Linear Function Approximation

A linear function approximator is a function *y=f(x,w)* that is linear in the weights, though not necessarily linear in the input *x*:

```
y = w_1 * f_1(x) + w_2 * f_2(x) + ... + w_n * f_n(x)
```

where *x*, *y*, and *w* can be vectors, the *f_i*() functions can be linear or nonlinear, and *w_i* is the *i*th element of the *w* vector. Examples of linear function approximators include:

- **Lookup table**
  There is a separate weight for each possible value of *x*. There are only *n* possible values for *x*, and *f_i(x)=1* when *x=i* and *f_i(x)*=0 otherwise.
- **Linear**
  The output is just the dot product of *w* and *x*. The individual functions are just *f_i(x)=x_i*, where *x_i* is the *i*th element of vector *x*.
- **Radial Basis Functions**
  Each *f_i(x)* function looks like a smooth bump. Each *f_i()* function has a "center" location, and *f_i(x)* is a monotonic function of the distance from *x* to the center. The "distance" may be Euclidean distance (circular bumps), or there may be a diagonal covariance matrix (ellipsoidal bumps parallel to the axes), or there may be a full covariance matrix (general ellipsoidal bumps). To be a linear function approximator, the bumps must not move or change shape.
- **Wavelets**
  Each f_i(x) is a wavelet, typically the product of a cosine and a Gaussian. This is particularly useful in image applications, because the human visual system seems to use similar functions.
- **CMAC**
  Each *f_i(x)* function has a value of 1 inside of *k* square regions in input space, and 0 everywhere else. A hash function is used to make sure that the *k* squares are randomly scattered. The functions are chosen so that for any give *x*, there will be exactly *c* different *f_i(x)* functions active, and their square regions will be offset from one another slightly. Because of the hash function, the CMAC has the nice property that it makes good use of its weights even when all the training examples are in one small region of input space which wasn't known beforehand. One particularly interesting example of this is when the input space is very high dimensional, but all training examples come from a simple, low-dimensional manifold. For some reason, people often implement CMACs with *k*=1, which destroys this useful property. The hash function was originally proposed to be reminiscent of random neuron wiring in the brain.

Linear function approximators have several nice properties. For supervised learning, the weights can be found immediately at the cost of a single matrix inversion, without any gradient-descent or incremental learning. For reinforcement learning, the weights can be found at the cost of solving a single linear program.

For incremental reinforcement learning algorithms there are also a few useful properties. For a lookup table, almost all the incremental algorithms are guaranteed to converge to optimality. For other linear function approximators, TD(lambda) is guaranteed to converge when doing on-policy training (transitions are trained on with a frequency proportional to their frequency in the Markov chain).

Unfortunately, very few other convergence results are true. TD(lambda) can diverge for off-policy training. Q-learning can diverge even with on-policy training. SARSA can oscillate wildly and periodically forget everything useful it had learned so far. For incremental algorithms, limiting the function approximator to linear function approximators does not help convergence very much.

Fortunately there are ways to ensure that all these algorithms will converge: use the the residual form of each algorithm. In that case, they will converge for both linear and nonlinear function approximators.

There are also results indicating that nonlinear function approximators may be more powerful in general than linear function approximators for learning high-dimensional functions. For example, if the target function is fairly smooth (has little energy in the high requencies), and the function approximator is nonlinear, then it is

known that the number of weights needed for a good fit grows only polynomially with the dimensionality of the input. This is true for such diverse function approximators as sigmoidal neural networks and linear combinations of sine waves. It may even be true for all of the popular nonlinear function approximators. But it has been shown that it is not true for any linear function approximator. This result suggests that linear function approximators may be most useful in cases where the input space is low dimensional, or where the training examples all come from a low-dimensional manifold in a high-dimensional space.

More Information

Back to Glossary Index