



CS 237B: Principles of Robot Autonomy II

Problem Set 1: Learning-based Perception and Control

Due Jan 24th 11:59PM

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.

You will submit your homework to Gradescope. Your submission will consist of (1) a single pdf with your answers for written questions (denoted by the  symbol) and (2) a zip folder containing your code for the programming questions (denoted by the  symbol).

Remember, your written part must be typeset (e.g., \LaTeX or Word).

Introduction

For this homework, you will explore different elements of learning-based perception and control. In particular you will investigate the following:

1. Machine learning (SVM) for pedestrian detection
2. Classification and sliding window detection
3. Actor-Critic

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

We strongly encourage you to take a look at all the code, even sections that you will not be directly working on.

IMPORTANT: Install Additional Software Dependencies

Make sure you have Python 3.7 installed. We highly recommend to use a virtual environment such as [Anaconda](#)¹ or [virtualenv](#)².

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

¹<https://anaconda.org>

²<https://virtualenv.pypa.io/en/latest/>

- Download and install Anaconda (Python 3.7 version) from <https://www.anaconda.com/distribution/>
- Open a terminal and run the following command

```
$ conda create -n cs237b python=3.7.5 -y
```

- To activate the virtual environment run

```
$ conda activate cs237b
```

Important: Every time you re-open the terminal window you will have to re-run this command to activate the environment!

You will require a few additional software dependencies (Tensorflow, Matplotlib...) to complete the remaining problems. To maintain consistency within this class, we already listed all dependencies on `requirements.txt` in the git repo you cloned. Navigate to the repo. If you use a virtual environment, run:

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

You might have to use `--user` based on your python setup.

Finally, download the dataset files from: <https://stanford.box.com/s/7uccz78ikgqnvckq0y2m3tb3z46j3mvf>. Move `pedestrian_dataset.npz` to `Problem_1` folder. Unzip `P4_cats_and_dogs.zip`, and place the resulting `datasets` folder in the `Problem_2` folder. Now we're ready to get started with the first homework.

Problem 1: Tensorflow and HOG+SVM Pedestrian Detection

Computer vision for robotics often involves processing complex and high dimensional data (up to millions of pixels per image) and extracting relevant features that can be used by other components in a robotic autonomy stack. Nowadays, many computer vision techniques rely on deep learning and machine learning algorithms for classification and depend heavily on computational tools that can efficiently process, learn, and do inference on the data.

In this problem, you will familiarize yourself with TensorFlow as a tool for machine learning. You will learn the basics of TensorFlow by implementing a Support Vector Machine (SVM), a typical machine learning classification algorithm, on Histogram of Oriented Gradients (HOG) image descriptors to identify pedestrians in images. HOG is a technique for detecting and extracting edges of objects in an image. By detecting the edges, HOG produces features of an image which can be used for many machine learning algorithms.

Before you get started, please read through the following jupyter notebooks on colab to get familiar with Tensorflow:

- [Tensors and Operations](#)
- [Automatic Differentiation](#)
- [Custom Training](#)

Support Vector Machines (SVM) are supervised learning models that classify data. In its simplest form, an SVM finds a decision boundary between two categories of data and classifies each datapoint $x^{(i)}$ based on which side of the boundary it lies on. In this problem, we will assume that the decision boundary is a hyperplane $xw = b$ (a linear SVM)³, and thus the prediction is of the form

$$\tilde{y}^{(i)} = \text{sgn}(x^{(i)}w - b)$$

³In Python-based machine learning computation frameworks, the convention is for datapoints to be row vectors so that a list of datapoints $[x^{(1)}, x^{(2)}, \dots, x^{(n)}]$ corresponds to a matrix with the datapoints as rows. This is why in this problem we write xw instead of $w^T x$.

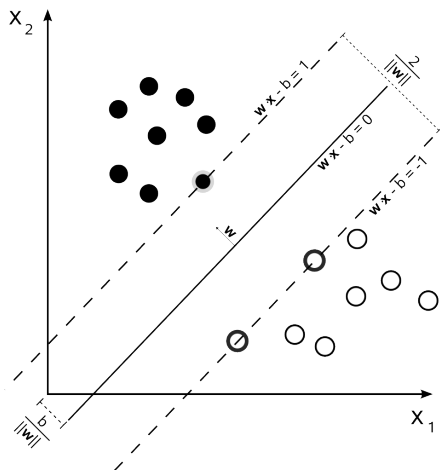


Figure 1: Illustration of a linearly separable dataset.

