Homework 5

15-381/781: Artificial Intelligence (Fall 2016)
OUT: Nov. 21, 2016
DUE: Dec. 2, 2016 at 11:59pm

Instructions

Homework Policy

Homework is due on autolab by the posted deadline. Assignments submitted past the deadline will incur the use of late days.

You have 6 late days, but cannot use more than 2 late days per homework. No credit will be given for homework submitted more than 2 days after the due date. After your 6 late days have been used you will receive 20% off for each additional day late.

You can discuss the exercises with your classmates, but you should write up your own solutions. If you find a solution in any source other than the material provided on the course website or the textbook, you must mention the source. You can work on the programming questions in pairs, but theoretical questions are always submitted individually. Make sure that you include a README file with your andrew id and your collaborator's andrew id.

Submission

Please create a tar archive of your answers and submit to Homework 5 on autolab. You should have three files in your archive: a completed `problems.py` and `sbproblems.py` for the programming portion, a PDF for your answers to the written component, and a README file with your Andrew ID and your collaborators' Andrew IDs. Your completed functions will be autograded by running through several test cases and their return values will be compared to the reference implementation. There is a `sbsample.txt` that contains sample inputs and outputs for reference.

1 Written

1.1 Social Choice (15 points)

In this problem, we will consider a natural definition for the ranking of the alternatives of an election, called the maximum likelihood estimation (MLE) ranking. Recall the noise model on slide 17 of Lecture 20. That is, for some $p \in (1/2, 1)$, we assume that each voter gets each pairwise comparison correctly with probability p (i.e., he agrees with the "ground truth" ranking of the alternatives), and the results are tallied in a voting matrix. The MLE ranking is then the ranking that maximizes the probability that we obtain the given voting matrix under this noise model.

Prove that ranking by Borda scores may not coincide with the MLE ranking. That is, under the noise model described above, give a voting matrix that corresponds to complete rankings submitted by voters, where

ranking the alternatives by their Borda score (and breaking ties according to a fixed tie-breaking order) does not return the MLE ranking, for any $p \in (1/2, 1)$.

Hint: What have we proved about the Kemeny rule, and what is the relation between Borda and Kemeny?

1.2 Game Theory (20 points)

Let $x_1, \ldots, x_n$ be variables (for example, they could be vectors that represent probability distributions), and let $a_1, \ldots, a_n$ be non-negative real numbers such that $\sum_{i=1}^{n} a_i = 1$. The expression $\sum_{i=1}^{n} a_i x_i$ is called a *convex combination* of the $x_1, \ldots, x_n$. In this problem, we will investigate the effect of taking convex combinations of two solution concepts: Nash and correlated equilibria.

(a) (10 points) Let $p_1, \ldots, p_n$ be probability distributions representing n correlated equilibria in a 2-player game. Prove that any convex combination of $p_1, \ldots, p_n$ is also a correlated equilibrium.

(b) (10 points) Give an example of a 2-player game with at most 2 actions per player, where there exists a convex combination of two (mixed) Nash Equilibira that is not a mixed Nash Equilibrium.

1.3 Multi-Robot Systems (15 points)

The company Guber provides automatic pick-up and delivery services for goods. At any time, customers can send to a control center their requests for receiving selected goods at their homes. For instance, customer c can ask for a set of goods $G^c = \{g_1^c, g_2^c, \ldots, g_n^c\}$. The goods are available at known locations. Each good i has a capacity requirement q_i and a value v_i.

Periodically, customer requests are organized into a batch and dispatched from the control center to a team of n mobile robots that perform the pick-up and delivery tasks. Each robot k has a limited capacity Q_k. The entire set of requests G^c from a customer c is assigned to a single robot (no split of requests), respecting robot's capacity constraint.

Based on the knownledge of the capacity requirements of the goods, of robots' capacity limitations and mobility skills (e.g., wheeled vs. legged), and of the locations of goods and customers, Guber's control center computes the set R_k of all routes that can be feasibly assigned to a robot k for picking-up and deliver the goods for one or more customers. For each robot k, each feasible route r has known traveling cost c_r^k.

