



CS 237B: Principles of Robot Autonomy II

Problem Set 1: Learning-based Perception and Control

Due Feb 2nd, 2024 11:59PM

Submission Instructions

You will submit your homework to Gradescope. Your submission will consist of:

1. a single pdf with your written answers for report questions (denoted by the  symbol). This must be typeset (e.g., L^AT_EX or Word).
2. a .zip folder containing your code for the programming questions (denoted by the  symbol).

Honor Code

- **Collaboration:** You may collaborate with other students on the homework. However, each student should independently write up their own solutions and **clearly list the names of their collaborators** in their write-up.
- **Committing code to GitHub:** Please do not push your homework code to public GitHub repositories (for example, a fork of our repository). If you wish to commit your solutions to GitHub, please create a **private repository** by making a new copy of our repository, rather than forking it.

Introduction

For this homework, you will explore different elements of reinforcement learning and machine learning. In particular you will investigate the following:

1. Model Based and Model Free Reinforcement Learning
2. Classification and sliding window detection

Further, in terms of software development, you will

- Use TensorFlow
- Learn about retraining a pretrained model for image recognition
- Use Tensorboard to visualize a neural network model and observe the training process

Starter code for this problem set has been made available online through github; to get started download the code by running `git clone https://github.com/PrinciplesofRobotAutonomy/CS237B_HW1.git` in a terminal window. We strongly encourage you to take a look at all the code, even sections that you will not be directly working on.

Install Additional Software Dependencies

Make sure you have Python 3.9 installed. We highly recommend using a virtual environment from [Miniconda](#).

To set up a virtual environment with Miniconda the steps are the following:

- download and install Miniconda from <https://docs.conda.io/en/latest/miniconda.html>
- open a terminal and run the following command

```
$ conda create -n cs237b python=3.9 -y
```

- activate the virtual environment

```
$ conda activate cs237b # activates the python environment
```

- install dependencies for this class

```
$ pip install -r requirements.txt
```

Important: Every time you re-open the terminal window you need to activate the environment with `conda activate cs237b`.

However, feel free to use any method to obtain a working version of Python with requirements from `requirements.txt`.

One good, GPU-enabled *alternative* is to use [Google Colab](#).

A Note on TensorFlow

This homework is written in eager TensorFlow allowed by TensorFlow ver. 2. TensorFlow is still heavily used in industry and the eager implementation is similar to other popular machine learning frameworks, so if you decided to use another one later in your career, the transition should be seamless.

Problem 1: Markovian Drone

In this problem, we will apply techniques for solving a Markov Decision Process (MDP) to guide a flying drone to its destination through a storm. The world is represented as an $n \times n$ grid, i.e., the state space is

$$\mathcal{X} := \{(x_1, x_2) \in \mathbb{R}^2 \mid x_1, x_2 \in \{0, 1, \dots, n-1\}\}.$$

In these coordinates, $(0, 0)$ represents the bottom left corner of the map and $(n-1, n-1)$ represents the top right corner of the map. From any location $x = (x_1, x_2) \in \mathcal{X}$, the drone has four possible directions it can move in, i.e.,

$$\mathcal{U} := \{\text{right}, \text{up}, \text{left}, \text{down}\}.$$


The corresponding state changes for each action are:

- **right:** $(x_1, x_2) \mapsto (x_1 + 1, x_2)$
- **up:** $(x_1, x_2) \mapsto (x_1, x_2 + 1)$
- **left:** $(x_1, x_2) \mapsto (x_1 - 1, x_2)$
- **down:** $(x_1, x_2) \mapsto (x_1, x_2 - 1)$

Additionally, there is a storm centered at $x_{\text{eye}} \in \mathcal{X}$. The storm's influence is strongest at its center and decays farther from the center according to the equation $\omega(x) = \exp\left(-\frac{\|x - x_{\text{eye}}\|_2}{2\sigma^2}\right)$. Given its current state x and action u , the drone's next state is determined as follows:


- With probability $\omega(x)$, the storm will cause the drone to move in a uniformly random direction.
- With probability $1 - \omega(x)$, the drone will move in the direction specified by the action.
- If the resulting movement would cause the drone to leave \mathcal{X} , then it will not move at all. For example, if the drone is on the right boundary of the map, then moving right will do nothing.

The quadrotor's objective is to reach $x_{\text{goal}} \in \mathcal{X}$, so the reward function is the indicator function $R(x) = I_{x_{\text{goal}}}(x)$. In other words, the drone will receive a reward of 1 if it reaches the $x_{\text{goal}} \in \mathcal{X}$, and a reward of 0 otherwise. The reward of a trajectory in this infinite horizon problem is a discounted sum of the rewards earned in each timestep, with discount factor $\gamma \in (0, 1)$. Take a look at `Problem_1/value_iteration.py`.