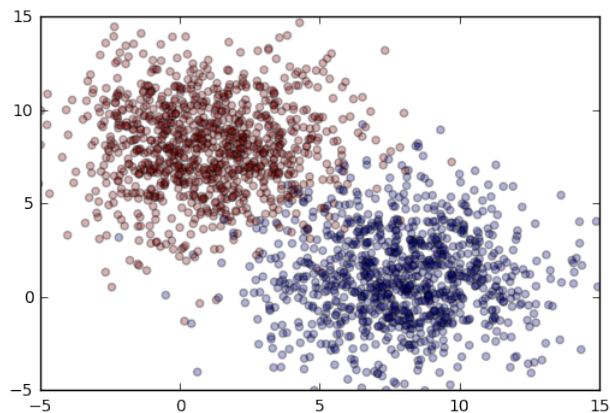


Figure 2: Toy dataset that is not linearly separable.

Suppose the data we have is labeled with either $+1$ or -1 (pedestrian or not a pedestrian, cat or dog, tree or signpost, etc.). This can be visualized in Figure 1 where the black dots represent a label of $+1$ and the white dots represent a label of -1 . For simplicity, we will first consider the case where the data is separable (no overlap) and can be separated by a straight line.

This means we can select two parallel hyperplanes that separate the two classes such that the distance between them is as large as possible. The region between the two hyperplanes is called the “margin” (dashed lines in Figure 1) and the hyperplane that lies halfway is called the maximum-margin hyperplane. Maximizing the region is equivalent to solving the following optimization problem:

$$\min_w \|w\|_2^2 \quad \text{subject to} \quad y^{(i)}(x^{(i)}w - b) \geq 1$$

Now suppose that the data is not separable (see Figure 2). Notice that there are some points that overlap in the data so the notion of a maximum-margin hyperplane as an optimal classifier cannot be applied. In this case, optimality can be defined with respect to the following. First, we introduce the hinge loss:

$$\ell_{\text{hinge}}(x, y) = \max(0, 1 - y(xw - b))$$


This loss function penalizes points for being inside the margin ($xw - b \in [-1, 1]$) and especially penalizes points on the wrong side of the hyperplane, i.e., when $(xw - b)$ and y have opposite signs. Thus, now not only do we want to maximize the margin, but also minimize the hinge loss.


Thus, given a dataset $(x^{(1)}, x^{(2)}, \dots, x^{(n)})$ with correct labels $(y^{(1)}, y^{(2)}, \dots, y^{(n)})$, the “soft-margin” linear SVM optimization problem is


$$\min_{w, b} \frac{1}{n} \sum_{i=1}^n \max(0, 1 - y^{(i)}(x^{(i)}w - b)) + \lambda \|w\|^2 \quad (1)$$

where λ is a hyperparameter which gives the relative weighting for the two terms.

It is important to note that although Equation (1) describes a linear classifier, we can achieve nonlinear behavior by selecting an arbitrary basis function $x^{(i)} = \Phi(x)$ of the true input data (indeed, this is one of the greatest strengths of SVMs). Examples include adding higher-degree monomials in addition to xy -position values for 2D data, learning a linear classifier on features extracted from images instead of on the raw pixel values themselves, etc.

- (i)  Using the notation in Figure 1, prove that the perpendicular distance between the two planes is equal to $\frac{2}{\|w\|}$ (and thus minimizing $\|w\|$ is equivalent to maximizing the margin).

- (ii)  Take a look at `svm_tf.py`. We will edit three functions: `call`, `loss` and `svm`. First, edit the `call`⁴ function to compute $y_{\text{est}} = xw - b$ during the training case, and to compute the output label \tilde{y} for the prediction case. Next, edit the `loss` function to return the soft-margin hinge SVM loss as described above. Lastly, finish implementing the `svm` function by selecting your hyperparameters and completing the training procedure (`train_step` and `eval_step`).

- (iii)  Run your TensorFlow model on a linearly separable dataset by running


```
$ python svm_tf.py --type linear
```

How does your SVM do? Include the plot in your write up.

- (iv)  Run your TensorFlow model on a nonlinear dataset by running

```
$ python svm_tf.py --type non_linear
```

How does your SVM do? Explain why it makes the misclassifications the way that it does. Include the plot in your write up.


