Hierarchical Temporal Memory

Michoel Snow

June 22, 2017

A summary of Hierarchical Temporal Models (HTM)

Contents

Sparsely Distributed Representations

Introduction

A sparsely distributed representation, hereon referred to as SDR, is a way of representing information using binary operators where most of the binary operators are OFF at any one time. This is in contrast to a dense distribution in which up to 50% of the operators are ON at a time. Dense distributions cap at 50% ON because there is as much information in a bit being ON as being OFF. Mathematically this is true as the largest number of possible states exists when every bit has equal chance of being ON or OFF. Figure 1 shows a sparsely distribution, where the number of bits ON at any time is 2, and since the total number of bits is 100 the sparsity is $2/100 = 2\%$. Figure 2 shows a dense distribution where 50 out of the 100 bits are ON at any one time and so its sparsity is $50/100 = 50\%$.

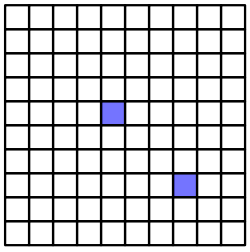

Figure 1: Sparse distribution

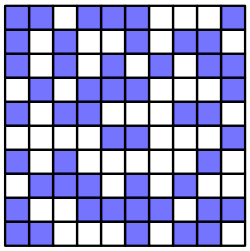

Figure 2: Dense distribution

In addition to sparsity, another measure is capacity, which is the total number of combinations possible given the distribution size and the sparsity. Capacity is calculated as

$$\binom{n}{w} = \frac{n!}{w!\,(n-w)!} \tag{1}$$

where n is the total size of the distribution and w is the number of bits ON at any one time. So for the sparse distribution the capacity is

$$\binom{100}{2} = \frac{100!}{2!\,(100-2)!} = 4950 \tag{2}$$

but even for this sparsity of 2% this value increases exponentially. When the size of the distribution is 256, a sparisty of 2% gives a capacity of 8.8×10^9

One advantage of sparse representations over dense ones is in terms of compression. To represent a maximally dense distribution you need as many bits as the distribution itself, e.g., 256 bits to represent a 256 bit dense distribution. This is because every bit has the same probability of being 0 or 1, ON or OFF. However, in a sparse distribution, this is not the case, and to represent it all you need are the indices of the on bits. So for a 256 bit sparse distribution with a sparsity of 2%, you only need 40 bits to represent it, i.e., 8 bits for the location of each of the 5 ON bits.

Overlap, Matching, and Noise of Sparse distributions

The following section is based off of Ahmad, S., & Hawkins, J. (2015). Properties of sparse distributed representations and their application to hierarchical temporal memory. See that paper for more information about matching of SDR vectors.

When comparing two SDR vectors a useful metric is the overlap, i.e., the number of bits that are ON in the same locations in both vectors. Two SDR vectors are considered matched if their overlap exceeds a threshold of θ. Where θ is a value less than or equal to the number of ON bits in both vectors. Given two random SDR vectors the probability that they are identical, i.e., an exact match, is the inverse of their capacity

$$P\left(x = y\right) = 1/\binom{n}{w} \tag{3}$$

However, you do not need a system where only exact matches, i.e., $w = \theta$ is acceptable, you want a system which can tolerate noisiness in the input. By lowering θ the sensitivity of the system increases but its robustness increases. We can quantify the decreased sensitivity of an inexact match by calculating the probability of a false positive. But first we need to quantify the possible number of SDR vectors which overlap with our given vector x:

$$\left|\Omega_x\left(n, w, \theta\right)\right| = \binom{w_x}{\theta} \times \binom{n - w_x}{w - \theta} \tag{4}$$

Where w_x is the number of components in x that are ON. The left hand side of the product is the ON bit space. It is the number of ways that θ ON bits fit in the total number of possible ON bits in the vector x, i.e., w_x. This is the same capacity formula as seen previously, Equation 3. The right hand side of the product is the OFF bit space. It is the number of ways that the remaining ON bits $w - \theta$ fit in the total number of possible OFF bits, $n - w_x$.

