$ that maximizes cumulative discounted rewards over the course of the episode:

$$\text{cumulative reward} = \sum_{t=0}^{T} \gamma^t r_t \tag{1.1}$$

where t is the time step indexing the sequence of states from the initial state s_0 to the final state s_T, and r_t refers to the reward the agent receives at time step t. There are two main approaches of reinforcement learning: policy learning and value-function learning. In this thesis, I use a value-function approach known as *Q-Learning.*

1.2.1 Q-Learning

Q-Learning is an RL technique that operates by updating an action-value function, $Q(s, a)$ [17]. It is called an action-value function because it maps state-action pairs to a number, known as the Q-value. The Q-value is a estimate of the *value* of being in a given state. The value is measured as the total cumulative reward the agent accrues as it moves from the current state to the final state in the episode. In other words,

$Q(s, a)$ is an estimate of the sum of discounted rewards accrued by the agent from the current state to the final state of the episode. Once the model's parameters are trained, the agent's action-selection policy is to choose the action with the highest Q-value given the current state.

Suppose an agent is in a state s, and the agent selects action a. The action leads to a new state s', and gives the agent reward r. Q-Learning follows the following update rule:

$$Q(s, a) \leftarrow Q(s, a) + \eta[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \tag{1.2}$$

The learning rate η determines how quickly the Q-value changes to incorporate new information. The term $\max_{a'} Q(s', a')$ is the highest Q-value for the new state s' and is therefore the model's current estimate for the value of being in the state s'. The bracketed portion in (1.2) represents the difference, or error, between the old estimate of the value of taking action a in state s, $Q(s, a)$, and the new (and hopefully better) estimate $r + \gamma \max_{a'} Q(s', a')$.

Why should $r + \gamma \max_{a'} Q(s', a')$ be a better estimate of $Q(s, a)$? Recall that the Q-value estimates the cumulative discounted reward the agent receives from current state to the final state in the episode. Let $Q^*(s_t, a_t)$ refer to the true value of taking action a_t in state s_t. Then

$$Q^*(s_t, a_t) = \sum_{\tau=t}^{T} \gamma^{(\tau-t)} r_\tau \tag{1.3}$$

where τ is used to index time from the current step t to final step T. A relationship between $Q^*(s_t, a_t)$ and $Q^*(s_{t+1}, a_{t+1})$ can be derived as follows.

$$\begin{aligned} Q^*(s_t, a_t) &= \sum_{\tau=t}^{T} \gamma^{(\tau-t)} r_\tau \\ &= r_t + \sum_{\tau=t+1}^{T} \gamma^{(\tau-t)} r_\tau \\ &= r_t + \gamma \sum_{\tau=t+1}^{T} \gamma^{(\tau-(t+1))} r_\tau \\ &= r_t + \gamma Q^*(s_{t+1}, a_{t+1}) \end{aligned} \tag{1.4}$$

It is this recursive relationship between one Q-value and the next that allows the Q-update rule (1.2) to work.

It is useful to visualize the function $Q(s, a)$ as a table with the rows representing possible states of the environment and columns being the actions available for each state. The value located at the s-th row and the a-th column is the Q-value for taking action a in state s. Once the table is "learned", the agent chooses what action to take by looking in the table and choosing the action with the highest Q-value. The table is "learned" by starting with a table with random values, and then in a sequence of trials, the agent tries actions and gets information about whether the action is beneficial via a reward r. The table is then updated according to (1.2). The Q-value tends to increase when r is positive, and decrease when r is negative.

However, as is indicated in (1.2), the update depends not just on the immediate reward r, but also on a "look-ahead" step: $\gamma \max_{a'} Q(s', a')$. After moving to state s', the agent looks in the Q table and finds the maximum Q-value for state s'. This value multiplied by the discount factor and added to the instantaneous reward r represents the current best estimate for $Q(s, a)$. So the update rule (1.2) adjusts the current entry of $Q(s, a)$ by an amount proportional to the difference $r + \gamma max_{a'} Q(s', a') - Q(s, a)$.

Q-Learning has been proven to converge to the optimal policy under fairly minimal conditions [15].

1.2.2 Q-Learning with Perceptrons

A table implementation of $Q(s, a)$, as described above works for environments with small state-action spaces, but for many applications, including this one, there are simply too many possible states to make this work. Instead of learning what action to take in a *specific* state s, it is desirable to learn what action to take in states *similar* to s.

The approach taken in this paper is to approximate $Q(s, a)$ with an ensemble of perceptrons. Perceptrons are artificial neurons that take an input vector $x = (x_1, x_2, ..., x_n)$, and return an *activation* based on a linear application of weights, $w = (w_1, w_2, ..., w_n)$. The most conceptually simple perceptron works as follows. The dot product $w \cdot x = \sum w_i x_i$ is computed, and compared with a *threshold* value. If $w \cdot x > \textit{threshold}$, then the perceptron outputs 1, else it outputs 0. The threshold value must be learned along with the weights. In practice it is often simpler to move the threshold to the left side of the equation and to think of it as another weight to be learned, called the *bias*: $w_0 = \textit{bias} = -\textit{threshold}$.

$$\text{output} = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 & \text{if } w_0 + \sum_{i=1}^{n} w_i x_i \leq 0 \end{cases} \tag{1.5}$$

Equation (1.5) is useful as perceptron activation if the goal is to classify an input vector into a binary category. However the discontinuity sometimes is not desirable because small changes in weights can cause complete reversals in output [15]. A continuous function is also better suited for approximating the Q-function because it

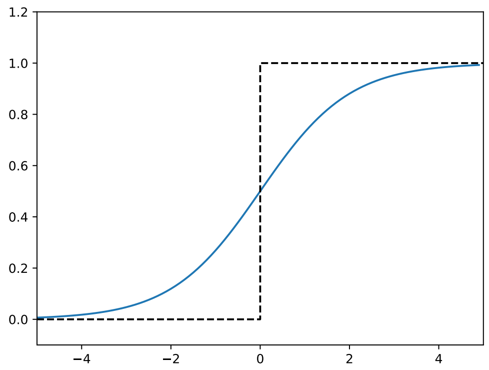


Figure 1.2: Comparison of sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$, with step function (dashed).

outputs a range of values as opposed to just a binary category, allowing for a ranking of the desirability of a number of actions for a given state. This task cannot be accomplished if perceptron output is limited to 1s and 0s.

To turn the discontinuous step function of (1.5), into a smooth continuous function, a common approach is to pass the dot-product output through the sigmoid function: $\sigma(z) = 1/(1 + e^{-z})$. Figure 1.2 shows the difference between these two activation functions. The sigmoid function has the effect of smoothing out the step, while maintaining the same bounds of 0 to 1.

Perceptron weights are updated according to the following rule:

$$w_i \leftarrow w_i + \eta(t - o)x_i \tag{1.6}$$

where t is the target value supplied by the training set, o is the output of the perceptron for input x, and the subscript i denotes which weight is being updated.

The function $Q(s, a)$ can be approximated using perceptrons as follows. Suppose states are represented by a vector of n numerical features: $s = (s_1, ..., s_n)$. Suppose also that there are m possible actions the agent can take in any state. Let the action set be denoted $A = \{a_1, a_2, ..., a_m\}$. A perceptron is created for each action, so that there are m weight vectors, each having length $n + 1$. Thus all the weights can be represented in a $m \times (n + 1)$ matrix W. Q-values for a given state are computed by multiplying the weight matrix by the state vector: Ws^T and applying the sigmoid function to the resulting vector.

$$\begin{pmatrix} w_{10} & w_{11} & w_{12} & \dots & w_{1n} \\ w_{20} & w_{11} & w_{12} & \dots & w_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{m0} & w_{m1} & w_{m2} & \dots & w_{mn} \end{pmatrix} \begin{pmatrix} 1 \\ s_1 \\ \vdots \\ s_n \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ q_m \end{pmatrix} \rightarrow \sigma(\cdot) = \begin{pmatrix} Q(s, a_1) \\ Q(s, a_2) \\ \vdots \\ Q(s, a_m) \end{pmatrix} \tag{1.7}$$

This is essentially how the "table lookup" process works when Q is modeled with perceptrons. To select the optimal action, the agent chooses the maximum value of the resulting vector.

How does learning happen in this system? When the Q-Learning rule (1.2) is compared with the perceptron learning rule (1.6), it can be seen that the target in the perceptron equation is determined by $r + \gamma max_{a'} Q(s', a')$, and that the output of the perceptron corresponds to $Q(s, a)$. The target and the output of the perceptron need to take the same range of values. If the target was simply defined as $r + \gamma max_{a'} Q(s', a')$, its range would depend on the reward scheme and would not necessarily have the same range as $Q(s, a)$, which in the case of a sigmoid activation function, is bounded between 0 and 1. To correct for this, the target needs to be transformed to also take values from 0 to 1. The approach I take is to simply use another sigmoid function: $t = \sigma(r + \gamma max_{a'} Q(s', a'))$. Putting pieces together, I write the update function for my perceptron-Q-function as

$$w_i \leftarrow w_i + \eta(\sigma(r + \gamma max_{a'} Q(s', a')) - Q(s, a))s_i. \tag{1.8}$$

The algorithm for Q-Learning with perceptrons is summarized as follows:

```
for epoch=1 to NumEpochs do
    for each training example do
        Initialize state $s$ ;
        repeat
            Select action a according to epsilon-greedy.;
            Take action a to obtain new state s', and reward r;
            Compute $Q(s', a')$ for all $a'$ in $A$ and find maximum.;
            Update perceptron weights according to (1.8);
            $s \leftarrow s'$ ;
        until end of episode;
    end
end
```

Algorithm 1: Q-Learning with Perceptrons

The ensemble of perceptrons used here to represent the Q-function can be thought of as a 1-layer neural network. This model essentially approximates the Q-function as a linear function of state features. For a given action a_k, the Q-value is the sigmoid of a linear combination of weights and state features:

$$Q(s, a_k) = \sigma(w_0 + \sum w_i s_i) \tag{1.9}$$

It is common to use multilayer neural networks to represent the Q-function, such as in [6]. These deeper networks have the advantage of being able to represent nonlinear relationships between Q-values and state features, but this comes at the cost of having more weights to learn. Larger number of weights require larger training sets and longer training times. I use only a single layer of perceptrons to see if the simplest possible network can obtain productive results.