- (v)  In the previous part, identity features were used as basis function Φ i.e., x_1 and x_2 were the features used for the SVM. We can denote this as $\Phi(x) = x$: our basis functions Φ were the features themselves. Unfortunately, linear basis functions are often not sufficient, as shown with the last example. Instead we can use different basis functions mapping the inputs into a space where the data is linearly separable (or at least better separable). We can see this by looking at a simple example. First, run your Tensorflow model by running

```
$ python svm_tf.py --type circle
```

and


```
$ python svm_tf.py --type inner_circle
```

and inspect the results. Comment on how your classifier performed, and why. Intuitively, what basis functions should be used for a best-performing SVM classifier? Include the plots and observation in your write-up.

- (vi)  Based on your previous answer, implement basis functions `circle_phi` and `inner_circle_phi` in `svm_tf.py` such that the problems become better separable in the projected space. After you finished implementing the functions, re-run the scripts with the additional `--feature custom` flag. E.g.

```
$ python svm_tf.py --type circle --feature custom
$ python svm_tf.py --type inner_circle --feature custom
```

Include the plots from both in your write up.

- (vii)  SVM is a linear classifier. You might have noticed, however, that the basis functions that you implemented for the circular dataset are clearly nonlinear. Explain why SVM is still a linear classifier.

⁴When sub-classing the `Model` class from TensorFlow, the method representing the forward pass during training has to be named `call`. Sub-classing, however, brings the advantage that all trainable weights are collected automatically for backpropagation.

- (viii)  Fortunately for you(!), we have implemented HOG feature vector extraction [1] in TensorFlow for you.⁵

You can explore the dataset and the HOGs in [hog_visualization.ipynb](#). Explore the dataset, and the associated HOG features. Your task is to set up the datasets for training, evaluation and prediction in the [get_hog_data](#) function in [svm_tf.py](#). Make sure your labels correspond to the correct images. You do not need to worry about shuffling the data, this is automatically done within the workflow (see [tf.data.Dataset](#)).

- (ix)  Run your SVM using HOG features for pedestrian classification, by typing

```
$ python svm_tf.py --type hog
```

What is the classification accuracy for this model? The above command should save your model in the [hog_model/](#) folder. Visualize and explore your results in the notebook by changing the data index, k . Notice that the HOG elements with positive weight correspond vaguely to the outline of a pedestrian (see Figure 6 in [1]). Submit a visualization of HOG features overlapped on a pedestrian image of your choice.

⁵See HOG.py. Typically, this step might be implemented in Numpy, rather than TensorFlow, since the HOG descriptors are used as inputs $x^{(i)}$ to the learning process, but aren't actually involved in the training process (i.e., extracting HOG feature vectors is similar to using custom features, rather than the identity feature, which would be raw pixels for the case of images). But we wanted to show you that TensorFlow can be a general-purpose computation framework. And if you're really ambitious, you might even try putting in TensorFlow variables instead of the constant x/y-convolution kernels to make this a "deep" neural network!

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 [2], 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.⁶ **Problem Setup:** We will be using the open source TensorFlow library (<https://www.tensorflow.org/>) to perform the numerical computations involved in training and evaluating neural networks in this problem. Make sure you have P4_cats_and_dogs folder unzipped from the earlier dataset. Take a look at the directory. The files for this problem should be organized as:

- `datasets/` → labeled images from the PASCAL Visual Object Classes Challenge 2007 [3]
 - `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 jupyter notebooks on colab to get familiar with Tensorflow 2.0 and Keras:

- [Quickstart 1](#)
- [Quickstart 2](#)

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 plan⁷: 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. 3), (c) implement a linear classifier⁸ that takes these feature vector summaries and outputs a probability vector over our classes, and (d) train just this final classifier

⁶In 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.

⁷This part is heavily inspired by https://www.tensorflow.org/how_tos/image_retraining/.

⁸See <http://cs231n.github.io/linear-classify/> for a good overview.