Starting with an extreme example, if the size of the SDR vector was 100, with 36 ON bits, how many possible SDR vectors are there which have an overlap of 36 bits. There is only 1 possible vector which is the size of x, has the same number of ON bits as x and perfectly overlaps x, and that vector is x. But what about if the number of overlap bits was one less than the number of ON bits, how many possible vectors could then overlap with x. As you can see in Figure 3, there are 36 possible ways to have 35 bits ON and 1 bit OFF, and from Figure 4 you can see that there are 64 different ways to have 1 bit ON and 63 bits OFF. Formalizing this through Equation 4:

$$\left|\Omega_x\left(n, w, \theta\right)\right| = \binom{36}{35} \times \binom{64}{63} = 36 \times 64 = 2304 \tag{5}$$

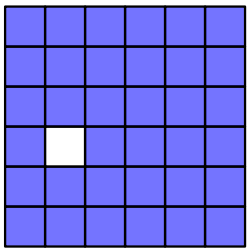

Figure 3: On Bit Space

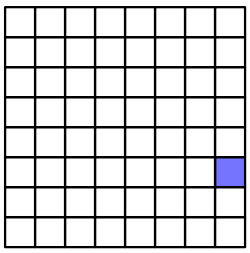

Figure 4: Off Bit Space

Using this equation we can now calculate the (approximate) probability of a false positive as:

$$fp_w^n(\theta) \approx \frac{|\Omega_x(n, w, \theta)|}{\binom{n}{w}} \tag{6}$$

See the original paper for why the approximate holds and what it is an approximate of. As θ increases the chance of a false positive decreases, which makes sense as with increasing θ there is a decreasing number of vectors which satisfy the overlap condition and so a smaller chance of a false positive. Additionally, the chance of false positives decreases as a function on the size of the vector. This makes sense intuitively, because as the size increases, the chance that another vector will 'accidentally' satisfy the overlap conditions, decreases. So for a vector of size 2048, with 40 ON bits and an overlap requirement of 30 bits, the chance of a false positive is 1×10^{-49}

Subsampling

Since the chance of satisfying the overlap condition decreases as a function of the vector size, with a large enough vector size you do not even need to match to the original full vector. You can recognize the desired pattern by matching instead to a small subset of the active ON bits. For our example SDR vector of size x with w_x ON bits, let x' be a subsampled version of x, such that $w_{x'} \leq w_x$. In order to calculate the probability that some random vector y will match with our new subsampled vector x' we first need to determine how many y vectors match our subsampled vector using a modified version of Equation 4:

$$|\Omega_x(n, w_y, \theta)| = \binom{w_{x'}}{\theta} \times \binom{n - w_{x'}}{w_y - \theta} \tag{7}$$

Then modifying Equation 6 given a matching threshold of $\theta \leq w_x$

$$fp_{w_y}^n(\theta) \approx \frac{|\Omega_{x'}(n, w_y, \theta)|}{\binom{n}{w_y}} \tag{8}$$

As you can see the only difference between Equation 6 and Equation 8 are the vectors being compared. Using a vector with a size, n, of 2048 (which is a typical HTM vector size), where $w_x = w_y = 40$, the subsampled vector is half the length of the original vector x, so $w_{x'} = 20$ and the threshold for matching, θ, is equal to 10, the chance of a false positive is about 1 in 2.5×10^{12}.

Union Property

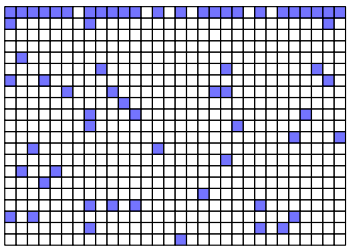

Figure 5: SDR Union. The top row is the union of all rows underneath

Given a set of M SDR vectors what is the probability that a new SDR vector satisfies the overlap threshold for any of the M vectors? The naive way to do this would be to compare the new vector to each of the M vectors, but given a very large M, this becomes a problem. Another approach is to create a union of the M vectors and compare the new vector to the union. A union of M vectors has as ON bit for each location in which any of the M vectors has an ON bit at that location, i.e., it is an OR function, as seen in Figure 5. The advantage of this property is that a fixed-size SDR vector can store a dynamic set of M SDR vectors. To check if a new SDR vector matches any of the M SDR vectors, it can be compared against the single Union vector instead of all the individual M vectors.