- (i)  Given $n = 20$, $\sigma = 1$, $\gamma = 0.95$, $x_{\text{eye}} = (15, 7)$, and $x_{\text{goal}} = (19, 9)$, write code that uses value iteration to find the optimal value function for the drone to navigate the storm. Recall that value iteration repeats the Bellman update

$$V(x) \leftarrow \max_{u \in \mathcal{U}} \begin{cases} R(x, u) + \gamma \sum_{x' \in \mathcal{X}} p(x'; x, u) V(x') & \text{if } x \text{ is not a terminal state} \\ R(x, u) & \text{otherwise} \end{cases}$$

until convergence, where $p(x'; x, u)$ is the probability distribution of the next state being x' after taking action u in state x , and R is the reward function.

- (ii)  Plot a heatmap of the optimal value function obtained by value iteration over the grid \mathcal{X} , with $x = (0, 0)$ in the bottom left corner, $x = (n-1, n-1)$ in the top right corner, the x_1 -axis along the bottom edge, and the x_2 -axis along the left edge.


Hint: We provide a function that plots the heatmap: `visualize_value_function()`.

- (iii)  Recall that a policy π is a mapping $\pi : \mathcal{X} \rightarrow \mathcal{U}$ where $\pi(x)$ specifies the action to be taken should the drone find itself in state x . An optimal value function V^* induces an optimal policy π^* such that

$$\pi^*(x) \in \operatorname{argmax}_{u \in \mathcal{U}} \left\{ R(x, u) + \gamma \sum_{x' \in \mathcal{X}} p(x'; x, u) V^*(x') \right\}$$

Note that the optimal policy is only defined for non-terminal states.

Use the value function you computed in part (a) to compute an optimal policy. Then, use this policy to simulate the MDP starting from $x = (0, 0)$ over $N = 100$ time steps.

- (iv)  Plot the policy as a heatmap. Plot the simulated drone trajectory overlaid on the policy heatmap, and briefly describe in words what the policy is doing.


Hint: Feel free to modify `visualize_value_function()` or write it from scratch yourself.

Another popular approach to reinforcement learning is to use a Q-network which encodes expected future discounted reward for both a given state and action pair. The optimal policy for a value function is given by:

$$\pi^*(x) \in \operatorname{argmax}_{u \in \mathcal{U}} Q(x, u)$$

A very popular approach is to approximate the Q-function as a feed-forward neural network. We will now prepare training data, train the Q-function approximation (or the Q-network now) and compare the approximately optimal policy to the optimal policy found by value iteration.


Take a look at `Problem_1/q_learning.py`.

- (v)  Given the same environment, sample 10^5 state transition triples

$$(x_i, u_i, x'_i) \quad \forall i \in [1, 10^5]$$

Make sure to generate transition samples with appropriate probability of getting blown off course by the storm.



Important: We are representing the state as a two dimensional vector with the two grid coordinates and the action as a single element vector. You are free to use your own state and action representation, but another choice might require a significant amount of tuning of the resulting Q-network.

- (vi)  In order to develop the Q-network loss function, we will start by writing down the Bellman equation of the optimal Q-function—our Q-network should aim to approximate that. Write down the expectation form of the optimal Q-function in terms of an equality

$$Q^*(x, u) = \dots \tag{1}$$

Hint: Your form should not contain the transition probabilities, since we do not know those.


Hint: Remember to account for the reward value and whether the state is terminal.

- (vii)  Create a feed forward neural network for representing the Q-network. Use 3 dense layers with a width of 64 (two hidden 64 neuron embeddings).
- (viii)  Train the neural network filling in the provided `Q_learning` Python function. Use the Adam optimizer. Experiment with the following step sizes $\{10^{-3}, 10^{-2}, 10^{-1}\}$ and pick the one that works best.


We will now develop the Q-network loss as an L2 penalized equality (1) residual.

$$\ell = \frac{1}{n} \sum_{i=1}^n \|\text{LHS}_i - \text{RHS}_i\|_2^2 \quad \forall i \in S_{\text{examples}}$$

Hint: The expectation operator can be silently dropped since we are (1) using a quadratic residual penalty and (2) summing over all of the samples—which is equivalent to taking empirical expectation over data.

- (ix)  Describe a dynamical system or a dataset situation in which using Q-learning could be easier than using value iteration.