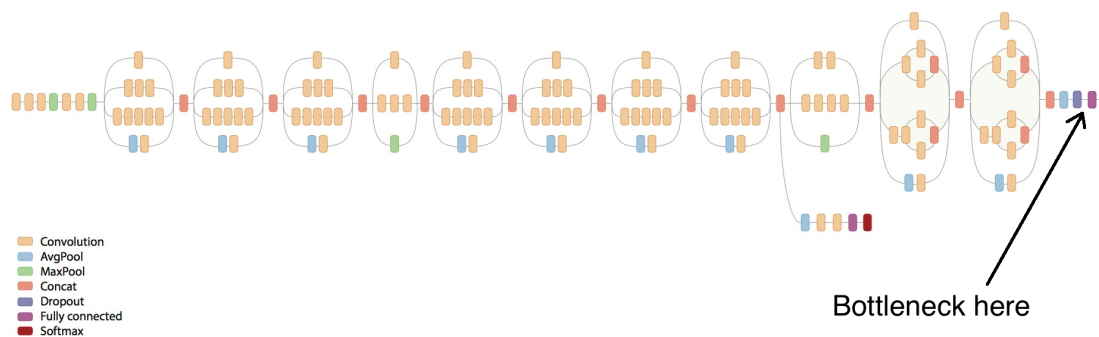






Figure 3: A visualization of the Inception-v3 CNN classifier (~ 25 million parameters) [2]. MobileNets [4] strive to achieve a similar level of accuracy with far fewer parameters.

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 created⁹, the linear classifier is defined, and finally trained. Fill in the first the missing code segment to create the linear classifier.
- (iii)  Finally, after training we want to merge both Inception-v3 and Linear Classifier models. Create the full model using TensorFlow's [Sequential 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. Look at the GRAPHS tab in TensorBoard to see the model structure of both the Inception-v3 and our classifier (should be all the way at the bottom of the graph). What is the dimension of each “bottleneck” image summary?¹⁰ How many parameters (weights + biases) are we optimizing in this retraining phase?

⁹By 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.

¹⁰The “?” in the first dimension of a Tensor indicates that this graph node can take an arbitrary number of vectors (a batch of vectors) as input (recall that row vectors correspond to data points). This question is asking for the dimension of each bottleneck vector.

- (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

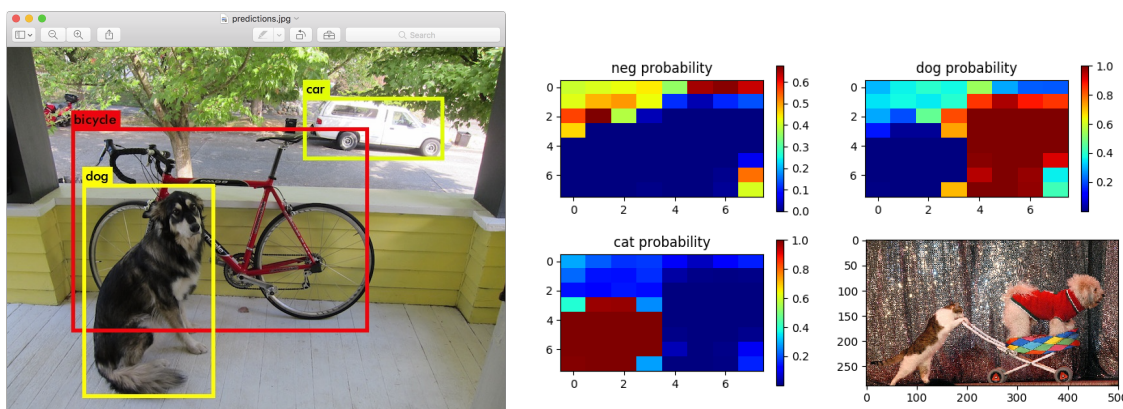



Figure 4: Object detection. On the left, YOLO [5]. On the right, us (sliding window classification).

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 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 4). In practice, achieving state-of-the-art performance in object detection requires training dedicated models with clever architectures (see YOLO [5], SSD [6]), 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.

- (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 5 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:

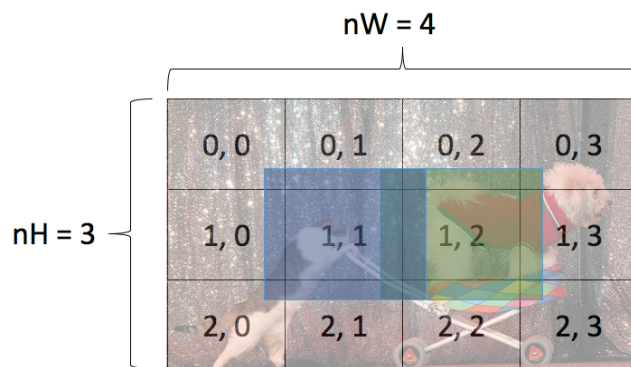


Figure 5: Sliding window with padding (part (ii)). Running a classifier on the blue window might yield an answer of “cat”; running the same classifier on the green window we might expect “dog.”

```
$ 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) In the TensorBoard GRAPHS tab, find the output of the final convolutional layer (`mixed_10`). What operation does it feed into? How is the feature vector for the image as a whole computed from the feature vectors for each image region?
- (x) 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 results¹¹. Assuming the final convolution layer has an output dimension of $[1, K, K, L]$ ¹². 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>
```