1.3 Histogram of Oriented Gradients

I represent states in the algorithm as features drawn from the region of the image defined by a bounding box. Due to the recent success and popularity of Convolutional Neural Networks (CNNs), I initially represented states using feature vectors computed using a pre-trained CNN. In practice, however, due to technical difficulties I found this method of extracting to be too slow for use in the algorithm. I needed a fast way of extracting useful information from images, so I turned to Histogram of Oriented Gradients (HOG) [4].

HOG features are obtained by detecting edges in images, computing their slopes and generating histograms of those slopes by region in the image. When Dalal and Triggs developed HOG features in 2005, they found these features were more effective than existing methods for detecting human subjects [4]. My program uses the scikit-image library to compute HOG features [16, 5]. In this section I briefly describe how HOG features are computed and used in my algorithm.

Step 1 - Preprocessing

The part of the image to be analyzed is cropped and resized to desirable dimensions. In my algorithm, given an image and a bounding box, I crop the image by the borders of the bounding box and resize the cropped image to dimensions 128 × 128. I chose these dimensions because they are close to the dimensions used in Dalal and Triggs' original HOG paper, which used 64 × 128 [4]. I also converted the images to grayscale because at the time I implemented my program, the Scikit-Learn HOG library function did not have the option to process color images. However HOG features may be computed in color images, and the most recent version of the library supports this [5]. The remainder of the steps are computed by the library function.

Step 2 - Computing Gradients

Gradients measure the change in intensity I in the image with respect to a small change in x and y. Each gradient is a 2-D vector $(\partial I/\partial x, \partial I/\partial y)$. The length of each gradient vector is the magnitude of the change in intensity. The direction of the vector points in the direction of maximum change. Areas in the image with high contrast have greater gradients and often correspond the edges of objects in images.

Step 3 - Spatial/Orientation Binning

In this step, the image is subdivided into small regions called cells. The size of the cells is a parameter of the function; in this work, I use cells of size 16×16 pixels. Histograms of gradients are computed in each cell in the image. Thus, HOG descriptors are actually arrays of histograms, not a single histogram. Another parameter of the function is to specify how many bins to group the gradients by, and whether to use "signed" gradients $(0 - 360°)$ or unsigned $(0 - 180°)$. Unsigned gradients will group gradient vectors going in opposite directions in the same bin, hence the shorter range of angles. Dalal and Triggs found that unsigned gradients with about 9 bins perform best, so I use these parameters here. The gradient from each pixel in a cell casts a vote in the tally weighted by the magnitude of the gradient.

Step 4 - Block Normalization

Since the histograms are computed with weights based on the magnitude of the gradients, the histograms will be sensitive to overall lighting of the image. Since lighting and contrast within the image varies greatly, it is important to normalize the gradients by locality in the image [11]. There are a number of different possible normalization schemes. In this work I use L2 normalization over blocks of 3×3 cells. Since each cell contains a vector of 9 histogram bins, and each block has 9 cells, each block can be concatenated into an 81-length vector. It is over this vector that the L2 normalization is computed. The blocks are drawn to overlap each other by one row and column of cells. Thus, with my choice of parameters, an image will have height 128 pixels broken into 8 rows of cells, and will have $8 - 3 + 1 = 6$ rows of blocks.

The resulting histogram array, then, with my parameters, will have shape $(6, 6, 3, 3, 9)$: a 6×6 array of blocks, each comprising of a 3×3 square of cells, each containing a 9-bin histogram vector. This array can be compressed into a single vector of length $6 \times 6 \times 3 \times 3 \times 9 = 2916$. This vector is the main component of the state in my Q-Learning system.

Figure 1.3 shows a visualization of the histograms of oriented gradients for an example image provided by Scikit-Learn [5] and for one of the dog images in the data set used in this project. The example from Scikit-Learn in Figure 1.3 shows the potential for HOG features to capture edges in images. Unfortunately for many images in my training set, such as the one pictured in Figure 1.3, edges are not so clear cut and the HOG features are not as impressive to look at. Still, the HOG features for this image do contain useful information on the dog's location in the image.

1.4 Related Work

Q-Learning was developed by Watkins in 1989 [17]. In 1992, Watkins and Dayan published a convergence proof that showed that Q-Learning can achieve the optimal policy for any Markov Decision Process with a finite number of states [18]. In the years since, Q-Learning has been successfully applied to a number of machine learning challenges [8, 1].

The artificial intelligence company Deep Mind has combined Q-Learning with deep neural networks to train a computer to play Atari video games at expert levels [13]. Deep Mind also used reinforcement learning in the development of AlphaGo, the first AI program to defeat an human expert Go player [14].

Reinforcement learning has been applied to train robots to perform a number of tasks [8, 2, 9, 10, 7]. There has been particular interest in using RL techniques to train robots to perform actions based on visual input. For example, Levine et al. develop an "end-to-end" system that links visual input directly to control over the robot's motors. Using this system, Levine et al. train robots to perform everyday tasks such as screwing a lid on a bottle, and placing a coat hanger on a rack [9]. In another work, Zhu et al. use reinforcement learning to train a wheeled robot to navigate a room based on visual input [20].

A number of researchers have applied reinforcement learning to object localization in images [19, 6, 3]. In this section I outline two papers that had particular influence on my work.

This thesis is most closely related to recent work by Caicedo and Lazebnik [6]. These authors use reinforcement learning to perform a top-down search for objects in an image. Their algorithm starts with a box covering a large region of the image and learns a policy to adjust the box over a series of transformations until it (hopefully) identifies an object's location. They use a pre-trained CNN to extract features from the region covered by the bounding box at a given time step. These features, plus an action-history vector define the state used in Caicedo and Lazebnik's reinforcement learning algorithm. States are transformed by actions that shift the box's (x, y) location in the image, or change its size, or aspect ratio. Caicedo and Lazebnik find that this approach works more efficiently than sliding-window approaches because it greatly reduces the number of regions in the image that need to be analyzed before locating an object. The authors find that their algorithm can often locate an object in about 11 to 25 steps.

My algorithm is modeled closely after Caicedo and Lazebnik's approach, but there are a few important differences. First, while Caicedo and Lazebnik's model is designed to locate a number of objects within in an image following a top-down approach, my model aims to only improve the fit of a smaller bounding box already known to overlap the the object. Second, Caicedo and Lazebnik use a Deep Q-Network model to represent their Q-function, which is more complex than the ensemble of perceptrons used here. Caicedo and Lazebnik's Q-learning model follows the approach Mnih et al. [13], which incorporates replay-memory in its update function. Lastly, Caicedo and Lazebnik use CNN features to represent state in their system, while I rely on HOG features, which are more primitive but faster to compute.

Chen et al. apply reinforcement learning to another top-down localization strategy that relies on narrowing the search space using context [3]. The authors liken their approach to playing a game of 20 questions. For example, if the algorithm is querying for cars in an image, it may first look for a road in an image, because cars are usually found on roads. Next the algorithm may look for buildings, because cars are often parked next to buildings. The contextual information of where the road and buildings are in the picture provide a probability map of where to look for cars.

In Chen et al.'s system, there are two types of classes: context classes (sky, road, buildings), and object classes (cars, airplanes, animals). Each have their own pre-trained classifiers. The goal of Chen et al.'s reinforcement learning system is to learn a policy that uses information from context classifers to refine the search space until the search space is narrow enough that an object classifier can thoroughly search the area to discover the objects location in a reduced amount of time.

Chen et al.'s RL system is organized as follows. States are defined as the search area in the image along with observed responses of context classifiers. Actions correspond to a decision to run a particular context classifier, or to determine if an object should be rejected from a search, or to stop gathering information on context and initiate object classifiers. Each time a context classifier is run, the search space is refined to incorporate the new contextual information. Each sequence of context searches is referred to as a search trajectory. A single trajectory may be thought of an episode in Chen et al.'s RL model. In training, rewards are computed as the change in Intersection-Over-Union (IOU) of the search area with the ground truth label as the search space is refined.

The authors approximate their Q-function using a linear function of image and history features, which is parametrically very similar to the ensemble of perceptrons used in this thesis. However Chen et al. apply a very different learning technique to learn the weights. To obtain Q-values for different states and actions, the authors systematically select a number of possible search trajectories. Since the set of possible trajectories is too large to exhaustively try them all, the authors limit trajectories to ones that only have positive rewards. For each trajectory, Chen et al. compute Q-values for each state s and action a by taking a discounted sum of rewards, as in (1.3). Once these (s, a, Q) samples are obtained, Chen et al. use them to compute their linear $Q(s, a)$ model using ridge regression on CNN features taken from the search spaces in the sample.

