CSCI 4360/6360 Data Science II
Department of Computer Science
University of Georgia

Assignment 3: All-Natural Data Science

DUE: Thursday, September 28 by 11:59:59pm

Out September 18, 2017

Questions

This homework assignment wraps up our module on nature-inspired computing methods, and incorporates our topics on computer vision as well.

0.1 Particle Swarm Optimization [**25pts**]

Particle Swarm Optimization (PSO) is yet another nature-inspired search algorithm that attempts to strike a balance between *exploration* (conducting fast but low-resolution searches of large parameter spaces) with *exploitation* (refining promising but small areas of the total search space).

Here is a Matlab plot of PSO in action: notice how the majority of agents (dots) very quickly gather in the bottom left corner (exploitation) representing the global minimum, but there are nonetheless a few dots that appear elsewhere on the energy landscape (exploration).

Rather than devote a third homework assignment's coding section to yet another document classification scheme, we'll explore PSO from a more theoretical viewpoint.
PSO was introduced in 1995, and was inspired by the movement of groups of animals: insects, birds, and fish in particular. Virtual particles "swarm" the search space using a directed but stochastic algorithm designed to modulate efforts to find the global extremum (exploitation) while avoiding getting stuck in local extrema (exploration). It is relatively straightforward to implement and easy to parallelize; however, it is slow

to converge to global optima, and ultimately cannot guarantee convergence to global optima.

Formally: N particles move around the search space $\mathcal{R}^n$ according to a few very simple rules, which are predicated on:

- each particle's individual best position so far, and
- the overall swarm's best position so far

Each particle i has a position $\vec{x}_i$, a velocity $\vec{v}_i$, and an optimal position so far $\vec{p}_i$, where $\vec{x}_i, \vec{v}_i, \vec{p}_i \in \mathcal{R}^n$.

Globally, there is an optimal swarm-level position $\vec{g} \in \mathcal{R}^n$ (the supremum of all $\vec{p}_i$), cognitive and social parameters c_1 and c_2, and an inertia factor ω.

The update rule for velocity $\vec{v}_i$ at time $t+1$ is as follows:

$$\vec{v}_i(t+1) = \omega\vec{v}_i(t) + c_1 r_1 \left[\vec{p}_i(t) - \vec{x}_i(t)\right] + c_2 r_2 \left[\vec{g}(t) - \vec{x}_i(t)\right]$$

where $r_1, r_2 \sim U(0,1)^n$.

[10pts] Explain the effects of the cognitive (c_1) and social (c_2) parameters on the particle's velocity. What happens when one or both is small (i.e. close to 0)? What happens when one or both is large? What effects do the random numbers r_1 and r_2 have? Relate the effects of these parameters to their "nature"-based inspiration, if you can.

ANSWER:

The cognitive (c_1) and social (c_2) parameters shift the particle's velocity in the direction of the of particle's percieved optimum and the swarm's perceived optimum respectively. When they are large, there can be large changes in the solution and when they are small, the particle's original velocity is maintained. In this case it will continue to move according to this inherent velocity until it naturally converges as ω decreases to zero. If they are large they should converge at a midpoint between a particle's percieved optimum and the global optimum.

[5pts] The inertia parameter ω in this formulation is typically started at 1 and decreased slowly on each iteration of the optimization procedure. Why?

ANSWER:

As the swarm moves closer and closer to an optimal solution, the particle should move less based on inertia and more based on cognitive and social cues

that direct it towards the optimal solution. These cues should also decrease as the difference between optimal and local position decreases.

[**5pts**] The update rule for position $\vec{x}_i$ at time $t+1$ is as follows:

$$\vec{x}_i(t+1) = \vec{x}_i(t) + \vec{v}_i(t+1)$$

Given an objective function f that can be evaluated using a position vector $\vec{x}_i(t)$, provide Python-like update statements for the best particle-specific estimate $\vec{p}_i(t+1)$, and the best global, swarm-level estimate $\vec{g}(t+1)$. **Note**: for the sake of consistency, let's assume you're searching for the global *minimum* of f.

Hint: Remember that particle-specific estimates $\vec{p}_i(t)$ are also position vectors.

ANSWER:

```
def new_x(x_t, p, g):
  p = argmin(f, p) # find p that minimizes f
  g = min(p)
```

[**5pts**] Give a *concrete* example of how the PSO formulation described here could be improved (better global estimate in the same amount of time, faster convergence, tighter global convergence bounds, etc); you don't have to provide a specific implementation, but it should be clear how it would work ("more power", therefore, is not a concrete example). Such formulations are easy to find online; I implore you to resist the urge to search! Please keep it brief; I'll stop reading after 2-3 lines.

ANSWER:

One could remove the random number wieghts for the cognitive and social cues and have them also decay as a function of the distance from the local and optimal solution respectively.