The issue with Union vectors, as seen in Figure 5, is that the more vectors included within the union the greater the probability of a false positive. For a single SDR vector the probability that any given bit is 0, i.e., OFF, is $1 - \frac{w}{n}$. For M vectors with the same w and n the probability that any single bit is OFF, in the union vector, is given by

$$p_0 = \left(1 - \frac{w}{n}\right)^M \tag{9}$$

And the probability that any given bit is ON is given by $1 - p_0$. The probability that a new SDR vector perfectly aligns with the union vector, i.e., $\theta = w$, in other words, a false positive can be calculated as

$$p_{fp} = (1 - p_0)^w = \left(1 - \left(\frac{w}{n}\right)^M\right)^w \tag{10}$$

Using the example shown in Figure 5 where $n = 30$, $w = 2$ and $M = 20$ the chance of a false positive is about 56%. While this is high, this is still kind of surprising given the amount of ON bits in the union vector. If we go back to our standard SDR sizes of $n = 2048$ with a $w = 2$, if $M = 20$ the false positive rate is about 1 in 2670, but simply increasing w to 20, dramatically decreases the false positive rate to about 1 in 1×10^{13}. Even increasing M to 50 only gives a false positive rate of about 1 in 170 million.

This was all done when we required a perfect match between the new SDR vector and the union vector, i.e., $\theta = w$, but what happens if we relax the overlap condition? The expected number of ON bits in a union vector is

$$\widetilde{w}_x = n\,(1 - p_0) \tag{11}$$

This growth rate, which is slower than n, is the reason why as n and w increase, the false positive chance drops dramatically. Given

this property the expected size of the union overlap set, i.e., the set of vectors which overlap with the union vector, is given by:

$$E\left[\left|\Omega_x\left(n,\widetilde{w},\theta\right)\right|\right] \approx \binom{\widetilde{w}_x}{b} \times \binom{n-\widetilde{w}_x}{w-\theta} \tag{12}$$

Thus the chance of a false positive can be approximated as

$$fp_{\widetilde{w}}^{n}\left(\theta\right) \approx \frac{\left|\Omega_x\left(n,\widetilde{w},\theta\right)\right|}{\binom{n}{w}} \tag{13}$$

As expected, this error increases as the threshold is lowered. For example, if $n = 1024$, $w = 20$, $M = 20$ and $\theta = 20$. The false positive rate is 1 in 5 billion, decreasing θ to 19 increases the chance of a false positive to 1 in 123 million, and then higher to 1 in 4 million when θ is 18. However, if you increased n to 2048 with the same w and M, and setting $\theta = 18$, the chance of false positive drops again to 1 in 223 billion, illustrating the robustness of the union vector.

Encoding

Introduction

Now that we understand the benefits of an SDR, we now have to figure out how to apply it to the problem we want to solve. The first step in this problem is converting whatever data into an SDR format, a process termed encoding. In this section I will go through the different encoding methods, but before that can happen we need to set the design principles for the encoders.

1. Semantically similar data should result in SDRs with overlapping active bits and the more similar the more overlap there should be
2. The same input should always produce the same SDR as output
3. The output should have the same dimensionality (total number of bits) for all inputs.
4. The output should have similar sparsity for all inputs and have enough one-bits, i.e., bits that have a value of one or are ON, to handle noise and subsampling

Scalar Encoder

A scalar encoder is the simplest type of encoder. In this type of encoder, with total number of bits n, there are a consecutive string of

ON bits of length w, which can represent any value in the range from the minimum value, *minval* to the maximal value, *maxval*. *minval* is represented by the bits $1 : w$ being ON and all other bits being OFF. The next bucket of values is represented with bits $2 : w + 1$ and so on until *maxval* is represented with bits $n - w : w$ ON and all other bits OFF. To create this type of encoder requires 4 parameters

1. *minval* - The minimum value,
2. *maxval* - The maximal value
3. w - The number of active bits
4. Exactly one of the following
   (a) n - The size of the SDR
   (b) The resolution of the SDR
   (c) The number of buckets in the SDR

