CSCI 699: Robot Learning
Problem Set #1: Due Sun, Sep 22, 11:59PM

Introduction

The coding portion of the homework assignment will be completed using Google Colab. Starter code for this problem set can be downloaded from https://github.com/USC-Lira/CSCI699_RobotLearning_HW1.git. The notebook should contain the code for installing all the necessary Python dependencies.

Submission instructions:

You will submit your homework to Gradescope. Your submission will consist of (1) a single pdf with your answers for the short answer questions (✍️) and (2) a zip folder containing the Colab notebooks for the programming questions (💻).

Your answers to the written portion must be typeset with a word processor or LaTeX.

Problem 1: Acrobot Forward and Inverse Kinematics [20 points]

(i) ✍ [3 points] **Degrees of Freedom**

Consider an open-chain robot manipulator in 3D space with the following configuration:

The manipulator consists of four links connected by joints.

(a) The first link is attached to a fixed base.

(b) The second link is attached to the first using a revolute joint.

(c) The third link is attached to the second using a prismatic joint.

(d) The fourth link is connected to the third link using a spherical joint.

Identify the number of joints in the system. Calculate the total degrees of freedom for this robot manipulator.

(ii) ✍ [5 points] **Acrobot Forward Dynamics**

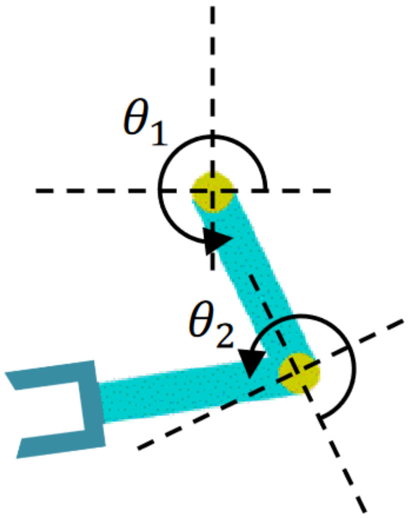

Figure 1: Acrobot

Recall the Acrobot agent, a simple two-link robotic system connected by hinges. The first link is fixed and the second link can rotate about the hinge joint that connect the two links together.

The lengths of the two links are denoted as l_1 and l_2, respectively. The joint angles in radians of the two links, as shown in Fig. 1, are denoted θ_1 and θ_2, respectively.

Compute the forward kinematics (position and orientation) of the tip of the Acrobot's second link in terms of the link lengths and joint angles generally. Then solve for the case when $l_1 = 2, l_2 = 3, \theta_1 = 20°$ and $\theta_2 = 30°$.

(iii) ✍ [6 points] **Acrobot Inverse Kinematics**

Given the desired position (x, y) of the tip of the second link, find the joint angles θ_1 and θ_2 that will achieve this configuration. Provide the formulation for the general case, and then solve for the case when $l_1 = 2, l_2 = 3, (x, y) = (0.63, 4.31)$.

(iv) ✍ [6 points] **Predict link lengths**

You are provided a dataset of joint angles and corresponding end-effector positions $(\theta_1, \theta_2, x, y)$ collected from an Acrobot of fixed link length. However, the recorded (x, y) positions are subject to some zero-mean Gaussian noise. Your task is to, using any method (linear algebra or machine learning), predict the true link length of the Acrobot used to generate this dataset. The dataset can be found in the homework GitHub.

Problem 2: Classification[1] [40 points]

Take a look at the Colab notebook provided with this problem: `p2_image_classification.ipynb`. You will first download a folder containing labelled images from the PASCAL Visual Object Classes Challenge 2007 [1].

The directory should be organized as:

1. `datasets/` → labelled images from the PASCAL Challenge
   - `datasets/train` → training images with labels for each category
     - `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
     - `datasets/train/cat`
     - `datasets/train/dog`
     - `datasets/train/neg`

Image Classification [20 points]

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 plan: we (a) download a ~ 25 million pre-trained model parameters [2], (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 classifier 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.

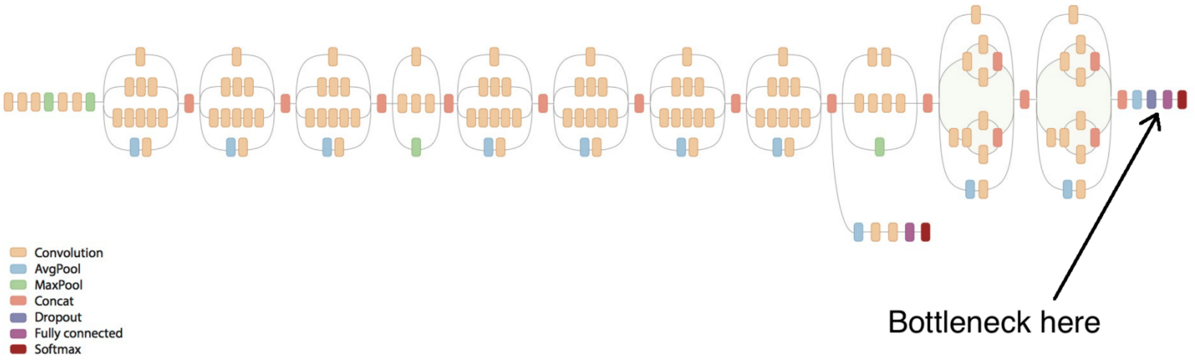

Figure 2: A visualization of the Inception-v3 CNN classifier (~ 25 million parameters) [2].