Given the number of requests in the batch and robots' capacity constraints, it might not be possible to satisfy all customer requests. Therefore, Guber wants to assign routes (i.e. pick-up and delivery subtasks) to the available robots with the goal of maximizing the difference between the overall value obtained from the delivered goods and the costs incurred for traveling. Every robot gets assigned at most one route.

(a) (11 points) Define an IP for Guber's task assignment problem above.

(b) (2 points) Can the resulting optimization problem be solved in polynomial time?

(c) (2 points) Based on Gerkey and Mataric taxonomy, how can the above problem can be classified?

1.4 Grad Problem: Monotonicity of Voting Rules [25 points]

In class, we defined Condorcet consistency, which, intuitively, was an axiom that one would like a voting rule to satisfy. There are many other natural axioms that one might like a voting rule to satisfy. One such example is *monotonicity*, which is defined as follows. Let $N = [n] = \{1, \ldots, n\}$ denote the set of voters, and let A denote the set of alternatives. Each voter gives a ranking over the alternatives, and we denote the ith voter's ranking by $\succ_i$, and the "vector" of all the rankings by $\vec{\succ}$. Such a vector of rankings is referred to as a *preference profile*. A voting rule is then a function f which, on input $\vec{\succ}$, returns a "winner" of the election $a \in A$, which we denote by $f(\vec{\succ}) = a$.

We say that a voting rule f is monotone if it satisfies the following condition. If $f(\vec{\succ}) = a$ and $\vec{\succ}'$ is any vector of rankings such that:

(i) if $a \succ_i x$, then $a \succ'_i x$,

(ii) for all $x, y \in A \setminus \{a\}$, $x \succ_i y$ if and only if $x \succ'_i y$,

then also $f(\vec{\succ}') = a$. Intuitively, the axiom states that if $\vec{\succ}'$ is obtained from $\vec{\succ}$ by pushing a even higher up the ranking and leaving everything else unchanged, then a should still win the election when the voters have rankings $\vec{\succ}'$. Unfortunately, as you will show in this exercise, not every voting rule satisfies this property.

(a) [5 points] Show that STV is not monotonic by giving a counter-example.

(b) [20 points] Show that Dodgson's rule is not monotonic by giving a counter-example. For a formal definition of Dodgson's voting rule, see Section 2 of this paper.

2 Programming

2.1 CNN (30 points)

2.1.1 Overview

A convolutional neural network (CNN) is a type of feed-forward neural network that is particularly well-suited to image-processing tasks. The architecture of a CNN is loosely inspired by the visual cortex. Compared to a more standard feed-forward network, CNNs are able to preserve certain spatial relationships in the image as they compute features, while at the same time maintaining invariance to trivial differences in images, such as translations, making them less prone to overfitting on image tasks.

2.1.2 Architecture

A typical CNN uses a number of different types of layers to process the input. The flow of processing an image and arriving at a classification decision is depicted in Figure 1.

