P2 - Supervised Learning with Tile Coding

Due Tuesday, 22nd of November by 2 PM in Gradescope

Policy: This project can be done in teams of up to two students (all students will be responsible for completely understanding all parts of the team solution).

Introduction

The objective of this project is to implement 2D tile coding and use it to construct the binary-feature representation for a simple case of supervised learning of a real-valued function.

As in all supervised learning, the training set consists of a set of examples, where each example is a pair, `input` and `target`, and the objective is to learn a function `f` such that `f(input)` is close to `target`. In this project, each `input` consists of two numbers or coordinates, let's call them `in1` and `in2`, each of which is between 0 and 6 (that is, `input` $\in [0,6)^2$), and `target` is a single (scalar) real number. Thus, a single example might be written `(in1, in2, target)`. Here is a training set of four examples that we will use in the project:

Example#	in1	in2	target
1	0.1	0.1	3
2	4	2	–1.0
3	5.99	5.99	2
4	4	2.1	–1.0

Receiving such a training set, your job will be to write the Python function `f(in1,in2)` which returns a guess of the target at the corresponding input. Your guesses should be close to the `targets` on the training set and, more importantly, close to the `targets` on any new examples that you might see in the future (generalization). You will "receive the training set" by receiving multiple calls to the Python function `learn(in1,in2,target)`, which you will also write. We will discuss these further in Part 2 below; first we focus on the tile-coding part for converting points in `input` space to large binary feature vectors (or rather small arrays containing the indices of the few 1 components of the binary feature vectors).

Part 1: Tile Coding

Implement your supervised-learning algorithm using tile coding over `in1` and `in2`. The first step is to write the function `tilecode(in1,in2,tileIndices)` which takes in `in1`, `in2` and an array (`tileIndices`) and fills the array with tile indices, one index per tiling. Use eight standard grid tilings (numTilings=8) of the `input` space, where each tile extends 0.6 (one tenth of the [0,6) range) in each of the `in1` and `in2` directions. You might think of each tiling as forming a 10x10 grid, but actually make it an 11x11 grid so that it extends beyond the `input` space by one tile width and height. We do this because each tiling after the first will be offset by –⅛ of a tile width and height in the `in1` and `in2` directions. Thus, the first tiling will look like the left side of figure 1 and the second tiling will look like the right side of the figure. Subsequent tilings will each be offset by a further –⅛ of a tile in each direction (more generally, by –1/numTilings, of course). This is not the best way to offset the tilings, and you will see artifacts in your results because of it, but we do it anyway because it is the simplest. The next better method would be to offset each tiling by a random amount.