(i) 💻 [5 points] First, we pre-compute image embeddings of the Inception-v3 bottleneck layer for all training images. This data will serve as our training dataset for the linear classifier. Refer to Section 1 in the notebook.

(ii) 💻 [5 points] Fill in the missing code segment to create the linear classifier in Section 2.

(iii) ✍️ [5 points] 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?

(iv) 💻 ✍️ [5 points] Now that we’ve trained our neural network, we can evaluate the performance of our classifier on images it hasn’t seen before. You will want to merge the pretrained Inception-v3 bottleneck and trained linear classifier into a full model. Fill in the code in Section 3 to classify the test images and compute the model’s accuracy. Report the model’s accuracy. Make sure to print out the filename of misclassified images, we will revisit them in the next section.

[1] This question is based on a question from Stanford CS237B.

Object Detection and Localization [20 points]

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 2). In practice, achieving state-of-the-art performance in object detection requires training dedicated models with clever architectures (see YOLO [3], SSD [4]), 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.

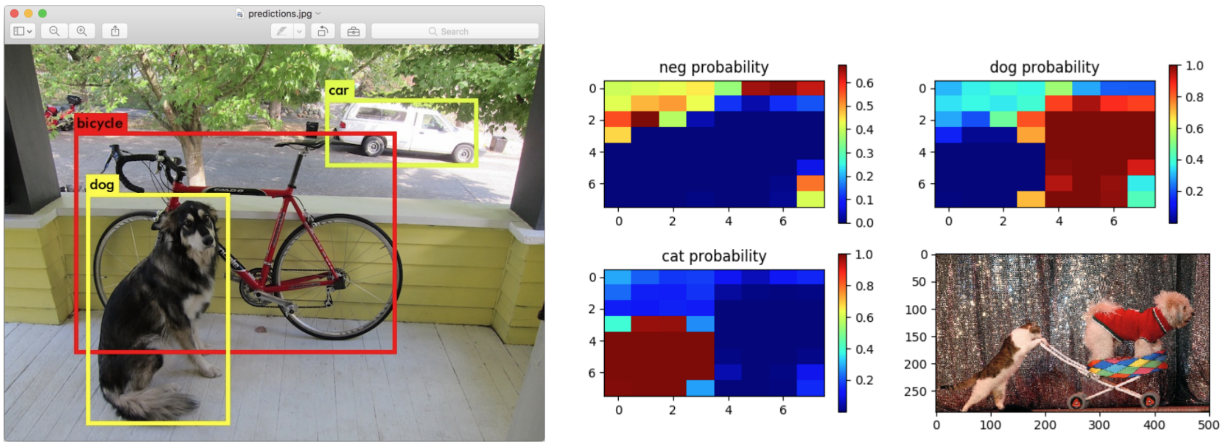

Figure 3: Object detection. On the left, YOLO [3]. On the right, us (sliding window classification).

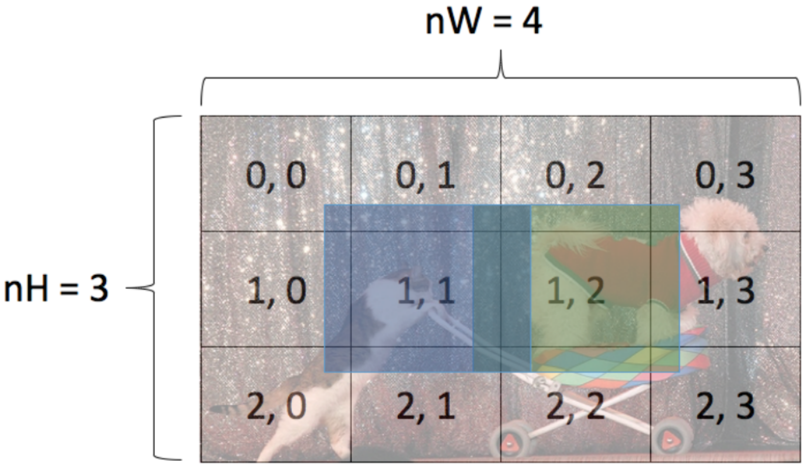

Figure 4: Sliding window with padding (part (ii)). Running a classifier on the blue window might yield an answer of "cat"; running the same classifier on the green window we might expect "dog.".

(i) 💻 In Section 4, 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.