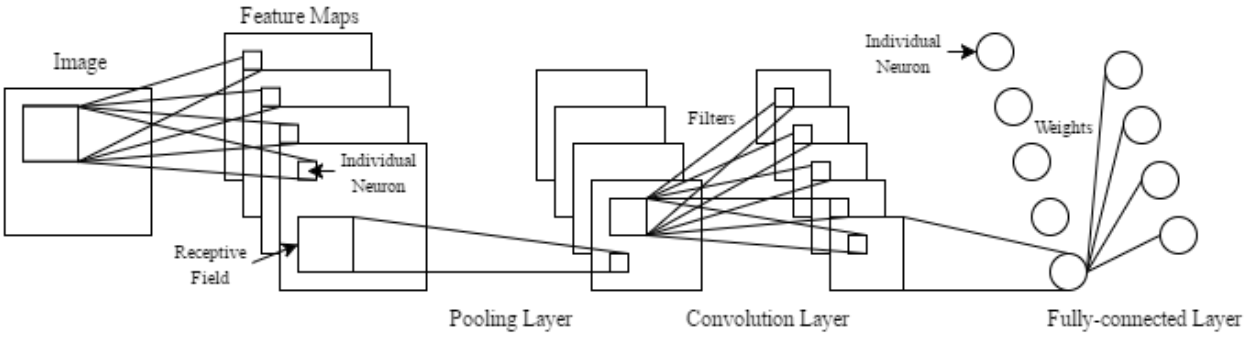

Figure 1: General Architecture of a CNN

In image processing, the input is an image, typically a matrix broken into three color channels (color channels not shown in the figure). The image is processed via convolution layers and pooling layers to produce *feature maps*, which are used as input to subsequent layers. Each feature map consists of many individual *neurons*, arranged in a 2D matrix, similarly to the input image. Between each layer, there is typically a *ReLU* (Rectified Linear Unit) to add non-linearity (not shown in the figure), making the model more expressive. Convolution and pooling layers reduce the dimensionality of the feature maps, and eventually the feature

maps are reduced to single neurons – this is typically called the fully-connected layer (there may be many fully-connected layers). The fully-connected layers are processed via weights and biases like a typical feed-forward network, however, we will see this can be generalized as a convolution layer with 1×1 filters and feature maps. Finally, a softmax is typically taken at the final layer of the network, which has one neuron per class – this is used to classify the image, as each neuron in the final layer maps to a class.

Convolution Layers As the name suggests, convolution layers are the main building block for a CNN. In a convolution layer, a set of learned *filters* are convolved with the feature maps to produce a new set of feature maps. In addition to the filters, a set of learned *biases* are added to the output feature maps to account for global properties of the training set. A convolution layer can be thought of as analogous to a layer in a regular feed-forward neural network where the feature maps are analogous to the neural units, and the filters are analogous to the weights, though the filters operate on the feature maps in a more complicated way.

We consider first how a single filter operates on a single feature map. Call this operation *conv*. The parameters to *conv* are an $a \times a$ feature map, M, a $b \times b$ filter, F, a *stride*, s, and a *padding*, (p_1, p_2, p_3, p_4). The padding essentially adds cells containing zeros to the outside of M, with p_1 extra cells on the top, p_2 on the bottom, p_3 on the left, and p_4 on the right. Call the padded feature map M'. The output of *conv* is a matrix, Y, of size $floor((p_1 + a + p_2 - b)/s) + 1 \times floor((p_3 + a + p_4 - b)/s) + 1$. Beginning with the top left corner of F at the top left of M', in steps of size s, we *convolve* F with M' to compute the values in Y. Formally, we set $Y_{i,j} = F \cdot M'_{is,js}$, where $M'_{x,y}$ is the region of M' covered by F when the top left corner of F is positioned at (x, y) in M' (shown in Figure 2). Here, "$\cdot$" is the dot product, i.e., the sum of the components of the matrices multiplied element-wise.

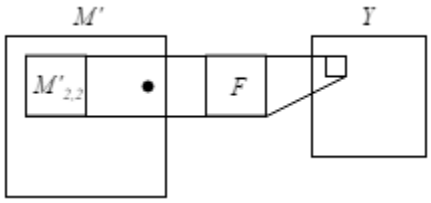

Figure 2: The "conv" Operation on One Feature Map

Now, to compute the next layer of feature maps, $\mathcal{M}^{out}$, from the input feature maps, $\mathcal{M}^{in}$, in a convolution layer, we do the following: suppose $\mathcal{M}^{in}$ contains n input feature maps, and we want m output feature maps. Then there are $n * m$ filters, $\mathcal{F}$, and m biases, $\mathcal{B}$, with the same dimension as the output feature maps. We let

$$\mathcal{M}_j^{out} = \mathcal{B}_| + \sum_{i=1}^{n} conv(\mathcal{F}_{i,j}, \mathcal{M}_i^{in}, s, (p_1, p_2, p_3, p_4)).$$

Pooling Layers Pooling is one type of non-linearity used by CNNs, and is also used to reduce dimensionality of the feature maps. Essentially, input feature map is partitioned into (possibly overlapping) receptive fields, each corresponding to a single neuron in the output feature map. In *max-pooling*, the corresponding neuron is given the value of the maximum value within its receptive field. Other values besides the max, such as the mean, can be used in pooling as well; however, max-pooling is the most common. Intuitively, pooling increases translation invariance, as the resulting map is not sensitive to the location of the max within the receptive field. Figure 4 depicts an example pooling layer operation.