- (xi) Include in your writeup the detection plot for your favorite image in `datasets/catswithdogs/`.
Another simple approach to object localization (finding the relevant pixels in an image containing exactly one notable object) is *saliency mapping* [7]. 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

¹¹The effective (nH, nW) are defined by how the model does its final pooling operation; for Inception-v3 it’s (8, 8).

¹²For a single input image.

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.

- (xii)  Read Section 3 of [7] 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.
- (xiii)  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!

Problem 3: Actor-Critic Methods

In class, we surveyed a few different reinforcement learning algorithms. Many reinforcement learning algorithms approximate the value function using a finite set of parameters θ , i.e., $V_\theta(s) \approx V^\pi(s)$ or $Q_\theta(s, a) \approx Q^\pi(s, a)$. For large state and action spaces, an exciting new approach involves approximating these functions using neural networks. Figure 6 describes the reinforcement learning process that allows us to regress the relevant parameters of our function approximators (our neural networks).

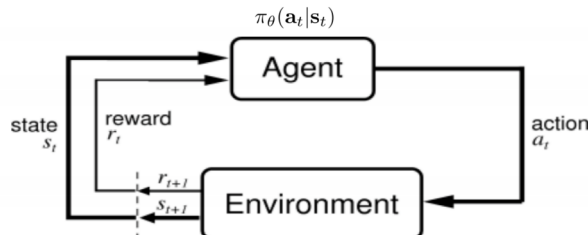


Figure 6: Anatomy of reinforcement learning.

In this problem, we will first dive deep into so-called policy gradient methods, that directly learn policies without explicitly learning value functions. Then, we will address the high-variance issue of policy gradient methods by reintroducing the value function in the learning process as part of the ‘actor-critic’ method. By the end of this problem, you will implement the critic component of an actor-critic algorithm, and see how your actor-critic solves a classic cart-pole (aka inverted pendulum) problem.

To pique your interest, let’s look at the cart-pole environment. Run

```
$ python cartpole-test.py
```

In case you’re not familiar with the cart-pole problem, your objective is to move the cart on the bottom just right to make the pole stand upright without going off screen. By the end of this problem, you will have a controller that successfully solves this problem. Let’s get started.

Policy gradient methods

Let’s define a parameterized policy $\pi_\theta(a_t|s_t)$ as a probability distribution over actions given a state of the system. Now recall that the objective of reinforcement learning is to find parameters (θ) of that policy such that they maximize an objective function J . More specifically, for this question, we will define our objective function as the expected reward over a finite horizon T given a Markov chain episode τ .

$$J(\theta) = \mathbf{E}_{\tau \sim \pi_\theta(\tau)}[r(\tau)], \quad (2)$$

where

$$r(\tau) = \sum_{t=0}^{T-1} r(s_t, a_t), \quad (3)$$

and

$$\pi_\theta(\tau) = p(s_0) \pi_\theta(a_0|s_0) \prod_{t=1}^{T-1} p(s_t|s_{t-1}, a_{t-1}) \pi_\theta(a_t|s_t). \quad (4)$$

Policy gradient methods are a popular class of reinforcement learning methods that attempt to learn such policies by directly taking the gradient of this objective.