(ii) ✍ [6 points] Run the detector and include the detection plot for your favorite image in `datasets/catswithdogs/`.

(iii) 💻 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` · `nW` times, we can run it just once and achieve comparable results. Assuming the final convolution layer has an output dimension of $[1, K, K, L]$. 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. Complete the `compute_convolutional_KxK_classification` function.

(iv) ✍ [7 points] Include in your writeup the detection plot for your favorite image in `datasets/catswithdogs/`.

(v) 💻 Another simple approach to object localization (finding the relevant pixels in an image containing exactly one notable object) is saliency mapping [5]. 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 [5] and implement the computation of Mij (described in Section 3.1) in the function `compute_and_plot_saliency`. The raw gradients w_{ijc} can be easily computed. Get familiar with them here and complete the `compute_and_plot_saliency` function in Section 5 of the notebook.

(vi) ✍ [7 points] Include in your writeup the saliency plots for one correctly classified and one 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: Representation Learning [40 points]

In this problem, you will explore utilizing autoencoders to learn a compressed representation of high-dimensional image data.

First, you will directly train autoencoders for image via maximum likelihood methods. Next, you'll compare these results to a more Bayesian approach, the VAE [6]. In a bonus task, you will implement Conditional VAEs, which extends vanilla VAE to include conditioning information in both the encoding and decoding process. We consider the MNIST dataset containing 28 x 28 binary images of hand-written digits. For more background on the dataset, see: here.

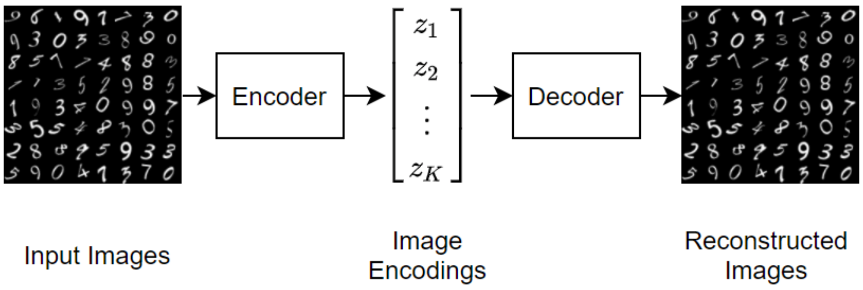

Figure 5: An autoencoder is a model that learns to reconstruct the input by transforming the input into a lower dimensional space.

Template code is provided for you in `p3_representation_learning.ipynb`.

Autoencoder and Variational Autoencoder [30 points]

(i) 💻 [2.5 points] You will train 3 separate models with different bottleneck dimensions: 32, 128, and 512. This is the size of the output of the encoder and input to the decoder. Complete the `Autoencoder` model class in Section 1 of the notebook.

(ii) ✍ [2.5 points] Plot the binary cross entropy loss (y-axis) for both the training and test sets versus the number of training iterations (x-axis). Show the train curves with a solid blue line and test curves with a solid red line. Make a single plot containing 1 row and 3 columns, one column per model. Note, the code for this is not provided. Create a new cell and insert your own code to complete this task.

(iii) ✍ [2.5 points] Comment on the values of the losses between the three different model architectures on the training and test sets. Is there overfitting? Underfitting?

(iv) ✍ [2.5 points] For each model, sample and plot a grid of 25 images from the trained model.

(v) 💻 [2.5 points] t-SNE plots are a useful tool for visualizing high-dimensional data in a lower-dimensional space while preserving the structure between the data points. Create t-SNE plots of the autoencoder embeddings. Color each point by its class label (digit 0 gets one color, digit 1 gets another color, etc). Show a representative set of samples for each class to observe the structure of the learned latent space. Note, the code for this is not provided. Create a new cell and insert your own code to complete this task.

(vi) ✍ [2.5 points] Attach the generated t-SNE plots for each model. What insights can you observe from the t-SNE plots? Comment on any differences you notice between the plots for the vanilla Autoencoder and VAE.

(vii) [15 points] Repeat the above tasks for the Variational Autoencoder.

(viii) 💻 Complete the Conditional VAE implementation to allow conditioning on the digit class during the encoding and decoding process.

(ix) ✍ [5 points] With the trained conditional decoder, sample the learned latent space to generate 3 variations of each digit class. You should have a plot containing 30 images.

(x) ✍ [5 points] Visualize the learned latent space of the Conditional VAE model with the same t-SNE visualization code. How does the t-SNE plot for CVAE compare to those of VAE and vanilla autoencoder?

References

[1] Mark Everingham and John Winn. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results.

[2] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception architecture for computer vision,. *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[3] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection,. *IEEE Conf. on Computer Vision and Pattern Recognition*, 2016.

[4] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. SSD: Single shot multibox detector. *European Conference on Computer Vision*, 2016.

[5] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps,. *arXiv preprint arxiv.org:1312.6034*, 2013.

[6] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.