Similarly to a convolution layer, a pooling layer takes a receptive field size (similar to the size of the filters in convolution), a stride, and a padding. In a pooling layer, there is a one-to-one correspondence between input feature maps, unlike in a convolutional layer. If Y is the output feature map corresponding to the padded

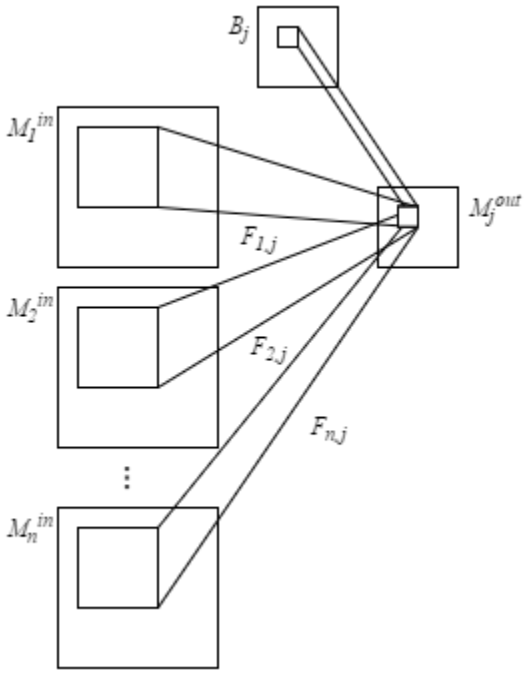

Figure 3: A Convolution Layer

input feature map, M', then $Y_{i,j} = max(M'_{is,js})$, where $M'_{x,y}$ is the region of M' covered by the receptive field when the top left corner of receptive field is positioned at (x, y) in M'.

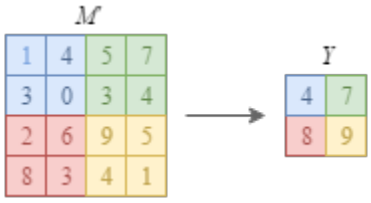

Figure 4: A Max-pooling layer

ReLU Layers Typically in-between layers, a ReLU (Rectified Linear Unit) is applied to add non-linearity. This allows the CNN to approximate more complex functions. In some cases a different non-linearity may be used, such as the sigmoid function; however, ReLU is the most common, as it leads to faster training. Formally, $ReLU(x) = max(0, x)$. This is applied element-wise to each feature map, replacing negative values with zeros.

Loss Layer Ultimately, the CNN needs to make some prediction about the input. In classification, this means assigning a class to the input. For image classification, the final layer in a CNN typically takes a softmax over the previous layer to output the class with the highest score. The softmax normalizes the weights of the final layer such that they can be interpreted as a probability distribution over the classes. The class with the highest probability is selected as the classification prediction.

2.1.3 Directions

To complete this assignment, you should have NumPy and SciPy installed. Instructions for this and numpy tutorials can be found at numpy.org. On Windows, it may be easiest to download Anaconda or some similar software that comes with the numpy and scipy packages.