0.2 Linear Dynamical Systems [**35pts**]

Linear dynamical systems (LDS) are multidimensional time series models with two components: an appearance component, and state component, that are used to model and identify dynamic textures. Dynamic textures are video sequences that display spatial regularities over time, regularities we want to capture while simultaneously retaining their temporal coherence.

The appearance component is a straightforward application of dimensionality reduction, projecting the original temporal state into a low-dimensional "state space":

$$\vec{y}_t = C\vec{x}_t + \vec{u}_t$$

where $\vec{y}_t$ is the current frame, $\vec{x}_t$ is the corresponding "state," $\vec{u}_t$ is white noise, and C is the matrix that maps the appearance space to the state space (and vice versa), sometimes referred to as the *output matrix*: it defines the appearance of the model.

The state component is nothing more than an autoregressive model, a linear combination of Markov assumptions that model the movement of the system in the low-dimensional state space:

$$\vec{x}_t = A\vec{x}_{t-1} + W\vec{v}_t$$

where $\vec{x}_t$ and $\vec{x}_{t-1}$ are the positions of the model in the state space at times t and $t-1$ respectively, $W\vec{v}_t$ is the *driving noise* at time t, and A is the transition between states that defines how the model moves.

[5pts] In the following questions, we're going to use only the state component of the LDS (i.e., we'll only use the second equation to model motion). How could we formalize "ignoring" the appearance component? What values could we use in the appearance component so that the original data $\vec{y}_t$ is also our state space data $\vec{x}_t$?

ANSWER:

We could flatten each frame to one row where we pass each value to the state equation instead of just the q best.

[10pts] To simplify, let's ignore the appearance component and focus on a toy example in two dimensions.

Suppose each $\vec{x}_t$ is an (x, y) pair from the plot. Set up the equations to solve for A (Note: your solution should generalize to n 2D points. Also, you can assume there is no noise term (i.e. $W\vec{v}_t = 0$)).

Hint: If there is no noise term, then each $\vec{x}_t$ can be written as $A\vec{x}_{t-1}$ for all t. Write a few of these out, then organize them into systems of equations.

ANSWER:

$x_t = Ax_{t-1}$
$x_t = A^2x_{t-2}$
...
$x_t = A^tx_1$

[10pts] An interesting property of any model is its behavior in the limit. Those familiar with certain dimensionality reduction strategies will notice the simplified autoregressive model from the previous step looks an awful lot like a power iteration for finding ap-

proximate eigenvalues and eigenvectors of a matrix: if M is your matrix of interest, you can iteratively update a vector $\vec{v}$ such that $\vec{v}_{n+1} = M\vec{v}_n$, and each additional iteration will bring $\vec{v}$ closer to the true leading eigenvector of M.

Assuming M is invertible, we have the full eigen-decomposition of $M = U\Sigma U^T$, where U is the matrix of eigenvectors $[\vec{u}_1, ..., \vec{u}_n]$, and Σ is the diagonal matrix of eigenvalues $\lambda_1, ..., \lambda_n$ sorted in descending order $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n$.

Write out the equation for $\vec{v}_{n+2}$ using **only** M and $\vec{v}_t$. Do the same for $\vec{v}_{t+3}$. Describe how this generalizes for n steps. What is happening in terms of M?

ANSWER:

$\vec{v}_{n+2} = M^{n+2-t} * \vec{v}_t$
$\vec{v}_{t+3} = M^3 * \vec{v}_t$

This generalizes for n steps by multiplying by M as many times as the difference in steps between the nth and current step.

[10pts] Now, rewrite those same equations, but instead of M, use its eigen-decomposition form. What happen as the number of iterations $n \to \infty$? What does this mean if there are eigenvalues $\lambda_i < 1$? What if $\lambda_i = 1$? What if $\lambda_i > 1$? What is therefore happening to the corresponding eigenvectors $\vec{u}_i$ of λ_i? (Note: $n \to \infty$ is known as the *steady state*)

Hint: The eigenvector matrix U has the property $U^TU = UU^T = I$, where I is the identity matrix.

ANSWER:

$\vec{v}_{n+2} = U\Sigma U^{T^{n+2-t}}\vec{v}_t$
$\vec{v}_{t+3} = U\Sigma U^{T^3} * \vec{v}$

As the number of iterations approaches infinity, Σ begins to dominate the function. U and U^T will form an identity matrix. So you'll end up with a sum of the values being multiplied by the current vector to produce the next vector. Rate of change of the vector will be larger or smaller with changes in λ.

0.3 Coding [**40pts**]

In this question, you'll implement a basic LDS to model some example dynamic textures.

You'll be allowed the following external Python functions: `scipy.linalg.svd` for computing the appearance model (output matrix) C, and `scipy.linalg.pinv` for computing the pseudo-inverse of the state-space observations for deriving the transition matrices.

No other external packages or SciPy functions will be allowed (besides NumPy of course).