Chen et al. find that their approach reduces the number of object proposals that need to be scanned to locate an object. This reduction provides a significant speed-up while keeping their model's performance as good as or better than other published methods.

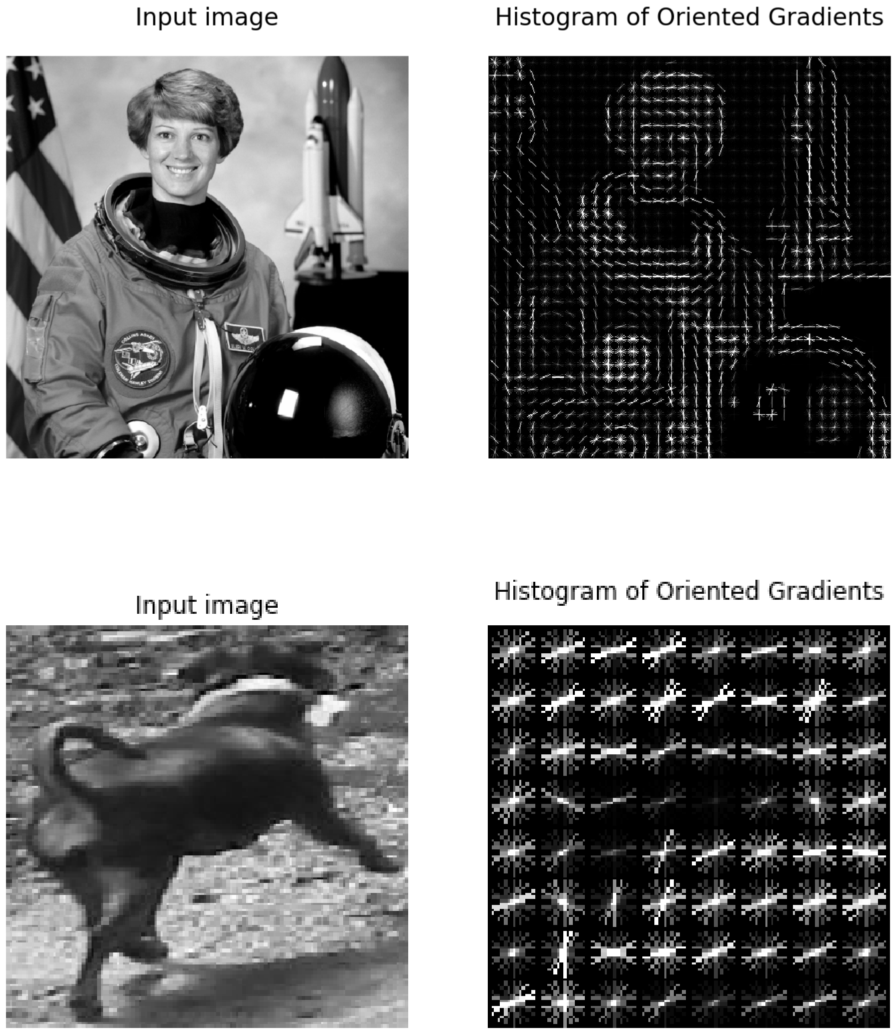

Input image
Histogram of Oriented Gradients
Input image
Histogram of Oriented Gradients

Figure 1.3: (Top) Example from Scikit-Learn showing an input image with a corresponding visualization of the image's HOG features. [5] (Bottom) The same diagram for a dog image used in training, resized to 128 × 128. The HOG visualization shows the slopes of detected edges in each cell.

Chapter 2

Methods

2.1 Dataset Description

The data used in this project are drawn from the Portland State University Dog-Walking Images dataset [12]. The dataset includes 500 images classified as “Prototypical Dog-Walking”, which means each image contains a single dog being walked on a leash by a person that the dataset labels as the “dog-walker.” There is also a set of “Non-Prototypical Dog-Walking” images that vary from this configuration in one or more ways; for example there may be more than one dog, or no leash, or the dog-walker and dog are running, standing, or swimming. To isolate the effect of the Q-Learning algorithm on finding distinct (non-overlapping) objects, I only use the prototypical images in my experiments.

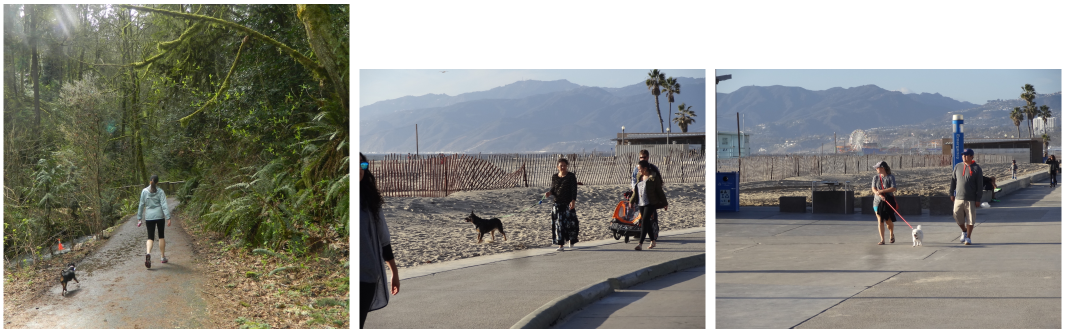


Figure 2.1: Examples of prototypical dog-walking from the Portland State Dog-Walking Images dataset [12]

The dataset includes human-labeled ground truth bounding boxes for each of the three relevant object categories: dog, person, and leash. My focus in this thesis is on the dog and human categories. My algorithm is class-specific, so I train the algorithm separately for each category.

For each object category, I split the data into a training set of 400 images, and a test set of the remaining 100. To save memory during training, I cropped the images so that the ground truth bounding box is centered, and surrounded by a margin equal to the ground truth's width or height (whichever was bigger). For some images, the objects were too close to the edges of the images to support this margin. In these cases, the margin is simply cut short.

2.1.1 Bounding Boxes, IOU, and Skew Generation

I represent bounding boxes as four-vectors: (x, y, w, h), where (x, y) marks the pixel location of the center of the box within the image, and w and h are its width and height in pixels.

In this thesis, I use *Intersection-Over-Union* (IOU) to measure the goodness-of-fit of a bounding box with respect to the ground truth label. As the name suggests, the IOU between two bounding boxes is computed by finding the area of the intersecting region of the two boxes, and dividing by the area of the the union of the two boxes.[1] From here on, when I refer to the *IOU-value* of a bounding box, I mean the bounding box's IOU with respect to the ground truth box.

For each ground truth bounding box $gt = (x, y, w, h)$, I randomly shift each component to generate new bounding boxes called *skews*. I generate ten skews for each ground truth box. These skews are used as the initial boxes to be improved by the

[1] The area of the union is the sum is of the boxes' areas minus the area of the intersecting region. Area is not double counted.

Table 2.1: Initial Bounding Box IOU-Value Statistics. I generate skewed bounding boxes to train and test my algorithm with. This table the shows the range of IOU-values the skews have with respect to their ground truth labels.

		Min	Max	Mean	Std Dev
Humans	Train	0.1004	0.9583	0.4561	0.1674
	Test	0.1004	0.8619	0.4504	0.1602
Dogs	Train	0.1001	0.9624	0.4395	0.1636
	Test	0.1002	0.8809	0.4411	0.1672

Q-Learning algorithm. The skews are generated according to a normal distribution, with the standard deviation equal to 25% width or height of the ground truth box. The skews are also restricted so that they do not exceed the boundaries of the image, and so that they overlap with the ground truth box with minimum IOU = 0.1.

Table 2.1 shows the range of the skews' IOU-values for the training set and test for each object category. The skews' IOU-values range from the cut-off value 0.1 to about 0.9, with a mean of about 0.45.

2.2 Algorithm Description

I implement the Q-Learning system outlined in Section 1.2.2, with states represented as HOG features extracted from the current bounding box combined with an action history vector.

At the beginning of each episode, the agent loads an initial bounding box and ground truth label from the training set. HOG features are computed for the bounding box, and used to initialize the state vector s.

In each step of an episode, the agent receives the current bound box and its correcponding state vector s. The agent selects an action that transforms the box either by translation (left/right, up/down), changing its size (bigger/smaller), or changing its aspect ratio (fatter/taller). The agent selects its action according to an ϵ-greedy policy. With probability $(1 - \epsilon)$, the agent chooses the action corresponding the highest Q-value for state s.

After the action is taken, new HOG features are extracted from the new box, and the action is encoded and added to the history vector. This constitutes the new state, s'. After each action, the new bounding box's IOU-value with respect to ground truth is computed. The agent receives an $r = +1$ reward if the IOU-value increases, an $r = -1$ reward if the IOU-value decreases, and 0 reward if the IOU-value stays the same. The agent then uses s' and r to update its policy according (1.8). After 15 actions, each episode terminates and the algorithm moves on to a new initial bounding box. Training ends after a fixed number of epochs, (200 in my experiments).

2.2.1 Action Definitions

The action set consists of nine possible transformations: $A = \{$*left, right, up, down, bigger, smaller, fatter, taller, stop*$\}$. For a given box $b = (x, y, w, h)$, the transformation of each action is given below.