Hint: You don't need to limit the example to the field of robotics.

- (x)  Include a binary heatmap plot that shows, for every state, if the approximate Q-network policy agrees or disagrees with the value iteration optimal policy.

Hint: Feel free to modify `visualize_value_function()` or write it from scratch yourself.

Problem 2: Classification and Sliding Window Detection

Even with the vast reduction in model parameters achieved by convolutional neural networks (CNNs), compared to fully connected neural networks, training modern visual recognition models from scratch can still take days on immensely powerful computing hardware. But by leveraging the feature-extraction prowess of a pre-trained image classification CNN, in this case Google’s Inception-v3 [1], even those of us without a supercomputer (and with a homework deadline!) can train a high quality image classifier on our own custom image data.¹ **Problem Setup:** We will be using TensorFlow to perform the numerical computations involved in training and evaluating neural networks in this problem.

Take a look at the directory Problem_2. The files for this problem should be organized as:



- datasets/ → labeled images from the PASCAL Visual Object Classes Challenge 2007 [2]
 - datasets/train → training images with labels for supervised classification learning
 - * datasets/train/cat → pictures of cats!
 - * datasets/train/dog → pictures of dogs!
 - * datasets/train/neg → pictures of neither (mostly planes, trains, and automobiles)
 - datasets/test → test images with labels to evaluate the performance of our model
 - * datasets/test/cat → pictures of cats!
 - * datasets/test/dog → pictures of dogs!
 - * datasets/test/neg → pictures of neither (mostly planes, trains, and automobiles)
 - datasets/catswithdogs → pictures with both! (for testing rudimentary detectors)
- retrain.py → CNN classifier retraining script
- utils.py → TensorFlow computation graph input/output utilities, feel free to take a look!
- classify.py → image classification test script
- detect.py → object detection three ways,

Before you get started, please read through the following to get familiar with Tensorflow and Keras: [TF & Keras Quickstart](#).

Image Classification

First, we concern ourselves with the task of *image classification*. That is, given an image belonging to one of a number of classes (here, “cat”, “dog”, or “neg”(ative) for neither) we would like to associate with each class a probability of the image’s membership.

Here’s the plan²: we (a) download ~ 25 million pre-trained model parameters, (b) chop the pre-trained model off at the layer right before final classification, where it has produced concise vector summaries of input images (the “bottleneck” layer, see Fig. 1), (c) implement a linear classifier³ that takes these feature vector summaries and outputs a probability vector over our classes, and (d) train just this final classifier on our regular computer. The idea is that the pre-trained Inception-v3 model has learned to produce good features for general image classification, so we can take these same features as inputs to our own classifier and train our classifier using our own data. This is a common procedure in many computer vision applications.

- (i)  Take a look at `retrain.py`. First, we pre-compute the output of the Inception-v3 bottleneck layer for all training images. This data will serve as our training dataset for the linear classifier. Fill in `get_bottleneck_dataset()`. You would want to refer to Tensorflow’s ImageDataGenerator guide [here](#).
- (ii)  Next, have a look at the `retrain()` function where the Inception-v3 model is created⁴, the linear

¹In this problem we’ll be classifying cat and dog pictures; technically our model, pre-trained on ImageNet (<http://www.image-net.org/>) datasets and classes, is particularly well-suited to extracting features relevant to small animal classification. If this seems a bit cheat-y, feel free to try this problem with your own truly custom dataset.

²This part is heavily inspired by https://www.tensorflow.org/hub/tutorials/tf2_image_retraining.

³See <http://cs231n.github.io/linear-classify/> for a good overview.

⁴By setting the `include_top` to `False`, the last layer is chopped off. ‘Avg’ pooling parameter will add the average pooling operation at the end of the network.