You'll also be provided the boilerplate to read in the necessary command-line parameters:

1. `-f`: a file path to a NumPy array file (the dynamic texture)
2. `-q`: an integer number of dimensions for the state-space
3. `-o`: a file path to an output file, where the prediction will be written

Your code will read in the NumPy array representing a dynamic texture video; it will have dimensions `frames` × `height` × `width`. We'll call this M, and say it has shape $f \times h \times w$. From there, you'll need to derive the parameters of the LDS: the appearance model C and the state space data X (both can be derived by performing a singular value decomposition on Y, which is formed by stacking all the pixel trajectories of M as rows of Y). Once you've learned C and X, you can learn the transition matrix A using the pseudo-inverse:

$$A = X_2^f (X_1^{f-1})^\diamond$$

where X_1^{f-1} is a matrix of $\vec{x}_1$ through $\vec{x}_{f-1}$ stacked as rows, X_2^f is a matrix of $\vec{x}_2$ through $\vec{x}_f$ stacked as rows, and $D^\diamond = D^T(DD^T)^{-1}$ is the pseudo-inverse of D.

Once you've learned C, X, and A, use these parameters to **simulate one time step**, generating $\vec{x}_{f+1}$. Use C to project this simulated point into the appearance space, generating $\vec{y}_{f+1}$. Reshape it to be the same size as the original input sequence (i.e., $h \times w$), and then **write the array to the output file**. You can use the `numpy.save` function for this. **Any other program output will be ignored.**

[**BONUS: 10pts**] Re-formulate your LDS implementation so that it also learns $W\vec{v}_t$, the driving noise parameter in the state space model. Recall that this first relies on the one-step prediction error:

$$\vec{p}_t = \vec{x}_{t+1} - A\vec{x}_t$$

which is used to compute the covariance matrix of the driving noise:

$$Q = \frac{1}{f-1} \sum_1^{f-1} \vec{p}_t \vec{p}_t^T$$

Perform a singular value decomposition of $Q = U\Sigma V^T$, set $W = U\Sigma^{1/2}$, and $\vec{v}_t \sim \mathcal{N}(0, I)$.

Implement the same 1-step prediction as before, this time with the noise term, so the state-space prediction is done as $\vec{x}_{t+1} = A\vec{x}_t + W\vec{v}_t$. Does your accuracy improve? Why

do you think this is the case?

[BONUS: 30pts] Re-formulate your LDS so that your state space model is a second-order autoregressive process. That is, your model should now be:

$$\vec{x}_{t+1} = A_1\vec{x}_t + A_2\vec{x}_{t-1}$$

Learning the transition matrices A_1 and A_2 is conceptually the same as before, but implementation-wise requires considerably more ingenuity to implement, so if you need help, please come to office hours!

Implement the same 1-step prediction as before, this time with the second-order model. Does your accuracy improve? Why do you think this is the case?

(it doesn't matter if you include the driving noise from the first bonus question here or not; as in, you don't have to implement the first bonus question to also get this one)

Administration

0.1 Submitting

All submissions will go to **AutoLab**. You can access AutoLab at:

- https://autolab.cs.uga.edu

You can submit deliverables to the **Assignment 3** assessment that is open. When you do, you'll submit two files:

1. **`assignment3.py`**: the Python script that implements your algorithms, and
2. **`assignment3.pdf`**: the PDF write-up with any questions that were asked

These should be packaged together in a tarball; the archive can be named whatever you want when you upload it to AutoLab, but the files in the archive should be named **exactly** what is above. Deviating from this convention could result in the autograder failing!

To create the tarball archive to submit, run the following command (on a *nix machine):

```
> tar cvf assignment3.tar assignment3.py assignment3.pdf
```

This will create a new file, `assignment3.tar`, which is basically a zip file containing your Python script and PDF write-up. Upload the archive to AutoLab. There's no penalty for submitting as many times as you need to, but keep in mind that swamping the server at the last minute may result in your submission being missed; AutoLab is

programmed to close submissions *promptly* at 11:59pm on September 28, so give yourself plenty of time! A late submission because the server got hammered at the deadline will *not* be acceptable (there is a *small* grace period to account for unusually high load at deadline, but I strongly recommend you avoid the problem altogether and start early).

Also, to save time while you're working on the coding portion, you are welcome to create a tarball archive of just the Python script and upload that to AutoLab. Once you get the autograder score you're looking for, you can then include the PDF in the folder, tarball everything, and upload it. AutoLab stores the entire submission history of every student on every assignment, so your autograder (code) score will be maintained and I can just use your most recent submission to get the PDF.

0.2 Reminders

- If you run into problems, ping the `#questions` room of the Slack chat. If you still run into problems, ask me. But please please please, **do NOT** ask Google to give you the code you seek! I will be on the lookout for this (and already know some of the most popular venues that might have solutions or partial solutions to the questions here).
- Prefabricated solutions (e.g. `scikit-learn`, OpenCV) are NOT allowed! You have to do the coding yourself!
- If you collaborate with anyone, just mention their names in a code comment and/or at the top of your homework writeup.