- *left/right*: $x \leftarrow x \pm \alpha \cdot w$
- *up/down*: $y \leftarrow y \pm \alpha \cdot h$
- *bigger*: $w \leftarrow w(1 + \alpha)$, $h \leftarrow h(1 + \alpha)$
- *smaller*: $w \leftarrow w(1 - \alpha)$, $h \leftarrow h(1 - \alpha)$
- *fatter*: $w \leftarrow w(1 + \alpha)$
- *taller*: $h \leftarrow h(1 + \alpha)$
- *stop*: no change in b

Each transformation depends on the shift factor α, which I set to 0.1. I found that this value was an adequate balance between making changes large enough so that the box makes sufficient progress toward its goal in each step, but small enough that the box doesn't lose track of the target object.

In the future it would be interesting to investigate a stop action that would trigger the end of the episode, and would generate an appropriate reward to the agent for doing so. In this work, however, for simplicity, episodes always terminate after a fixed number of actions have been taken (15 in my experiments). The *stop* action simply leaves the box unchanged, thus giving the agent reward 0 each time it is performed in training. This stop action may seem useless, but it is useful in situations when the box reaches a maximal IOU-value early in the episode, allowing the box the option to stay stationary rather than undergoing transformations that are only likely to reduce the box's IOU-value with ground truth.

2.2.2 State Definition

In each step of an episode, the image is cropped to the edges of the current bounding box and resized to 128×128 pixels. It is then converted to grayscale before its HOG descriptors are computed.

In my HOG specification, I choose 16×16 cells, with normalization blocks sized at 3×3 cells. Unsigned gradients are fitted into 9 bins spaced evenly between 0 and 180°. As derived in the HOG section, these parameters lead to HOG feature vectors of length 2916.

The state s, however, also contains history features that encode which actions have been performed in the last 10 actions. Since there are 9 possible actions, the history vector contains $9 * 10 = 90$ binary features. This history vector specification follows the approach taken by Caicedo and Lazebnik, who found that this history vector was useful in stabilizing search trajectories that would otherwise get stuck in repetitive cycles [6]. The history vector is especially useful in my specification because my *stop* actions otherwise would not change the state at all, and the box might get stuck in one place as soon as the *stop* action is chosen. This would be a good thing if the box was in a position with a high IOU-value with respect to ground truth when the *stop* action is preformed, but as can be seen Figure 2.2, in practice the algorithm often chooses the *stop* action before the box is very well localized.

The HOG descriptors together with the history features combine to make the total state vector length $2916 + 90 = 3006$. This means that together with bias terms, the weight matrix has $(3006 + 1) * (9) = 27,063$ values to be learned.

2.2.3 Algorithm Summary

For each object category (dogs and humans), the training set consists of 400 images, with 10 initial bounding boxes (skews from ground truth) per image. This means that each training set has 4000 training samples, where each sample is an image-box pair $(img, skew)$. To avoid any bias that may result from training from samples in the same order at each epoch, samples are shuffled at the beginning of every epoch. In addition to loading samples in a different order at each epoch, this shuffling means that each episode is likely to begin with a different image than the one used in the previous episode. This is intended to help to prevent the learning process from overfitting on a single image before moving on to the next image. Algorithm 2 describes the training procedure in detail.

```
for epoch=1 to NumEpochs do
    Shuffle the training set;
    for each (img, skew) in training set do
        current_box ← skew;
        Initialize state $s$ ← HOG features from skew, 0 history vector;
        for Action = 1 to ActionsPerEpisode=15 do
            Select action $a$ according to $\epsilon$-greedy.;
            Take action $a$ to obtain new_box;
            Add $a$ to history vector;
            Extract HOG features from new_box and combine with history
              vector to obtain state $s'$;
            Compute change in IOU-value with ground truth to obtain reward $r$;
            Compute $\max_{a'} Q(s', a')$.;
            Update perceptron weights according to Equation (1.8);
            current_box ← new_box;
            $s \leftarrow s'$ ;
        end
    end
end
```

Algorithm 2: Bounding Box Improvement with Q-Learning

2.2.4 Testing

The testing algorithm works similarly to the training algorithm discussed above, except that because there is no learning, there is no need to compute rewards or update weights. Epsilon is set to 0, meaning that actions are chosen solely on basis of highest

Q-value. The same $ActionsPerEpisode = 15$ parameter is used as in training.

The performance of a trained agent is measured by two criteria: the average change in IOU-value, and the *success rate*, which is defined as the fraction of the bounding boxes in the test set that were improved by the algorithm.

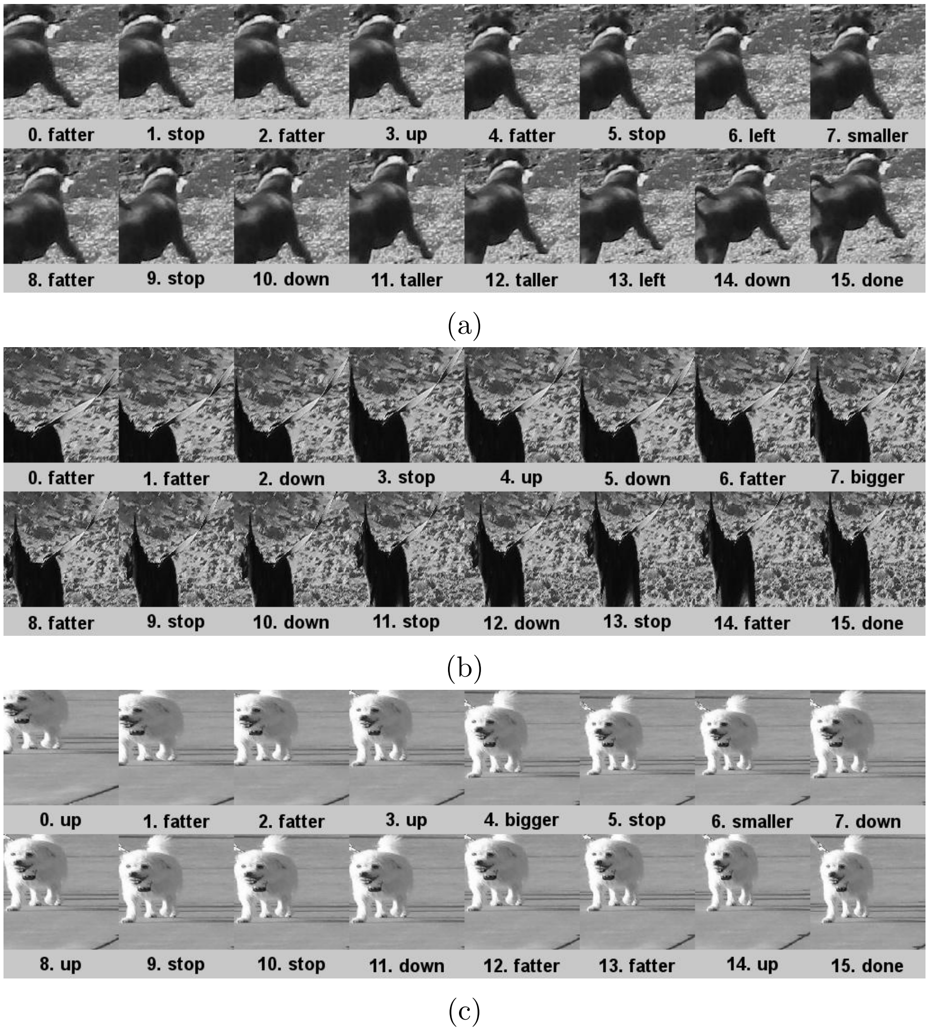
2.2.5 Experiment Design

My hypothesis is that the performance of my Q-Learning model is sensitive to the ϵ-greedy policy used during training. My experiments show the performance of the algorithm under a number of different ϵ-greedy schemes. The algorithm is trained using six different constant values of ϵ: 1.0, 0.75, 0.5, 0.25, 0.01, 0.0, and also an annealing scheme where ϵ is slowly reduced from 0.9 to approximately 0.1 over the course of training. The higher the value of ϵ, the more the agent learns with exploration; the lower the value, the more the agent learns while exploiting what it has already learned.

The values of ϵ chosen in these experiments are designed to show what happens to performance if the learning leans more heavily on exploration, exploitation, or if the two strategies are evenly balanced.

Annealing operates under the idea that early in training, there is greater advantage to exploration, whereas later in training the learning algorithm needs to exploit what it already knows if it is to improve itself. In my experiments, ϵ is reduced linearly using the equation $\epsilon = 0.904 - .004x$, where x is the epoch number (starting with 1). With 200 epochs of training, this means that $\epsilon = 0.9$ in the first epoch, and is 0.104 in the last. In words, this means that the agent goes from taking random actions 90% of the time, down to 50% of the time after 100 epochs of training, and by the time training completes, the agent choosing the action it thinks is best almost 90% of the time.

To account for the fact that there will be variability in performance from run to run, I performed 5 independent runs for each ϵ-policy. In each run, the agent learns using a given ϵ-policy and then the learned Q-matrix is used for action choice on the test set with ϵ set to zero. The average change in IOU with respect to ground truth, and the success rate are recorded.