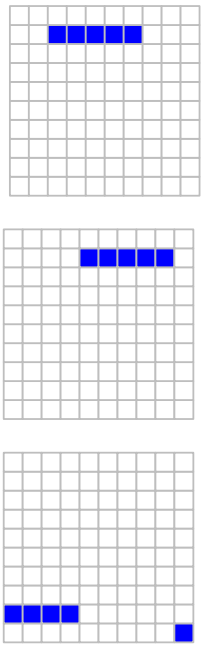

Figure 6: Scalar encoding of 85°, 87° and 12°

The resolution of the SDR is the difference for which inputs greater than or equal to that value are guaranteed to have different representations; conversely, inputs which have a difference less than that value will have the same representation. The number of buckets in the SDR is the number of groups that the values between *minval* and *maxval* will be split into. Those last three terms can all be calculated from one another, hence only one of them is needed, i.e., $n = buckets + w - 1 = \frac{maxval - minval + 1}{resolution} + w - 1$. To determine the bits that are ON for any value, v, in the given range, you need to determine the bucket, i, it is in:

$$i = floor\left[buckets \times \frac{v - minval}{maxval - minval + 1}\right] \tag{14}$$

Then activate w consecutive bits starting from the index i. It is easy to see how this type of encoder satisfies the 4 conditions.

When setting up this type of encoder the number of buckets and number of ON bits should reflect the noisiness/variability in the data. The more noisy the data, the more values that should have the same SDR representation and thus the larger the buckets should be. This does not mean that more bits are on, i.e., that w is larger, just that a wider range of values should all fall into the same bucket, i.e., the resolution should be larger. However larger buckets, come at the cost of decreased precision.

The downsides of this approach are that it requires you have a general sense of the range of the data, as all values below the minimum value will have the same representation as the minimum value and all values greater than the maximum value will have the same

representation as the maximum value, if the range goes from 0 to 100, -50 has the same representation as 0 and 1 million has the same representation as 100. One solution to this issue is to apply periodicity to the SDR. Instead of having a minimum and maximum value for the SDR, you just need a range of values which the SDR represents. For example if the range is 100, then instead of 150 having the same representation as 100, as it would in the simplest scalar encoder, any value greater than the max would wrap around and so 150 would have the same representation as 50. While this might not seem so helpful when encoding numbers (although sometimes it is) this is very useful for signals with inherent periodicity like the days of the week. For example, if I wanted to build a scalar encoder of a calendar which encoded not only the date but also the day of the week, for my day of the week encoder I don't care if it is Wednesday this week or Wednesday next week, all inputs of Wednesday should have the same SDR representation.

Another issue with this type of encoder is the limited number of states it can represent, specifically, for any given number of n bits with w ON bits the maximal number of states/buckets is $n - w + 1$. The next type of encoder solves this issue.

nupic implementation

The code for the scalar encoder, as of nupic version 0.6.0, is of the form

```
ScalarEncoder(w,minval,maxval,[periodic=False, n = 0, radius
 = 0, resolution = 0, name, clipInput=False, forced=False])
```

w	Number of ON bits which encode a single value, must be odd to avoid centering problems
minval	Minimum value of the input signal
maxval	Maximum value of the input signal
periodic	If true, then input "wraps around" such that the input must be strictly less than maxval. Default is False
radius	inputs separated by more than the radius will have non-overlapping representations
n	Total number of bits in the output must be $\geq$ w
resolution	Inputs separated by more than the resolution will have different, but possible overlapping, representations
name	Optional string which will become part of the description

```
>> ScalarEncoder(w=3,minval=-10,maxval=10,
   periodic=False,resolution=2,
   forced=True).encode(8)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0]
>> ScalarEncoder(w=3,minval=-10,maxval=10,
   periodic=False,resolution=2,
   forced=True).encode(9)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1]
>> ScalarEncoder(w=3,minval=-10,maxval=10,
   periodic=True,resolution=2,
   forced=True).encode(9)
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1]
```

clipInput	If True, non-periodic inputs < minval or > maxval will be clipped to minval and maxval, respectively
forced	If True, skip some safety checks. Default is False

There are three mutually exclusive parameters that determine the overall size of of the output. Exactly one of **n, radius, resolution** must be set. "0" is a special value that means "not set".