We can show

$$\nabla_{\theta} J(\theta) = \mathbf{E}_{\tau \sim \pi_{\theta}(\tau)} [\nabla \log \pi_{\theta}(\tau) r(\tau)] \quad (5)$$

$$\approx \frac{1}{N} \sum_{i=1}^N \nabla \log \pi_{\theta}(\tau_i) r(\tau_i) \quad (6)$$

$$= \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_{it} | s_{it}) \right) \left(\sum_{t=0}^{T-1} r(s_{it}, a_{it}) \right). \quad (7)$$

The core of policy gradient is to maximize performance, so we update our parameters using gradient ascent, i.e.,

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla_{\theta} J(\theta_t)}. \quad (8)$$

Note that $\nabla_{\theta} \hat{J}(\theta)$ is a stochastic **estimate** of our gradient. All methods that follow this general idea are called policy gradient methods. By adding in the additional assumption of ‘causality’ that roughly says that the action at time t' cannot affect the reward at time t when $t < t'$ (the Markov property from the underlying MDP), we can improve our estimate of the gradient by modifying it to

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_{it} | s_{it}) \left(\sum_{t'=t}^{T-1} r(s_{it'}, a_{it'}) \right) \right), \quad (9)$$

(note the change in the starting index of the sum of rewards).

Variance reduction

In practice, the above estimate of the objective’s gradient can be quite noisy, i.e. have a *high variance*. We will therefore make two modifications to the estimate to reduce this variance. These changes have more principled explanations as well, but those explanations are beyond the scope of this class. The first modification we will make is to add a discount factor $\gamma < 1$ to our estimate of the ‘reward-to-go’




$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_{it} | s_{it}) \left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'}) \right) \right). \quad (10)$$


Next, we will add a ‘baseline’ reward estimate

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_{it} | s_{it}) \left(\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'}) \right) - V_{\pi_{\theta}, \phi}(s_{it}) \right) \right). \quad (11)$$

The baseline we added is referred to as an approximation of the ‘reward-to-go’ function, or our best estimate of V^{π} parameterized by ϕ .

$$V_{\pi_{\theta}, \phi}(s_t) \approx \sum_{t'=t}^{T-1} \mathbf{E}[r(s_{t'}, a_{t'}) | s_t]. \quad (12)$$

- (i)  We skipped a few steps between Equation (2) and Equation (5). Show $\nabla_{\theta} J(\theta) = \mathbf{E}_{\tau \sim \pi_{\theta}(\tau)} [\nabla \log \pi_{\theta}(\tau) r(\tau)]$. Hint: Use identity $f(t) \nabla_x \log f(t) = \nabla_x f(t)$.
- (ii)  Justify why Equation (9) works. In particular, comment on the direction/magnitude of updates in Equation (8). We’re only looking for justification, not a formal proof.
- (iii)  Given the intuition developed in part (ii), explain why policy gradient is susceptible to high variance.

- (iv)  When we added the discount factor γ to our gradient estimate in (10), we made sure to specify this one must be smaller than 1 (i.e. $\gamma < 1$). Why is that? For a given time t on episode i , think about how γ scales the different rewards for $t' > t$.

Actor-Critic methods

Actor-critic methods are a sub-class of policy gradient methods that split the policy gradient task into two function approximation tasks. One of them is the learning of a good policy π_θ , referred to as the ‘actor’, and the other is the learning of the reward-to-go function $V_{\pi_\theta, \phi}(s)$, referred to as the ‘critic’. Imagine you’re learning to add two numbers a and b with a TA. In the beginning, you don’t have any knowledge about this problem, so you solve stochastically. For your first problem (‘episode’), say $a = 1, b = 1$, your action could be ‘-12’. A good TA would give you a negative reward to this action, and you would adjust your policy on how to answer this question accordingly. After a few trials, you perform a good action (in this case ‘2’) to gain a positive reward from your TA. You would adjust your policies to learn how to perform addition, and then receive a new problem (say, $a = 2, b = 3$). Your learning episode continues until your learning is completed. In this example, you’re an actor who is consistently adjusting your best guess of the optimal policy (an action a that will maximize your reward r given state s), and the TA is the critic. In real world, however, we don’t have a TA, or knowledge of true optimal values, so we train both the actor and critic in parallel.

There are lots of different ways of accomplishing either tasks leading to the multiple variants of actor-critic methods. Here, we will ask you to implement an algorithm that implements the critic part of an actor-critic method using the temporal difference error. In your implementation, you’re free to explore, but please keep [Actor](#) intact to ensure your Homework 2 code runs without any issue.

