Structured Prediction & Algorithm Decision Flow Chart

Lecturer: Drew Bagnell *Scribe: Mike Taylor*

1. Overview

Structured prediction applies to a class of problems that attempt to predict a number of items that can reasonably be expected to be related.

Some Examples:

- Labeling / classifiying a lidar point cloud. Labels of closely spaced points are often correlated.
- Parsing a sentence.
- Recognition of image contents. For instance, the presence of a road would indicate a higher likelihood of finding a car.
- Estimating terrain surface

1.1 Solution Strategy #1: Optimization

$$\text{Solve:} \quad Y_{pred} = \text{argmin}_y / \text{max}_y \, L(Y, \text{Features})$$

Estimating weight vectors on features can happen in two ways. The likelihood of the estimate can be maximized or the number of errors can be minimized.

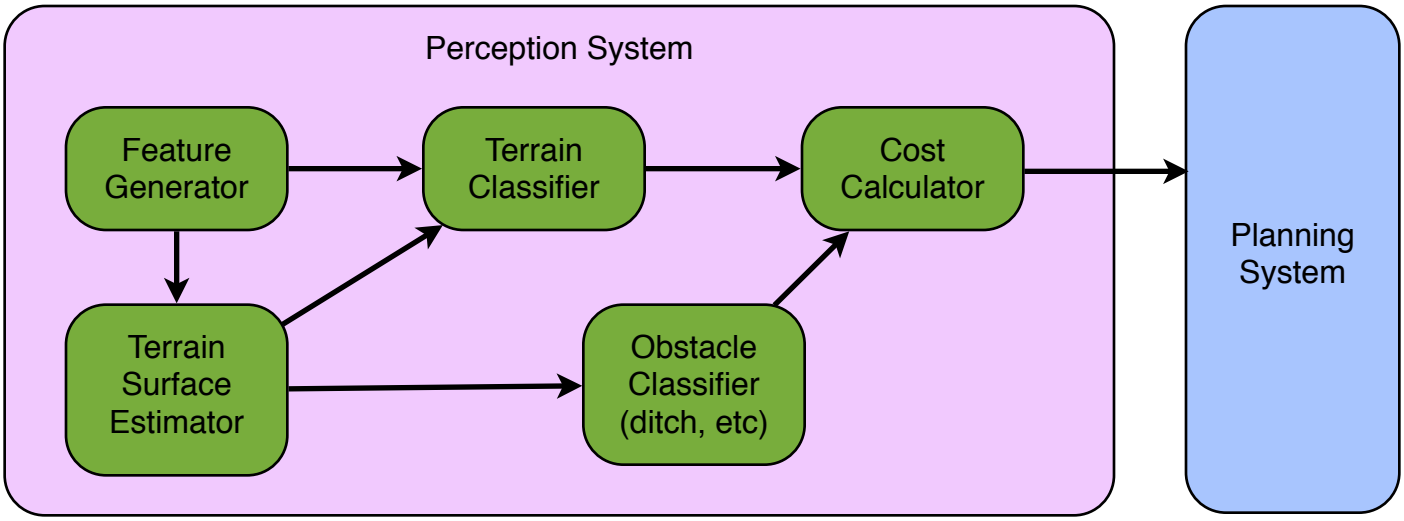
Some examples of appropriate problems:

- Predicting most likely terrain surface
- Analyzing point clouds: similar to MRFs where differences between neighbors are penalized.

1.2 Solution Strategy #2: Sequences of Predictions

Method: Run through data points making a series of predictions where each successive prediction is based on predictions on previous data points. For example, a point cloud is analyzed in a manner where the prediction on the second point is influenced by the prediction on the first point. The third point's prediction would then be based on the prediction on the second point and so on.