The code in `problems.py` provides a framework for loading and running pre-trained CNN models. A CNN model can be created by calling

`cnn = Cnn().load('imagenet-vgg-f.mat')`,

which loads the model from the given `.mat` file. To run the model, use `cnn.classify(imageFile)`. This loads the given image file, scales it to match the input dimensions of the model, and does some other normalizations. The image is then processed by each layer in the network sequentially. In order for this to work, you will need to complete the implementation for a number of `CnnLayer` classes, namely `ConvLayer`, `PoolLayer`, `ReluLayer`, and `SoftmaxLayer`. For each of these classes, the `processLayer` method needs to be implemented. `processLayer` takes in one argument, `prevLayer` which is the incoming set of feature maps. `prevLayer` is a $n \times n \times d$ NumPy array, where n is the size of the feature maps, and d is the number of feature maps in the layer. The `processLayer` method should return the next layer as a NumPy array.

ConvLayer (10 points) Implement the `processLayer` method for the `ConvLayer` class. This takes the input feature maps and outputs the result of the convolution operation described above. `filters` is an $n \times n \times d_1 \times d_2$ numpy array where n is the filter size, $d_1$is the number of incoming feature maps, and d_2 is the number of output feature maps.`biases` is a size $d2$ numpy array with one bias value to be added to each of the output feature maps. `stride` is an integer indicating the stride size in both the x and y direction. `pad` is a 4×1 numpy array with the paddings `[top, bottom, left, right]`.

PoolLayer (10 points) Implement the `processLayer` method for the `PoolLayer` class. This takes the input feature maps and outputs the result of the max-pooling operation described above. `size` is a 2×1 numpy array with the dimensions of the receptive field of the pooling operation. `stride` is an integer indicating the stride size in both the x and y direction. `pad` is a 4×1 numpy array with the paddings `[top, bottom, left, right]`.

ReluLayer (3 points) Implement the `processLayer` method for the `ReluLayer` class. This outputs the result of applying the ReLU operation element-wise to each feature map.

SoftmaxLayer (3 points) Implement the `processLayer` method for the `SoftmaxLayer` class. This outputs the result of applying the softmax function to the input feature map. The input feature map will be a $1 \times 1 \times d$ numpy array at this point, where d is the number of classes, so it can be thought of as a d-dimensional vector, z. The softmax function, $\sigma(z)$ is then given by

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{d} e^{z_k}}, \forall j \in \{1, ..., d\}$$

NormalizationLayer (4 points) Implement the `processLayer` method for the `NormalizationLayer` class. This outputs the feature-wise sliding window normalization, also called the"Local Response Normalization" of the input feature map. This takes arguments n, κ, α, and β. Formally, for an $a \times a \times d$ matrix of d $a \times a$ input maps, X, the output, Y, is given by

$$Y_{i,j,k} = \frac{X_{i,j,k}}{(\kappa + \alpha \sum_{q \in Q(k)} X_{i,j,q}^2)^\beta}$$

where $Q(k)$ is the window, given by the range of integers, $[max(0, k - floor((N-1)/2)), min(d, k + ceil((N-1)/2))]$. However for this assignment, assume that $\beta = 1$ and please ignore the β parameter in the code. In other words you should implement:

$$Y_{i,j,k} = \frac{X_{i,j,k}}{\kappa + \alpha \sum_{q \in Q(k)} X_{i,j,q}^2}$$

2.2 Stackelberg Strategies (15 points)

In a 2-player normal form game, a Stackelberg strategy is where one of the players is a leader and the other is a follower. In contrast to the default situation where both players pick their respective strategies at the same time, a Stackelberg strategy is when the leader, which is identified as player 1, first commits to a (mixed) strategy which the follower, player 2, knows. Then player 2 commits to his own strategy using his knowledge of player 1's strategy.

An optimal Stackelberg strategy would be a Stackelberg strategy where player 1's expected utility is maximized. The optimal Stackelberg strategy can be computed in polynomial time by solving multiple LPs. See Slide 14 of Lecture 23 for a description of the algorithm.

Given a 2-player normal form game, you will implement the function `stackelberg(u1, u2)` in `sbproblems.py` which will return the optimal Stackelberg strategy for the given game. You should use the `cvxopt` library again for solving LPs. If there is a tie (i.e. the expected utility is off by an absolute error of 1e-5), you should return the optimal strategy induced by the lowest indexed pure strategy of player 2.