(a)

(b)

(c)

Figure 2.2: Three examples of episodes. Each subfigure shows a progression from the initial skew (0) to the final bounding box (15) generated by the algorithm. Each frame is labeled with the action chosen to execute next in the episode.

Chapter 3

Results

3.1 Effect of Epsilon-Greedy Policy

Table 3.1 shows the test performance of each trained agent, measured by average change in IOU and success rate, organized by ϵ-greedy policy and object category (Dogs or Humans). There are five runs for each category. These same data are represented in Figure 3.1, which shows the mean and standard deviation of each five-run group.

From visual inspection, a couple of observations can be made. First, it appears that the highest performance occurs when trained at very low values of epsilon. The highest average change in IOU-value and the highest success rate occur at epsilon= 0.01. For the 'dog' category trained at epsilon =0.01, the mean average change in IOU-value is 0.136, with an average success rate of 78.54%. The corresponding mean average IOU-value change for the 'human' category is also 0.136, with an average success rate of 76.26%. Performance is somewhat lower for the epsilon = 0.0 case. For higher values of epsilon, performance declines and become more variable. These trends hold for both average IOU change and success rate in both object categories. The success of low-epsilon runs runs suggest that exploitative actions during training are more advantageous than exploratory random actions, even early in training.

Second, the annealing policy is the next highest performing epsilon category. For dogs, the annealing policy produced a mean average change in IOU-value of 0.121, and mean success rate of 72.9%. For humans, the annealing policy produced mean average change in IOU-value of 0.0745, and mean success rate of 64.9%. In the annealing runs, epsilon never reaches values below 0.1. The high performance of the low-epsilon cases suggests that the annealing policy may be improved if reconfigured to reach lower values of epsilon earlier in training. Unfortunately due to time constraints, I was not able to do this for this thesis.[1]

3.2 Performance by Initial Bounding Box IOU

Intuitively it makes sense that any algorithm will have difficulty improving localization if the initial IOU is already high. It is therefore important to examine how IOU improvements relate to the IOU of the initial bounding boxes. This relationship is shown in Figure 3.2. These plots show final bounding box IOU by initial IOU for each example in the test set. Points above the 45° line indicate bounding boxes that the algorithm improved, while points below the line represent boxes that the algorithm worsened. I show this plot for a selection of individual runs to give an idea of the range of outcomes possible. Among agents trained with annealing, I select the best, median, and worst performing runs for both dogs and humans. Broadly, these plots show a bump in the middle, suggesting the highest performance occurs when initial IOU is about 0.4. After boxes with initial IOU of 0.6 or higher, performance quickly drops off.

To see this pattern more clearly, Figure 3.3 plots average IOU change by initial IOU for the highest performing runs for dogs and humans. This is computed by

[1] The initial epsilon-greedy policies I experimented with were the annealing policy and constant epsilon values 0.25, 0.5, and 0.75. Among these runs, the annealing policy performed best. Analysis of the annealing run's performance suggested that annealing only began to perform better than epsilon=0.25 once epsilon dropped to below 0.25 in the annealing process. This is what sparked interest in performing additional the runs at lower epsilon values 0.0 and 0.01.

Table 3.1: Test Results. Average IOU Change is the mean change in the IOU-value of bounding boxes in the test set with respect to their ground truth labels. Positive values show an average improvement, whereas negative values show an average worsening. The Success Rate is defined is as fraction of bounding boxes in the test set that were improved by the algorithm. A success rate > 0.5 means more boxes are improved than worsened. The table shows both performance measures for 5 independent runs in each ϵ-policy, and object category.

	Dogs		Humans	
Epsilon	Avg IOU Change	Success Rate	Avg IOU Change	Success Rate
Annealed	0.1185	0.706	0.0660	0.609
	0.1272	0.724	0.0771	0.672
	0.1177	0.742	0.1057	0.701
	0.1082	0.686	-0.0172	0.478
	0.1351	0.787	0.1428	0.784
0.0	0.0988	0.792	0.1136	0.759
	0.1034	0.785	0.1154	0.78
	0.1101	0.767	0.1116	0.774
	0.1142	0.758	0.1269	0.811
	0.1257	0.771	0.1237	0.800
0.01	0.1388	0.765	0.1239	0.755
	0.1188	0.792	0.1482	0.771
	0.1626	0.81	0.1318	0.733
	0.1336	0.761	0.1455	0.794
	0.1269	0.799	0.1324	0.76
0.25	0.0277	0.550	-0.0018	0.476
	0.0345	0.593	0.0089	0.488
	0.0210	0.570	0.0480	0.558
	0.0825	0.663	-0.0211	0.387
	0.0377	0.582	0.0070	0.521
0.5	0.0960	0.708	0.0048	0.489
	-0.0623	0.370	0.0506	0.617
	0.0515	0.592	-0.0537	0.385
	0.0195	0.513	0.0448	0.584
	-0.0474	0.405	0.0005	0.489
0.75	-0.0168	0.457	-0.0008	0.493
	0.0215	0.543	-0.0515	0.389
	-0.0396	0.408	0.0665	0.674
	-0.0389	0.409	0.0384	0.588
	-0.0431	0.382	0.0031	0.482
1.0	-0.0511	0.367	-0.255	0.076
	-0.0499	0.379	-0.066	0.346
	-0.0337	0.433	-0.028	0.408
	0.0270	0.555	0.0152	0.526
	-0.0778	0.333	-0.0326	0.409

grouping boxes in the test set by initial IOU in bins of width 0.1. These plots show a common pattern of performance rising in the low-range of initial IOU, reaching a peak in the range of 0.3-0.4, then dropping off steeply. It is not surprising that boxes with already high IOU are not improved by the algorithm, but it is interesting that average improvement is also lower for boxes with low initial IOU. One possible explanation is that that when the box only overlaps with ground truth by a small amount, there is not as much information in the HOG features to guide the algorithm towards improvement. There is a trade-off between this information effect and the effect of there not being enough room for improvement for higher IOU bounding boxes. When the two effects roughly balance, improvement reaches a peak.

3.3 Effect of Number of Epochs Trained

One arbitrary parameter of training is the number epochs to train for. In the experiments discussed in this thesis, this parameter is set at 200 epochs. In earlier experiments, I performed runs with as many as 1000 epochs, but found that performance tapered off after about 200 epochs, which is how that value was chosen. However, those earlier runs were done with constant-epsilon, and only one run was done at a given parameter configuration, so it is worth looking again at how performance varies over the course of training.

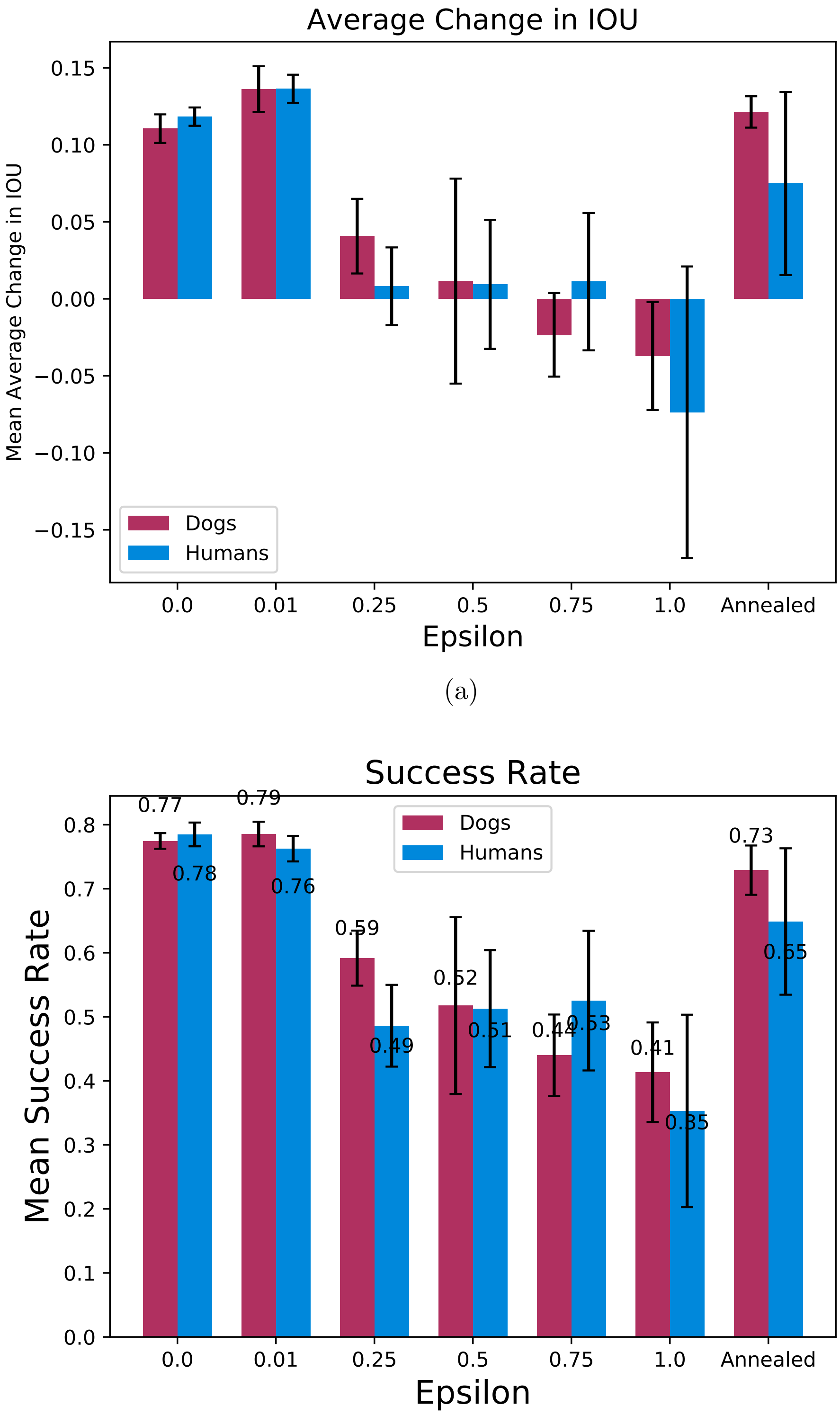
One way of visualizing this is to see how average reward changes over the course of training. Recall that the agent receives a +1 reward if an action increases IOU with ground truth, and -1 if an action reduces IOU. If the average reward is positive during a given epoch, then the agent is taking more actions that improve IOU than worsen it. The average reward per epoch, then, provides a rough measure how the model is performing on the training set. Figure 3.4 shows mean reward for dogs and humans averaged over five runs for each epsilon-greedy configuration. It shows that for constant-epsilon cases 0.25, 0.5, and 0.75, average reward quickly levels off and fails to improve for most epochs of training. It also shows that lower-epsilon runs generate predictably higher rewards. This is expected, as the lower the value of epsilon, the more often the algorithm is using its learned localization policy as opposed to taking actions at random. The annealing policy shows constant improvement over the course of training, which can be more or less completely attributed to its declining epsilon value. The annealing runs begin at 0.9 and decline by .004 each epoch. This implies that ϵ reaches 0.75, 0.5, and 0.25 at epochs 38.5, 101, and 163.5 respectively, which almost exactly predicts where the annealing reward plots cross with their constant-epsilon counterparts. There is a slight arc to the annealing reward plots, suggesting that they may level off somewhere if they were trained for more epochs.