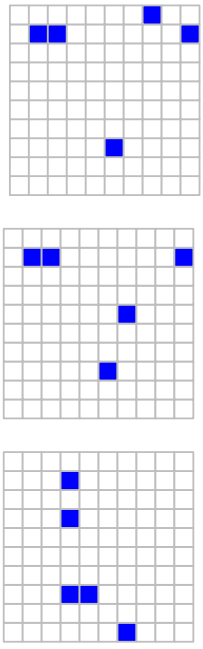

Figure 7: RDSE encoding of 85°, 87° and 12°

Randomly Distributed Scalar Encoder

In the scalar encoder each value is represented by a value of one at a specific bit, i, followed by the corresponding $w-1$ bits. In a randomly distributed scalar (RDS) encoder, a hashing function is applied to the individual bits before they are assigned locations in the SDR. This allows for an infinite number of representations as the bits are no longer consecutive but still maintain semantic similarity. The semantic similarity is maintained as each ON bit represents a certain static set of values, as dictated by the resolution. For example, if the resolution $= 5$ and $w = 3$, then for any random value x, the 3 ON bits represent $[x-7 : x-3]$, $[x-2 : x+2]$, and $[x+3 : x+7]$ even though they are in non-consecutive locations. So if you now want to encode $x+4$, the locations of the $[x-2 : x+2]$ and $[x+3 : x+7]$ ON bits are already known, and you just need to calculate a location for the $[x+8 : x+12]$ bit based on the hashing function.

However, with this representation there is the possibility of collision, i.e., that multiple values can have the same representation. This can be avoided in two ways, the first is to increase the size of the SDR, i.e., n, the second, and more important method, is to increase w. With a larger w even though one or two bits for different values might overlap there is a much smaller probability that there will be perfect overlap (or practically, an overlap greater than threshold). That being said, w should still be limited so that sparsity is maintained. For the math behind this see the SDR section Overlap, Matching, and Noise of Sparse distributions.

For the RDS encoder you need 3 parameters

1. w - The number of active bits
2. n - The size of the SDR
3. The resolution of the SDR

This time you need both n and the resolution as there is no minimum or maximum value, so n tells you the size of the SDR and the resolution tells you after how many values do you need to change your representation.

nupic implementation

The code for the RDSE, as of nupic version 0.6.0, is of the form

```
RandomDistributedScalarEncoder(resolution, [w=21, n=400,
     name=None, offset=None, seed=42, verbosity=0])
```

w	Number of ON bits which encode a single value, must be odd to avoid centering problems
n	Total number of bits in the output must be $\geq$ w
resolution	Inputs separated by more than the resolution will have different, but possible overlapping, representations
name	Optional string which will become part of the description
offset	Floating point offset used to map scalar inputs to bucket indices. If set to None, the very first input that is encoded will be used to determine the offset.
seed	Seed used by the numpy random number generator. If set to -1, the generator will be initialized without a fixed seed

```
>> RDSEenc = RDSE(resolution=5, w=3, n=20)
>> RDSEenc.encode(50)
[1, 0, 0, 0, 0, 0, 1, 0, 0, 0,
1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>> RDSEenc.encode(45)
[1, 0, 0, 0, 0, 0, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
>> RDSEenc.encode(5)
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 1, 0, 0]
```

Date Encoder

The date encoder encodes multiple aspects of the date simultaneously. Practically it is a multiencoder made up of multiple scalar encoders (can also use RDSE, if desired).

nupic implementation

The Date Encoder works slightly differently than the previous encoders. All encoders, by default, have value of 0, which the code interprets as not being implemented. You have to specify which aspects of the date encoder you want to implement. This is done by setting the parameters only for the encoders of interest. The input into the resulting encoder must be a datetime variable. The code for the date encoder, as of nupic version 0.6.0, is as follows.

```
DateEncoder(season=0, dayOfWeek=0, weekend=0, holiday=0,
   timeOfDay=0, customDays=0, name = ", forced=True))
```