This is particularly applicable to multi-module systems such as the perception system shown below:



2. Training for Structured Prediction Problems

Structured prediction systems can be trained in a local or global manner or even a combination of the two. Local approaches attempt to train modules one at a time based on a set of module inputs and desired outputs. Once all modules are trained, the system is considered complete. Global approaches

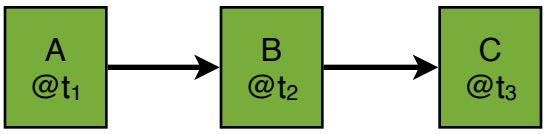
2.1 Local Training Methods

2.1.1 ***Poor Choice:*** Training in Parallel

One option is to train the various modules of a system in parallel. Here, the input for each module is the *desired* output of the modules that feed it. For example, training the terrain classifier module in the above system with the desired output of the feature generator and terrain surface estimator modules.

The main problem with training in parallel is that errors cascade in this system as modules are never trained on errors developed by upstream modules. If we assume that a module cannot recover (predict correctly) when an error occurs in an upstream module, and that each module can be trained to commit only ε errors, the entire system will commit εN^2 errors.

This issue can be seen in training driving modules. In this case, we can think of the modules as connected in time:



Here the input is an camera image of the road ahead and the output is a driving action, such as turn left, drive straight, or turn right. Unfortunately, these systems perform poorly because it will rarely see how to correct for a poor driving choice. Therefore, it will not be able to recover after it makes a mistake.

A second example can be seen in MRF-like applications where decisions are made based on feature data as well as the decisions of neighbors. With improper construction, the system will learn to weight neighbors far too high and the feature data too low. This is because the training provides only the correct input from these neighbors, thus they are more reliable in training than in practice.

In general, training in parallel should not be used.

2.1.2 ***Better Choice:*** Training in Series

a.k.a. Stacking or 'The Forward Algorithm'

Method: train A, then train B | A, then train C | A, B

This method trains a module with the output of the upstream modules that feeds it. This training data therefore includes mistakes made by the upstream modules. Using the previous analysis, this method will produce a system capable of achieving only εN errors, as opposed to the εN^2 errors seen with training in parallel.

Issues:

1. A module will require retraining if an upstream module is changed.
2. Only the final output module is trained based on the performance of the overall system.
3. It must be remembered that not all errors are created equal. For instance, a perception system that places an obstacle in an open field may be catastrophic, while placing an obstacle near a pile or rocks may be quite reasonable.

2.2 Global Training Methods

If all modules are differentiable, backwards propagation on the quality of the final output of the overall system is recommended. If not, search over the policy space using dynamic programming may yield results. The values used in this search, as applied to potential outputs of a module, are based on the 'goodness' of the final output of the system if that output were seen.