The reward plots for epsilon = 0.0 and 0.01 appear to follow a different trend than the rest of the constant-epsilon cases. Both of these plots show consistent improvement in average reward throughout the course of training. Surprisingly, the epsilon = 0.01 case shows higher average rewards than the epsilon = 0.0. This contrasts strongly with the pattern observed in the 0.25, 0.5, 0.75, and annealing runs. It also worth noting that average reward for the epsilon = 0.0 case is actually lower than that of the epsilon = 0.25 case for much of training. What is causing this pattern is worthy of further study.

Performance by epoch can be more directly observed by testing the weights at various points of training against the test set. Plotting performance of these weights by number of epochs trained provide a visualization of the agent's "learning curve." Figure 3.5 shows success rate for dogs and humans averaged over the five runs for each epsilon-greedy policy. It shows that although there are ups and downs, constant-epsilon runs level off in performance relatively early on in training (after about 50 epochs). This leveling off occurs for both the high-epsilon runs of 0.25 and greater as well as for the low-epsilon cases of 0.0 and 0.01. However, the level of performance is much higher for the low-epsilon runs.

In contrast, no leveling off is observed for the annealing runs. For dogs, the annealing performance appears to be closing the gap with the low-epsilon runs in the last few epochs of training. For the 'human' annealing runs, the gap with low-epsilon runs is greater, but the trend of annealing performance is nonetheless on the rise in the last epochs of training. This suggests that for annealing runs, performance may increase if trained for more than 200 epochs, or if the annealing function is reconfigured to train at lower epsilon values. A straightforward extension of my experiments would be to simply train these agents for more epochs, continuing the same annealing policy, which would reach $\epsilon = 0$ at epoch 226, and see what happens to their performance. Due to time constraints, I was not able to carry out this extension.

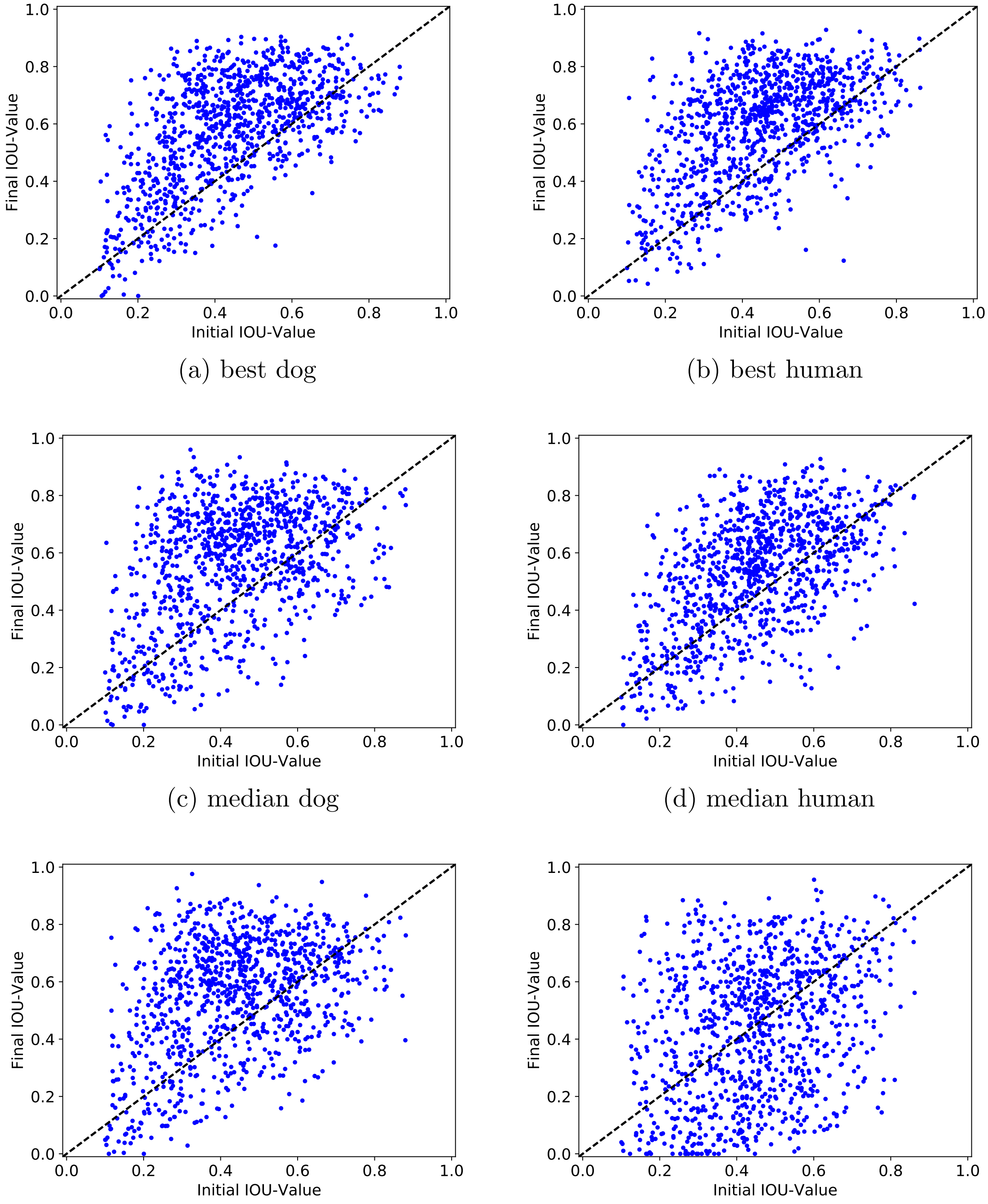
One mystery that emerges from comparing the learning curve plots with the average reward plots is that for the low-epsilon runs of 0.0 and 0.01, performance on the test set levels off between 25 and 50 epochs of training, but the corresponding average reward by epoch of training shows constant improvement throughout training. Determining the cause of this warrants further investigation. It is possible that the improvement in average reward may be due to the weights overfitting to the training set, but this would require further testing to verify.



(a)

(b)

Figure 3.1: Average Change in IOU, and Success Rate averaged over 5 independent runs for each ϵ-policy. Error bars indicate standard deviation.