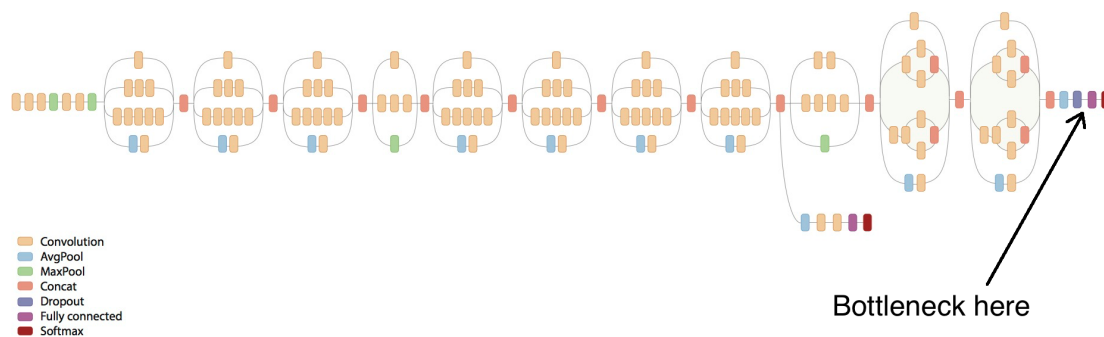




Figure 1: A visualization of the Inception-v3 CNN classifier (~ 25 million parameters) [1]. MobileNets [3] strive to achieve a similar level of accuracy with far fewer parameters.

classifier is defined, and finally trained. Fill in the first the missing code segment to create the linear classifier. Name your retrain layer as "classifier" using the name argument.


- (iii)  Finally, after training we want to merge both Inception-v3 and Linear Classifier models. Create the full model using TensorFlow's [Keras Model API](#) by filling in the rest of `retrain()`.
- (iv)  Now we're ready to test our work. Start the re-training process by

```
$ python retrain.py --image_dir datasets/train
```


We can visualize the progress of the training process by starting up the TensorBoard visualizer in another terminal window:

```
$ tensorboard --logdir=retrain_logs
```

and navigating to <http://127.0.0.1:6006> in your browser. What is the dimension of each "bottleneck" image summary? How many parameters (weights + biases) are we optimizing in this retraining phase?

- (v)  Instead of pre-computing the bottleneck dataset and training the linear classifier on this dataset, we could create and train the full model in the first place and train on the original image dataset. One obvious downside to this approach is that this process takes so much more time, since we're re-doing forward-pass on the entire dataset. However, there are more serious issues with bringing the classifier to convergence with this approach—what might one issue be?

Hint: What happens to training when you have dropout layers?

- (vi)  Now that we've trained our neural network, we can evaluate the performance of our classifier on images it hasn't seen before.

In `classify.py` complete the `classify` function. Make sure to print out the filename of misclassified images. Evaluate the trained model using

```
$ python classify.py --test_image_dir datasets/test/
```

Pretty good, eh? Note the filenames of a few of the misclassified images; we'll revisit them in part (xiii) of this problem.

Object Detection and Localization

Near-human-level image classification is pretty neat, but as roboticists, it is often more useful for us to perform *object detection* within images (e.g., pedestrian detection from vehicle camera data, object recognition and localization for robotic arm pick-and-place tasks, etc.). Traditionally, this means

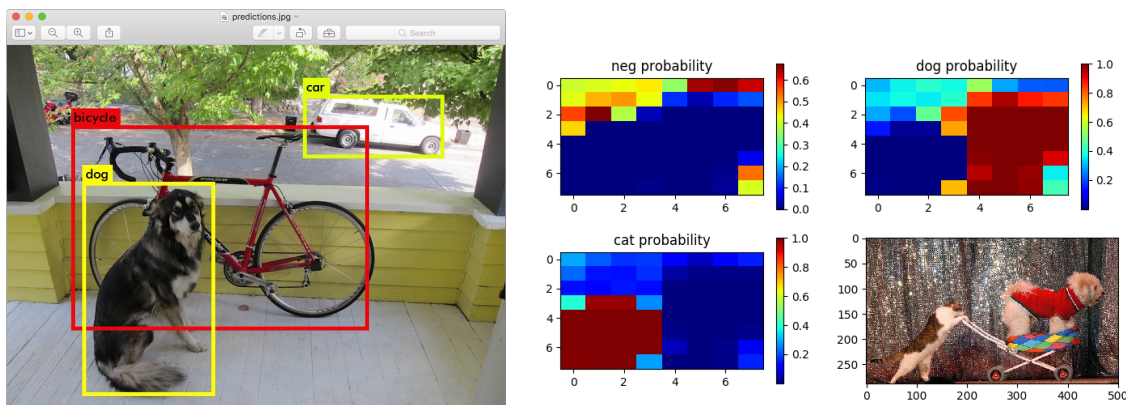


Figure 2: Object detection. On the left, YOLO [4]. On the right, us (sliding window classification).

drawing and labeling a bounding box around all instances of an object class in an image, but we'll settle for a heatmap today (see Figure 2). In practice, achieving state-of-the-art performance in object detection requires training dedicated models with clever architectures (see YOLO [4], SSD [5]), but in the spirit of bootstrapping pre-trained models we can convert our image classifier into an object detector by applying it on smaller sections (“windows”) of the image.