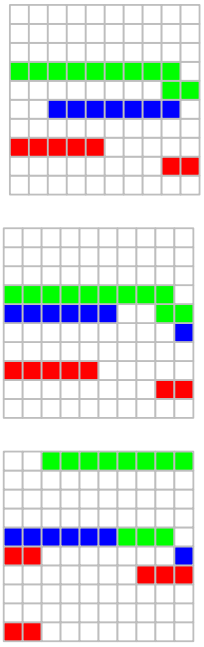

Figure 8: Date encoding of 3/1/17 12pm, 3/4/17 12pm and 12/15/17 11pm

season	Season of the year. Accepts either tuple of $(w, radius)$, where radius is the size of the season in days, or integer which corresponds to w and uses the default radius of 91.5 days.
dayOfWeek	Day of week, starting from Monday. Accepts either a tuple of $(w, radius)$, where radius is the number of days per bucket, or an integer, which corresponds to w and uses the default radius of 1 day. The total number of bits, n, is $w \times \lceil 7/radius \rceil$
weekend	Binary for the weekend (Sat, Sun). Accepts an integer which corresponds to w. $n = w \times 2$
holiday	Binary for holidays built into nupic (as of now only christmas). Accepts an integer which corresponds to w. $n = w \times 2$
timeOfday	Time of day based on 24 hour clock. Accepts either a tuple of $(w, radius)$, where radius is the number of hours per bucket, or an integer, which corresponds to w and uses the default radius of 4 hours. $n = w \times \lceil 24/radius \rceil$
customDays	Custom list of days. Accepts a tuple of $(w, DayList)$, where DayList is a list of datetime dates to be used.

```
>> DE = DateEncoder(season = 3)
>> dt = datetime.strptime('01/01/17
   12:03', '%m/%d/%y %H:%M')
>> DE.encode(dt)
[1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
>> dt = datetime.strptime('02/01/17
   12:03', '%m/%d/%y %H:%M')
>> DE.encode(dt)
[1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>> DE = DateEncoder(season = (3,200))
>> DE.encode(dt)
[1, 1, 0, 0, 0, 1]

>> DE = DateEncoder(dayOfWeek = 1)
>> DE.encode(dt)
[0, 0, 1, 0, 0, 0, 0]
>> DE = DateEncoder(dayOfWeek = (1,3))
>> DE.encode(dt)
[0, 1, 0]

>> DE = DateEncoder(weekend = 3)
>> DE.encode(dt)
[1, 1, 1, 0, 0, 0]
>> dt = datetime.strptime('02/04/17
   12:03', '%m/%d/%y %H:%M')
>> DE.encode(dt)
[0, 0, 0, 1, 1, 1]
```

Encoder Application

At the end of the day, encoders are just like any other tool and in addition to understanding how they work you also need to understand how to use them. The first step is knowing which data to encode. In most cases there will be more parameters available than are strictly necessary. In **??** I go into how the model can decide which encoders are useful and how to encode them, but that only works with the built in encoders. If you start using your own encoders you will have to decide which ones to include and the appropriate parameters. As a general rule you only want to encode information once, e.g., don't need to encode the temperature in Celsius and Fahrenheit.

The second step is understanding how to set the parameters and how they are dictated by the data you are encoding. For example if you want to represent the ambient temperature in Celsius with a scalar encoder, you wouldn't set a max value of $400°$, instead it would probably be around $40° - 50°$ depending on how much you care about accurately recording outliers. Second, the resolution will very much depend on why you are encoding the variable. If you are concerned about global warming, where every 0.1 degrees matters

then the resolution should be much smaller than if you are trying to predict ice cream sales where differences of $1^\circ - 2^\circ$ don't really matter.

Spatial Pooling

Swarming

Once you have decided on the encoders for your data, the next step is determining the parameters of the HTM that best fit your data as well as which parts of your data to use. Just because you have the data does not automatically make it relevant when trying to predict your outcome. Swarming is the process within the nupic code which determines the best model for the given dataset. In this context, best means the most accurate model based on component selection, i.e., choice of optional components such as encoders, spatial pooler, temporal pooler, etc., and parameter optimization for those selected components.

Swarms are only needed for prediction not for anomaly detection

go back to the swarming in nupic video for more information on swarming and what some of the more deep down parameters mean

To use your won parameter or permutations when running the swarm, append the swarm.py code with your file name, e.g.,

»> python swarm.py permutations.py

- Small swarm size - Used mainly for debugging
- Medium swarm size - best for most cases but will only look at pairs of field combinations
- Large swarm size - Will go beyond pairs of field combinations and will look at more permutations of parameters, but will take a lot more time