(a) best dog

(b) best human

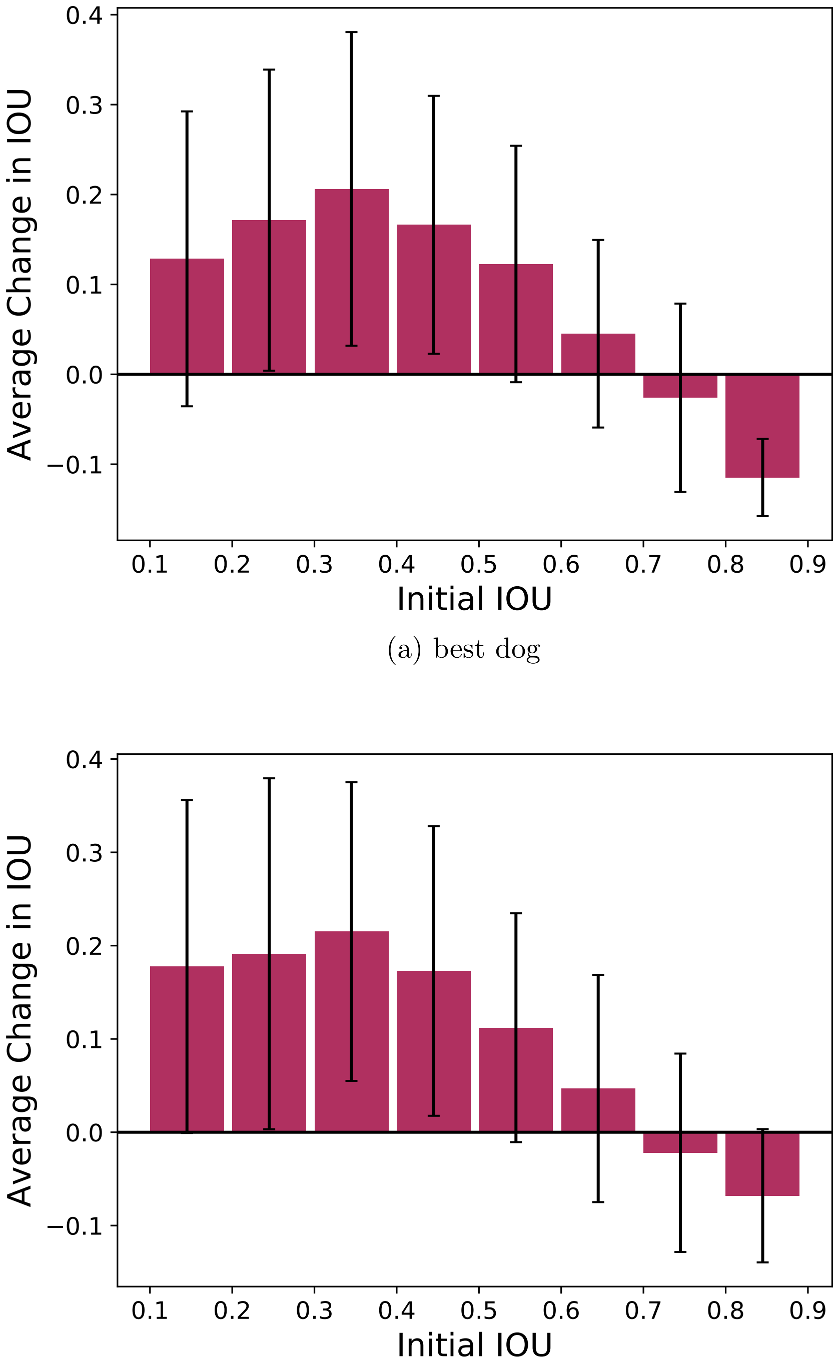
(c) median dog

(d) median human

(e) worst dog

(f) worst human

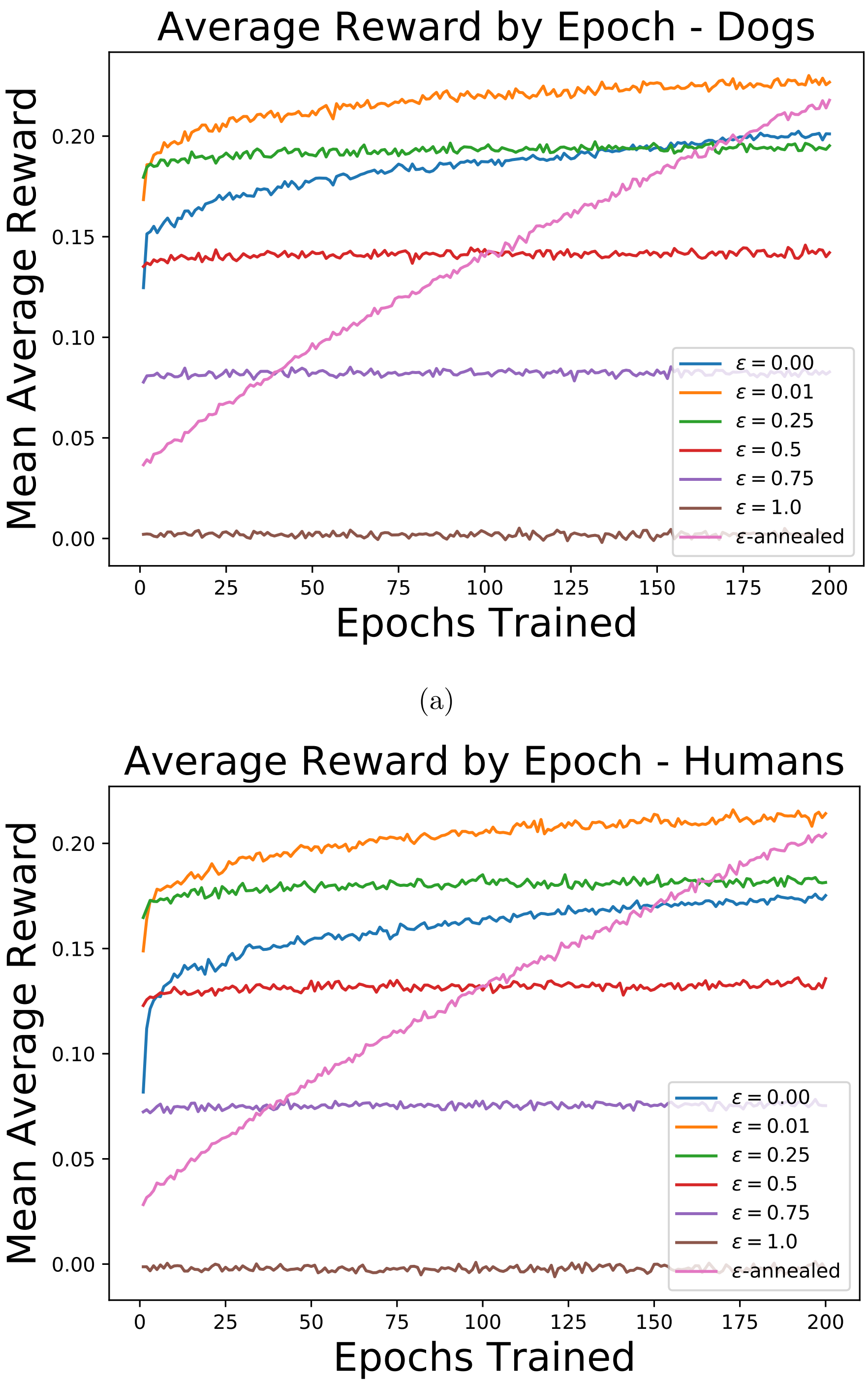
Figure 3.2: Final IOUs plotted against initial IOUs on the test set for a selection of trained agents. Points above the 45° line indicate an improved bounding box. Shown are the best, median, and worst results among agents trained with annealing.



(a) best dog

(b) best human

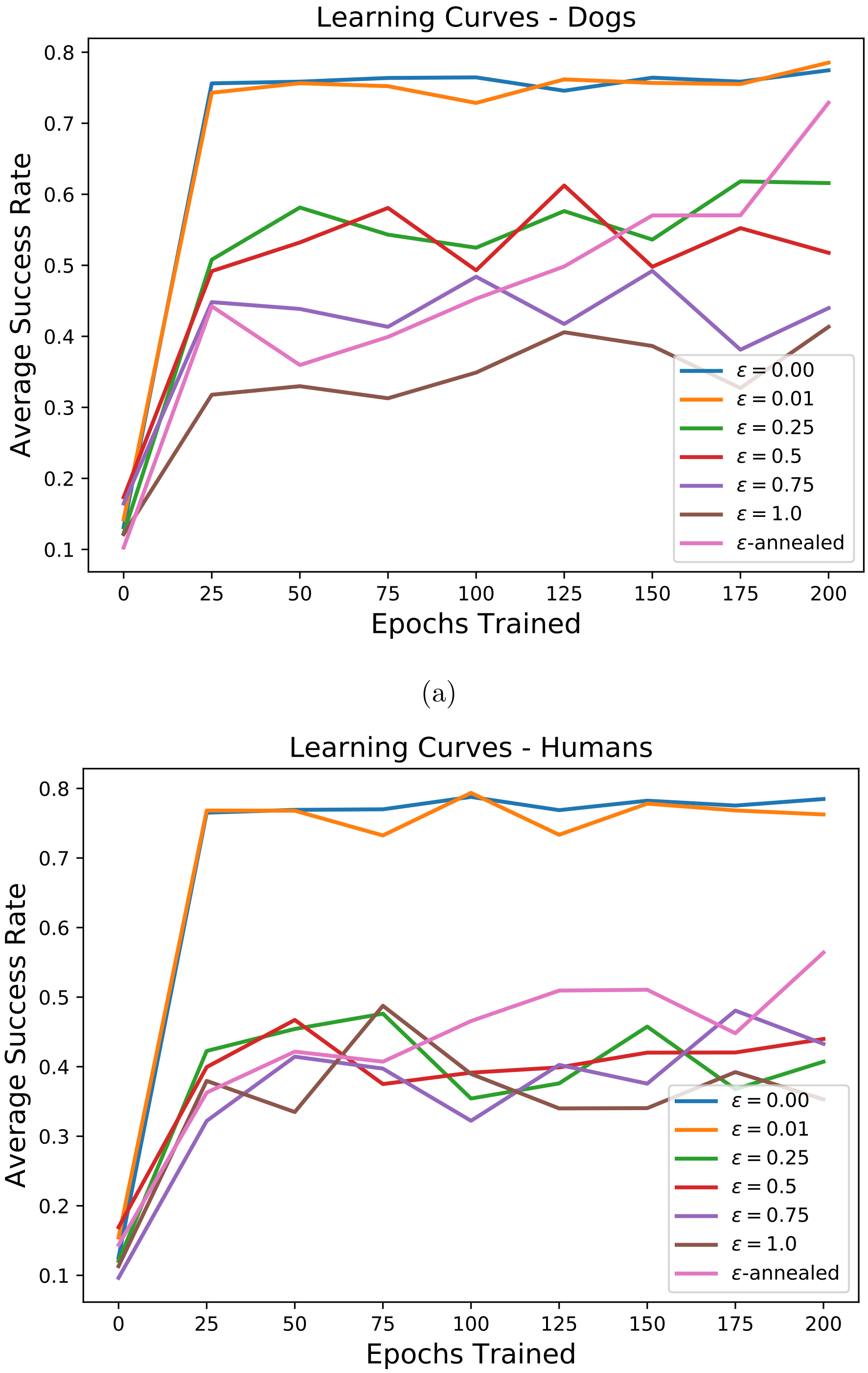
Figure 3.3: Average IOU Change by Initial IOU-value for the highest performing runs with annealing policy: (a) dogs and (b) humans. Subplots (a) and (b) correspond with Figure 3.2(a), and 3.2(b) respectively