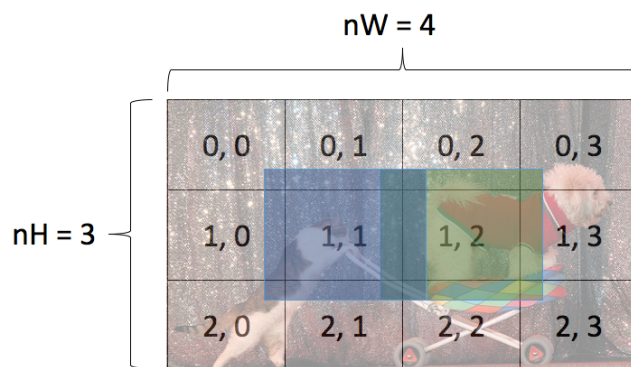





Figure 3: Sliding window with padding (part (ii)). Running a classifier on the blue window might yield an answer of “cat”; running the same classifier on the green window we might expect “dog.”

- (vii)  In `detect.py` complete the `compute_brute_force_classification` function. The arguments `nH` and `nW` indicate how many segments to consider along the height and width of the image, respectively. Evaluating the classifier on the blue window in Figure 3 will yield a probability vector that there is a cat vs. a dog vs. neither at window (1,1). Pad your windows by some amount of your choosing so that the impacts of convolutional edge effects are reduced. Run the detector with the command:


```
$ python detect.py --scheme brute --image <image_path>
```


- (viii)  In addition to filling out `compute_brute_force_classification`, include the detection plot for your favorite image in `datasets/catswithdogs/`.
- (ix)  Messing with indices and computing sliding windows is not only a lot of work for you, but computing *on* them is a lot of work for your computer! There's a slicker way. In the convolution/pooling process

associated with running the classifier on the image as a whole, the final image features are *already* being computed for image sub-regions. That is, instead of running the classification model $nH \cdot nW$ times, we can run it just once and achieve comparable results⁵. Assuming the final convolution layer has an output dimension of $[1, K, K, L]$ ⁶. To classify the entire image we are averaging over dimension 2 and 3 to get a tensor of shape $[1, L]$. We would then run this tensor through the linear classifier to get a class per batch element. Instead we can now classify each $K * K$ patch independently. Thus, we take the convolution output tensor and reshape it to $[1 * K * K, L]$ before running it through our linear classifier.


Add the missing lines `compute_convolutional_KxK_classification` and run this detector with the command:

```
$ python detect.py --scheme conv --image <image_path>
```

(x)  Include in your writeup the detection plot for your favorite image in `datasets/catswithdogs/`.

(xi)  Another simple approach to object localization (finding the relevant pixels in an image containing exactly one notable object) is *saliency mapping* [6]. The idea is that neural networks, complicated and many-layered though they may be, are structures designed for tractable numerical gradient computations. Usually these derivatives are used for training/optimizing model parameters through some form of gradient descent, but we can also use them to compute the derivative of class scores (the output of the CNN) with respect to the pixel values (the input of the CNN). Visualizing these gradients, in particular noting which ones are largest, can tell you for which pixels the smallest change will affect the largest change in class evaluation.

Read Section 3 of [6] and implement the computation of M_{ij} (described in Section 3.1) in the function `compute_and_plot_saliency`. The raw gradients w_{ijc} can be easily computed in Tensorflow using GradientTapes. Get familiar with them [here](#) and fill the missing parts indicated by the comments in the `compute_and_plot_saliency` function.

(xii)  In addition to filling out `compute_and_plot_saliency`, include in your write up the results of running the command:

```
$ python detect.py --scheme saliency --image <image_path>
```

on both a correctly and incorrectly classified image from `datasets/test/`. In particular, for the incorrectly classified image, you may be able to gain some insight into what the CNN is actually looking at when getting it wrong!

⁵The effective (nH , nW) are defined by how the model does its final pooling operation; for Inception-v3 it's (8, 8).

⁶For a single input image.

References

- [1] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception architecture for computer vision,” in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.
- [2] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, “The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results.”
- [3] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” 2017, Available at <https://arxiv.org/abs/1704.04861>.
- [4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 2016, pp. 779–788.
- [5] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “SSD: Single shot multibox detector,” in *Proc. European Conf. on Computer Vision*. Springer, 2016, pp. 21–37.
- [6] K. Simonyan, A. Vedaldi, and A. Zisserman, “Deep inside convolutional networks: Visualising image classification models and saliency maps,” 2013, Available at <https://arxiv.org/abs/1312.6034>.