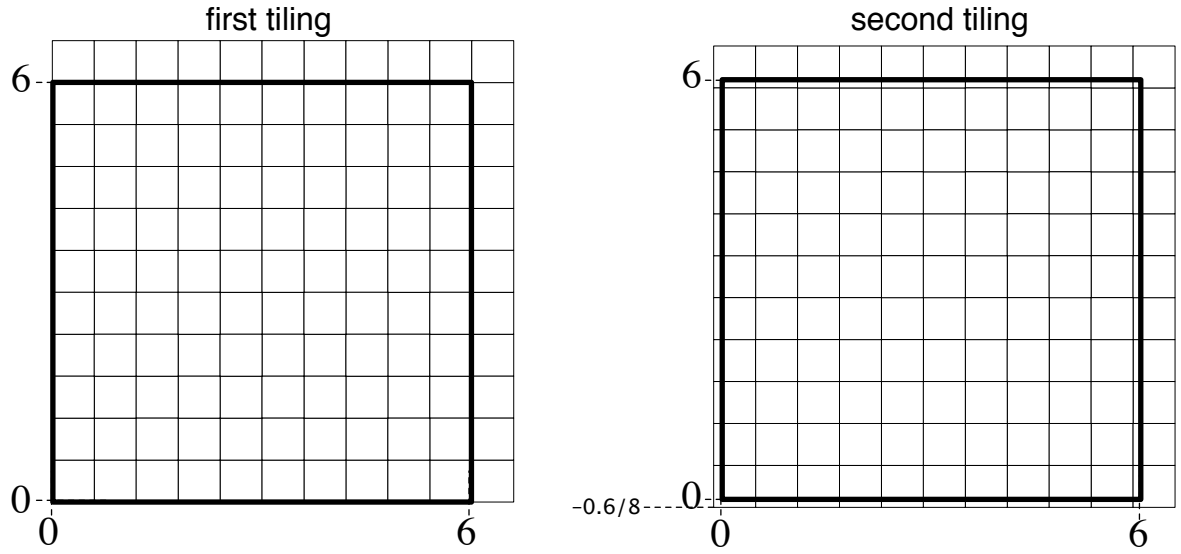

Figure 1. The relationship between the `input` space (in bold) and the first and second 11x11 tilings.

Every tile has a different tile number (index). Assuming that you number the tiles in the natural way, the tiles in the first tiling will run from 0 to 120, and the tiles in the second tiling will run from 121 to 241 (why?). A given `input` point will be in exactly one tile in each tiling. For example, the point from the first example in the training set above, in1=0.1 and in2=0.1, or 0.1,0.1, will be in the first tile of the first seven tilings, that is, in tiles 0, 121, 242, 363, 484, 605, 726 (why?). In the eighth tiling this point will be in the 13th tile (why?), which is tile 859 (why?). If you call `tilecode(0.1,0.1,tileIndices)`, then afterwards `tileIndices` will contain exactly these eight tile indices. The largest possible tile index is 967 (why?).

What to turn in for Part 1. Download the template or shell file Tilecoder.py (from the dropbox) and edit it to contain your function `tilecode`. Test your code by running that file and printing the results. Turn in your modified Tilecoder.py file. The code just calls your `tilecode` function on the four example input points given above and prints out the tile indices for each case. If you number the tiles as suggested, then the first example should produce the eight integer tile indices given above. Another check on your code is that none of the indices should be negative or greater than 967. Finally, the second and fourth examples should produce very similar sets of indices (they should have many tiles in common) (why?). We will use this printout to check that your `tilecode` function is working properly and to help in giving partial credit. Finally, provide a list of very brief answers to the six "why" questions in the project specification above (as the first page of a file P2.pdf, put it under a heading "Answers to questions about part 1"). After you have completed part 1, comment out the four print statements at the end of the file in preparation for using your tilecoder in part 2.

Part 2: Supervised Learning by Gradient Descent (ADALINE)

In this part you will use your tile coder in conjunction with linear function approximation to learn an approximate function over the entire `input` space. The learning algorithm we use is stochastic gradient descent in the weights of a linear function approximator; it is known as the Least-Mean-Square (LMS) algorithm in the literature. As a linear approximator, the function it produces is a linear combination of a vector of learned weights $\boldsymbol{\theta} = (\theta_1, \theta_2, ..., \theta_n)$, and a vector of features $\boldsymbol{\phi}(i) = (\phi_1(i), \phi_2(i),..., \phi_n(i))$ for a given `input` i:

$$f(i) = \theta_1\phi_1(i) + \theta_2\phi_2(i) + \cdots + \theta_n\phi_n(i).$$

As a gradient-descent learning algorithm, the weights are updated, given an input i, a corresponding feature vector $\boldsymbol{\phi}(i)$ and target value $target(i)$, and an estimated function value $f(i)$, by

$$\theta_j \leftarrow \theta_j + \alpha\left[target(i) - f(i)\right]\phi_j(i), \qquad \forall j = 1, \ldots, n.$$

Your task in this part of the project is to implement the first of these equations in the python function `f(in1,in2)` and the second in the Python function `learn(in1,in2,target)`, using feature vectors constructed by the tilecoder you wrote in Part 1, and assuming the general setup outlined in the introduction.

First, note that n in the above equations is the total number of tiles in your tile coder, or 968. Because your arrays are probably indexed from 0, your loops will probably go from 0 to 967. Second, note that, because you are doing tile coding, you don't have the feature vector $\boldsymbol{\phi}(i)$ in an explicit form. Instead you have a list of indices j where $\phi_j(i)$ is 1, with all others assumed 0. If you think about it, this really simplifies your job of

implementing both equations. Of course you will need to keep the weight vector $\mathbf{w}$ in explicit form. That is, you really will need a vector of 968 floating point numbers; this is your memory; initialize all of its elements to zero. The step-size parameter α should be set to 0.1 divided by the number of tilings.