Taking our estimate of $\nabla_\theta J$, we can rewrite the estimate of the reward-to-go as

$$\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{t'}, a_{t'}) \right) - V_{\pi_\theta, \phi}(s_t) \approx r(s_t, a_t) + \gamma V_{\pi_\theta, \phi}(s_{t+1}) - V_{\pi_\theta, \phi}(s_t). \quad (13)$$

The quantity on the right is referred to as the *temporal difference error* or TD error,

$$TD_{\text{error}} := r(s_t, a_t) + \gamma V_{\pi_\theta, \phi}(s_{t+1}) - V_{\pi_\theta, \phi}(s_t). \quad (14)$$

Roughly speaking, the Bellman equation says that the TD error of a valid ‘reward-to-go’ function must equal zero. It is therefore possible to train a function approximator by minimizing the TD error. This can be done in various ways, but we choose to generate ‘labels’


$$y_t = r(s_t, a_t) + \gamma V_{\pi_\theta, \phi}(s_{t+1}) \quad (15)$$

and minimize the squared error of those predictions

$$\min_{\phi} \sum_{i,t} (V_{\pi_\theta, \phi}(s_{it}) - y_{it})^2. \quad (16)$$

Equipped with our TD error, we can now train our two function approximators quite efficiently.

Let’s summarize. Actor-critic is a type of policy gradient method. Specifically, we run two neural networks in parallel to learn V^π and π_θ simultaneously. The actor is a neural network that learns π_θ to minimize J . The critic is a neural network that learns ϕ to approximate V_{π_θ} by minimizing the squared TD error. Now we’re ready to look at the code.

- (iv)  Take a look at the [Actor](#) class in [A2C.py](#). How many layers are there in the Actor neural network? What type of activation functions were used?

In `A2C.py`, have a look at the function `train_actor_critic`. The first thing it does is initialize an environment using OpenAI's gym package¹³ that you saw in the beginning of this problem. The cart-pole environment has a state space as well as an action space. The cart-pole's state space is a four dimensional vector corresponding to the cart's position (x), the cart's velocity (\dot{x}), the pole's angle (θ) and the pole's angular velocity ($\dot{\theta}$), in that order.

$$[x, \dot{x}, \theta, \dot{\theta}]$$


The possible actions that can be applied to the environment are to push the cart towards the left (action 0), or push it towards the right (action 1). After initializing the actor and the critic, the function enters a training loop. The training loop consists of running a number of episodes, where an episode is a sequence of 200 time steps (the gym environment defines this maximum episode length). A transition happens every time `env.step` is called. As you can see, this function returns the next state but also the reward for that transition. The cart-pole environment is setup to simply return a reward of 1 for every step it takes until termination. Termination happens when one of the following happens:


- Pole angle is $\pm 12^\circ$
- Cart position is more than $\pm 2.4m$
- Episode length is greater than 200

The training loop adds each transition (state, action, reward, next state) to a replay buffer (a simple array containing recently observed transitions).

Next is the training of the function approximators. First, the critic evaluates the temporal difference error, and uses it to improve its current approximation of the value function (by calling the critic's `train_step`). This is essentially what we are asking you to implement. Then, the evaluated temporal difference error is passed on to the actor, that uses it to improve the current policy. The reward accumulated over each episode is printed in your terminal as the training progresses.

This whole process repeats until the maximum number of episodes is reached, or if you press Ctrl+C. The actor is then run for a few 'test' episodes that will also render for you. The last couple of lines export your actor network, which will become relevant on the next homework.

- (v)  In `A2C.py`, finish implementing the class `Critic` so that it performs the duty of a critic in our actor-critic implementation. You can look at how the `Actor` was implemented for a template. As a hint, `train_step()` is already implemented for you so that you should only have to setup the computation graph in the constructor, but feel free to modify it. The `train_step` method should take a `batch` of state, rewards and next states and return a tensor containing the estimated TD error for each sample in the batch. Moreover, it should run the training operation `train_op` that is defined in the class constructor.

- (vi)  Time to run! Test that your `Critic` network is working for the classic cart-pole problem. Run

```
$ python A2C.py
```

and see your Actor-Critic neural network in working. Include the plot of the rewards accumulated over each training episode in your write-up.

If your actor-critic neural network isn't giving you the maximum score of 200, don't stress! We will grade based on how often your actor gets a high score (you should be able to get at least above 100 somewhat consistently) and the correctness of your implementation. Also keep in mind actor-critic algorithms sometimes require a certain amount of tuning to work well in practice.

¹³<https://gym.openai.com/>

References

- [1] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1. IEEE, 2005, pp. 886–893.
- [2] 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.
- [3] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, “The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results,” Available at <http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html>.
- [4] 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/1704.048614>.
- [5] 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.
- [6] 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.
- [7] 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>.