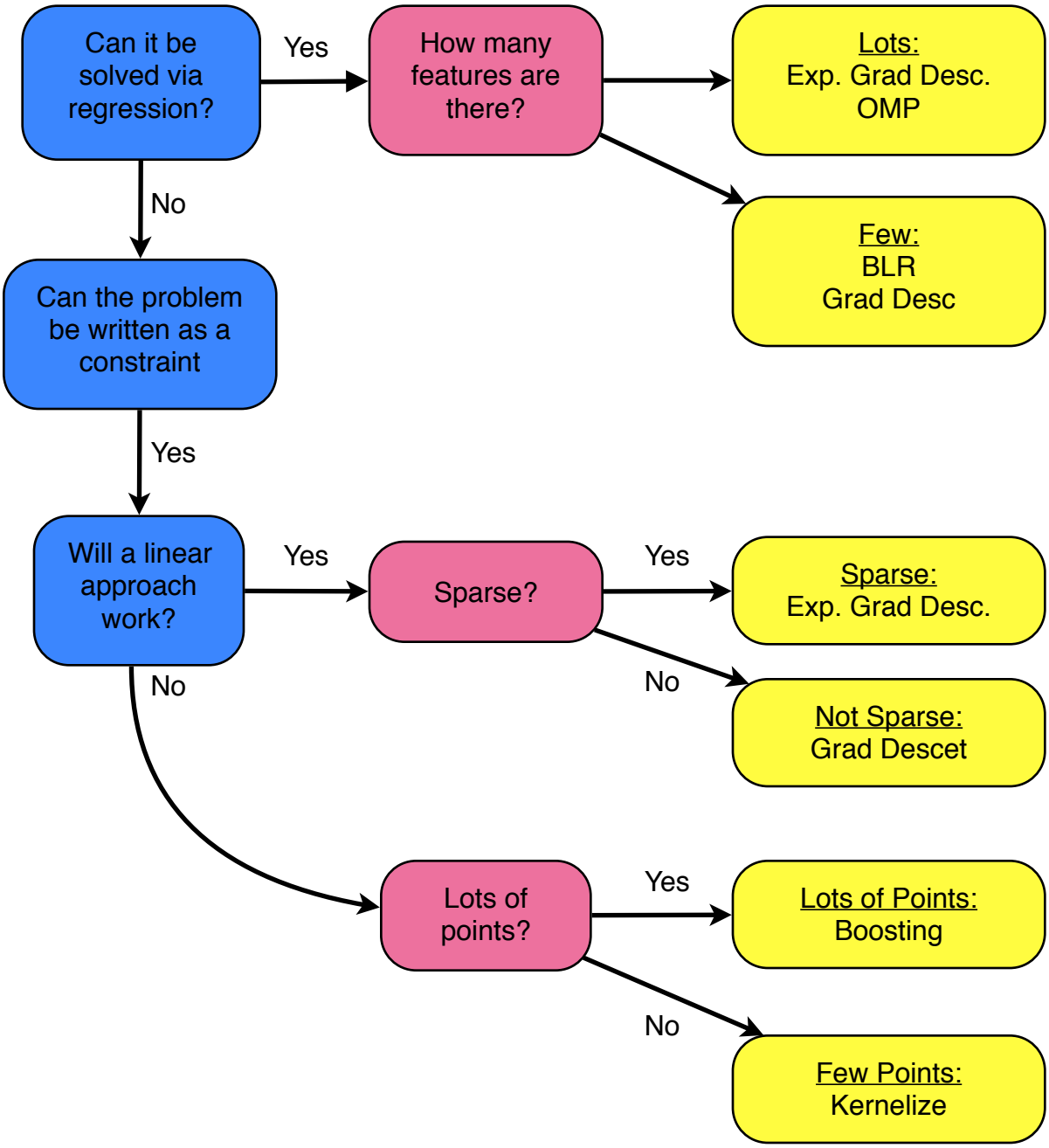
Problems with these methods:

A. Optimization Theoretic
   1) Improvements in performance often require changing multiple parameters at the same time.
   2) It can be hard to predict the goodness of a particular outcome.
   3) Vanishing Gradients: the final output of the system usually has a very small dependence on the initial modules. It may help think of these sorts of systems as behaving like filters- the longer the chain of modules, the greater the filtering. Therefore, gradients are usually very small (although they could be huge).
B. Information Theoretic
   1) These problems have a giant hypothesis space
   2) Example: The terrain classifier module could be trained to produce outputs that result in great system performance...but bear little resemblance to what most people would consider terrain classification.

Overall Advice:

A. Combine global and local cost functions:
   1) Cost: $\alpha \cdot$ local cost + $(1-\alpha) \cdot$ global cost
   2) Most folks chose this approach
B. Start with local training then move to global training.

Algorithm Decision Flow Chart Part 1



Algorithm Decision Flow Chart Part 2

If the above flow chart fails:

Decompose the problem into a set of modules. Try to answer the question: “What are the sufficient steps to get the problem done if each step is done moderately well?” Optimize each module using a smart local process, then move to a global optimization.

If there is too little data, there are a few options:

1. Hire help to develop more data.
2. Expand on your current data set. Adding noise is one approach.
3. Build a system based on current abilities and then run self supervised learning.
4. Use a related task and keep the features identified by this module.
    - “Transfer Learning” or “Semi-Self Supervised”