Completing Part 2 consists of the following steps:

1. Download the file SuperLearn.py (in the dropbox) which contains test and template code.
2. Edit the functions `f` and `learn` to implement the desired functions, calling your `Tilecoder.tilecode` function when needed.
3. Run SuperLearn.py. The last line of the file calls the `test1` function, which calls your learning algorithm on the four example points and prints out results. As a result of learning, your approximate function value should move 10% of the way from its original value towards the target value. The before value of the fourth point should be nonzero (why?). Make sure this is working correctly before going on to the next step.
4. Change the last line of SuperLearn.py to call test2 instead of test1 and execute it again. The test2 function constructs 20 examples using the target function: $target = \sin(in1 - 3)\cos(in2) + N(0, 0.1)$, where N(0,0.1) is a normally distributed random number with mean 0 and standard deviation 0.1. The examples are sent to learn, and then the resultant function f (after 20 examples) is printed out to a file (f20) in a form suitable for plotting by excel or similar. A Monte Carlo estimate of the Mean-Squared-Error in the function is also printed to standard output. The program then continues for 10,000 more examples, printing out the MSE after each 1000, and then finally prints out the function to a file (f10000) for plotting again. You should see the MSE coming down smoothly from about 0.25 to almost 0.01 and staying there (why does it not decrease further towards zero?).
5. Make 3D plots of the function learned after 20 and 10,000 examples, using the data printed in step 4. One way to do this is to use the program plot.py provided in the dropbox, as in "python plot.py f20". After 10,000 examples, the learned function should look very much like the target function minus the noise term. You can test that by typing the target function minus the noise into one of the online 3D plotting programs, such as that at http://www.livephysics.com/ptools/online-3d-function-grapher.php.
6. After only 20 examples, your learned function will not yet look like the target function. Explain in a paragraph why it looks the way it does. If your learned function involves many peaks and valleys, then be sure to explain both their number, their height, and their width. Suppose that, instead of tiling the input space into an 11x11 grid of squares, you had divided into an 11x21 grid of *rectangles*, with the `in1` dimension being divided twice as finely as the `in2` dimension. Explain how you would expect the function learned after 20 examples to change if this alternative tiling were used.

What to turn in for Part 2. Turn in 1) your edited version of SuperLearn.py, 4) Append your explanations from step 6 to P2.pdf (put it on a new page and under the heading "Explanation of part 2", 3) your brief answers to the "why" questions in steps 3 and 4 (put it on a new page and under the heading "Answers to questions about part 2"), 4) Append to P2.pdf the printout from step 3 together with the MSEs printed from step 4 (put them under a new page and under the heading "MSEs"), and 5) Append the two 3D plots from step 5 to P2.pdf (in a new page each and under headings "F20" and "F10000").

How and what to hand in:

Submit your assignment on Gradescope. Make sure your code runs in a **Python 3** interpreter. **Comment all the lines where you print something in the submitted files (your scripts should not print anything to the screen)**. You should submit your **Tilecoder.py** (don't forget to comment out the four print statements) for part 1, and your **SuperLearn.py** for part 2 in "**Programming Assignment 2 Code**" in Gradescope. **Use the updated SuperLearn.py template**. You also need to submit **collaborator.txt, even if you don't have a collaborator,** along with the two Python 3 files. Be sure to **include all mentioned files even if they are empty**. **Do not call functions that will overwrite your submission files**, e.g., f20, f10000, etc. in your final submission. **File names are case-sensitive and should be named exactly as mentioned in this document**. Finally, submit your **P2.pdf** in "**Programming Assignment 2 PDFs (Marked)**" in Gradescope. **Your P2.pdf should be in the format and ordering specified in this document. Every part has to be put in a new page and under headings mentioned with the order: 1) Answers to questions about part 1, 2) Explanation of part 2, 3) Answers to questions about part 2, 4) MSEs, 5) F2, and 6) F10000.**