(a)

(b)

Figure 3.4: Average reward by epoch, averaged over 5 independent runs (a) dogs, and (b) humans. Best viewed in color.



(a)

(b)

Figure 3.5: Success Rate, defined by the fraction of bounding boxes in the test set improved by the algorithm, averaged over 5 independent runs for each ϵ-policy, plotted by number of epochs trained. Shown for (a) dogs, and (b) humans. Best viewed in color.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

This thesis implements and analyzes one way reinforcement learning can be applied to the task of object localization. I proposed that the performance of this algorithm is dependent on the ϵ-greedy configuration used during training, which determines how often the agent chooses an action in an exploratory manner, as opposed to exploiting what it has already learned. Given a bounding box proposal that overlaps an object, the algorithm seeks to improve the bounding box through a series of actions that transform the box, adaptively reacting to new information as it does so. This is in contrast to the commonly used method of bounding box regression, which adjusts the box based on a static analysis of features drawn from the initial bounding box. Q-learning is applied, using an ensemble of perceptrons to approximate the $Q(s, a)$ function. States are represented primarily by Histogram of Oriented Gradients (HOG) descriptors drawn from the current bounding box. An ϵ-greedy algorithm is used to balance exploration and exploitation during training.

I implemented, trained, and tested this model for six different constant values of ϵ, and under an annealing policy whereby ϵ is reduced gradually over the course of training. I find that the highest performing runs occur for very low values of ϵ: 0.01 and 0.0. With $\epsilon = 0.01$, average IOU-value increase is 0.136 for both 'dog' and ‘human’ categories. The corresponding average success rates are 78.5% for dogs and 76.3% for humans. For $\epsilon = 0.25$ and higher, performance is much lower and more variable.

I find that annealing performs substantially higher than cases where $\epsilon > 0.25$, but significantly less than the $\epsilon = 0.01$ case. For images of dogs, the algorithm trained with annealing improved bounding boxes roughly 70% of the time, with an average IOU improvement of about 0.12. For humans, the algorithm with annealing improved bounding boxes about 65% of the time, with an average improvement in IOU of 0.075. Under the annealing policy in this experiment, epsilon never reaches values less that 0.1. The success of the low-epsilon runs suggests that the annealing policy may be improved if the annealing function were modified to accommodate lower epsilon values during training. However, this requires further testing to verify.

Are the improvements provided by this algorithm good enough for use cases? The answer depends on the nature of what the application wants to accomplish with visual object localization. If the application is a life-or-death situation, as may be the case with self-driving cars, a more accurate method is probably needed. However, there may be many lower-staked applications for which this method would be a significant help. As mentioned in the introduction, cell phones now feature applications that allow the user to to perform visual information queries by pointing their phone at objects. These programs perform localization on any number of relevant objects in the phone’s camera’s range of vision. For this application, a method such as the active-localization Q-Learning model developed here may be quite helpful. This is especially true if the program can identify situations where active-localization may have an edge over bounding box regression.

4.2 Future Work

This work could be extended along a number of different avenues. Analysis of the algorithm's learning curve suggests that simply increasing the number of epochs trained under the same annealing method may improve performance. One can also explore how different configurations of annealing compare. This thesis implemented a linear annealing policy, but there are a myriad of different choices that could be explored. Since the algorithm appears to perform better when trained at lower values of epsilon, perhaps having an annealing function with a faster drop-off would be advantageous. This could be accomplished with an exponential function or by using a steep linear drop off, followed by many epochs trained with low epsilon values. It would be interesting to compare how an annealing policy with this steep drop off would compare with the low-epsilon cases explored in this thesis.

One puzzle that comes from my results is the fact that the performance of the low-epsilon cases levels off after just 25-50 epochs of training, while the average reward for those same runs rises throughout the 200 epochs of training. One possible explanation is that after about 50 epochs, the weights begin to overfit to the training data. This requires further investigation.

Other extensions to this work could include further investigation of the parameter-space. The learning rate η and the discount rate γ could be varied, as well as the number of actions per episode.

For the purposes of computational speed, this thesis uses HOG features, but current state-of-the-art object classification and localization techniques all use Convolutional Neural Networks (CNN). My project initially used CNN features to represent states, but unfortunately I was not able develop a system that could extract these features fast enough to train my algorithm in a reasonable amount of time. However in principle, this should be possible, and it would likely improve performance. Using CNN features would also make the algorithm more comparable to existing bounding-box regression methods which also use these features.

My algorithm currently uses a highly simplistic stop mechanism, where each episode consists of exactly 15 actions. It is probably more desirable to implement a stop action that triggers the end of the episode. This would allow the number of actions per episode to adapt as needed to a given situation. Caceido and Lazebnick implement a version of this trigger in their paper [6]. It would be interesting to adapt their trigger to my model and see how it compares to the simplistic policy of constant actions per episode implemented here.

Perhaps the most important future work would involve a systematic comparison of performance of an adaptive localization method trained with reinforcement learning such as this one, with bounding box regression. The goal would be to understand in which kind of situation each method most excels.

Bibliography

[1] K. Arulkumaran et al. "A Brief Survey of Deep Reinforcement Learning". In: *IEE Signal Processing Magazine, Special Issue on Deep Learning for Image Understanding* (2017). URL: https://arxiv.org/pdf/1708.05866.pdf.

[2] M. Asada et al. "Purposive Behavior Acquisition for a Real Robot by Vision-Based Reinforcement Learning". In: *Machine Learning* 23 (1996), pp. 279–303.

[3] X. S. Chen, H. He, and L. S. Davis. "Object detection in 20 questions". In: *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*. Mar. 2016, pp. 1–9.

[4] N. Dalal and B. Triggs. "Histograms of Oriented Gradients for Human Detection". In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2005).

[5] *Histogram of Oriented Gradients — skimage v0.13x docs.* URL: http://scikit-image.org/docs/0.13.x/auto_examples/features_detection/plot_hog.html.

[6] S. Lazebnik J. C. Caicedo. "Active Object Localization with Deep Reinforcement Learning". In: *International Conference on Computer Vision (ICCV)* (2015), pp. 2488–2496. URL: http://slazebni.cs.illinois.edu/publications/iccv15_active.pdf.

[7] F. Kirchner. “Q-learning of complex behaviours on a six-legged walking machine”. In: *Proceedings Second EUROMICRO Workshop on Advanced Mobile Robots*. 1997, pp. 51–58.

[8] J. Kober, J.A. Bagnell, and J. Peters. “Reinforcement learning in robotics: A survey”. In: *The International Journal of Robotics Research* 32 (11 2013), pp. 1238–1274.

[9] S. Levine et al. “End-to-End Training of Deep Visuomotor Policies”. In: *JMLR* 17 (39 2016), pp. 1–40.

[10] S. Levine et al. “Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection”. In: *ISER* (2016).

[11] S. Mallick. *Histogram of Oriented Gradients*. 2016. URL: https://www.learnopencv.com/histogram-of-oriented-gradients/.

[12] M. Mitchell. *Portland State Dog-Walking Images*. URL: http://web.cecs.pdx.edu/~mm/PortlandStateDogWalkingImages.html.

[13] V. Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518 (2015), pp. 529–533.

[14] D. Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529 (2016), p. 484.

[15] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction, 2nd Edition (Draft)*, p. 142. URL: http://incompleteideas.net/sutton/book/bookdraft2017june19.pdf.

[16] S. van der Walt, J. Schönberger, et al. “scikit-image: image processing in Python”. In: *PeerJ* 2 (June 2014), e453. URL: http://dx.doi.org/10.7717/peerj.453.

[17] C.J.C.H. Watkins. “Learning from delayed rewards”. PhD thesis. University of Cambridge, 1989.

[18] C.J.C.H. Watkins and P. Dayan. “Q-learning”. In: *Machine Learning* 8 (1992), p. 279.

[19] K. Xu et al. “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention”. In: *ICML* 14 (2015).

[20] Y. Zhu et al. “Target-Driven Visual Navigation in Indoor Scenes using Deep Reinforcement Learning”. In: *ICRA* (2017).