# Representations for Subsymbolic AI

## 1 The Many Faces of Representation

In a typical subsymbolic AI system, there are at least three types of state representation to consider:

1. Raw - The original formulation of a system state.
2. Id- A numeric version of the original formulation that uniquely expresses each distinct state and serves as a key for table lookup (of states and information learned about them).
3. Code - An encoding of the original formulation to use as input to the subsymbolic machinery, e.g. neural network.

For example, the 3 representation types for a hand of cards in 5-card-draw poker might be:

1. Raw - $[\clubsuit Q, \diamondsuit A, \heartsuit J, \spadesuit 5, \spadesuit 9]$
2. Id - [(1,12),(2,1),(3,11),(4,5),(4,9) ]
3. Code - [(1000 00000000000010) (0100 1000000000000) (0010 0000000000100) (0001 0000100000000) (0001 0000000010000) ]

The raw version might be the input from the screen or a file, while the id simply converts each suit into an integer using the standard ordering (clubs,diamonds,hearts,spades) and each value into an integer between 1 (ace) and 13 (king). Internally, a language such as Python could use the id formulation to generate a hash key.

Actually, Python could hash on the raw representation as well – since it can hash on just about anything – but in some instances, you the programmer may need to devise a clever data structure for storing and retrieving states, so you may need to work with a purely numeric form of the original (raw) representation. In other instances, you still need to be aware of some details of the hashing mechanism. For example, if you fail to sort the cards in the raw representation prior to id generation, then Python could hash two permutations of the same 5 cards to two different addresses - an undesirable result for most card games.

Finally, the encoded version of the 5-card hand employs a series of *one-hot* vectors (i.e. exactly one of k bits is a 1, while the rest are 0's) to represent the suit and value of all 5 cards. Hence, each state code contains exactly 10 1's.

During problem-solving, the system might generate or read a raw state, S. It would then need to determine whether or not it has seen (and thus compiled information on) S. To do so efficiently, it would probably convert S to an id and then use that as the key into a large state hash table. If found, the object representing S may already contain the code

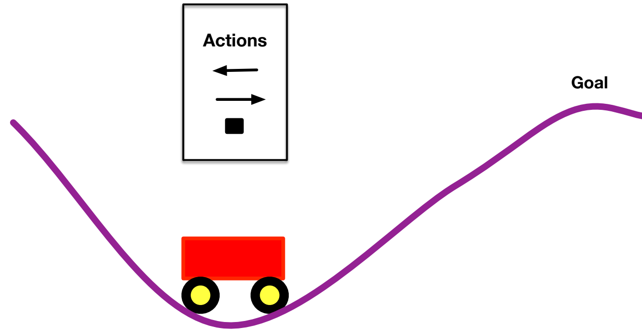


Figure 1: The Mountain Car problem.

for S, which the system could then feed as input to the subsymbolic machinery, which might predicted a class label (for deep learning), or an evaluation or vector of action probabilities (for reinforcement learning). If S is new, then such an object will be created and added to the hash table.

Many AI tasks involve extensive search through huge state spaces, with the same states arising many times during that process, and with each state involving a relatively intricate raw representation. The manner in which the system represents and stores states can have a profound effect upon its overall performance, so careful consideration should be given to representations.

In this document, the focus is on the encoded version of a state and how these encodings influence a neural network (using backpropagation) and its ability to achieve the appropriate levels of both specialization and generalization for the problem-solving task at hand.

## 2 The Mountain Car Problem

As a simple (but popular) example, the mountain car problem illustrates some of the fundamental issues of encoding system states in subsymbolic representations. This classic control problem involves a cart on a sloped landscape. As shown in Figure 1, the cart sits at the bottom of a one-dimensional valley, and the goal is to move the cart to the top of the right side of that valley. Unfortunately, the cart has very limited power and cannot simply drive up the steep slope. It must use the left and right slopes to help build enough momentum to make it all the way up the right side. Hence, the solution involves a *sloshing* motion whereby the cart slides back and forth between the two slopes while supplying just enough power to get a little higher with each oscillation.

At each timestep, the control system must choose between three actions: 1) apply a small force (-1) that accelerates the cart to the left, 2) apply a small force (+1) that accelerates it to the right, and 3) apply no force (0).

The state of the cart is defined by two variables, location (x) and velocity ($\dot{x}$). The dynamics of the system involve the following update equation for velocity:

$$\dot{x} \leftarrow \dot{x} + (.001)F - (.0025)cos(3x) \tag{1}$$

with the lower and upper bounds on velocity set at -0.07 and 0.07, respectively, and with F being the most recent action (-1, 1 or 0). Location is then updated as:

$$x \leftarrow x + \tau\dot{x} \tag{2}$$

with the lower and upper bounds for x being -1.2 and 0.6, respectively, and $\tau$ being the timestep. The relationship

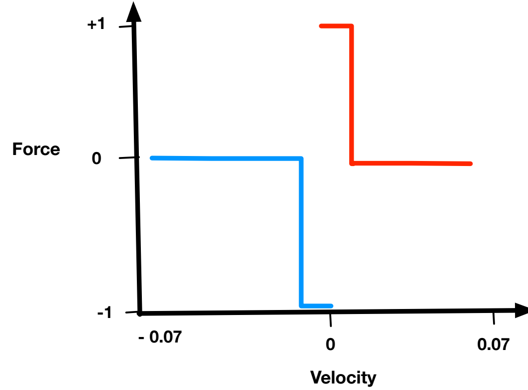


Figure 2: A mountain-car strategy that bases the control force solely upon the velocity. The cart coasts until just before stopping, at which point a force in the same direction as the current (small) velocity is applied. The blue line illustrates the approach to zero velocity from negative values, while the red line displays the approach from positive values.

between x and y (the height / depth) is given by:

$$y = cos(3(x + \frac{\pi}{2})) \tag{3}$$

The goal of a controller is to map the value pair (x, $\dot{x}$) to an action choice (-1, 1 or 0). This seems pretty straightforward, but issues arise due to two different sources of nonlinearity:

1. Along a single input dimension, e.g. $\dot{x}$, the appropriate action is often a nonlinear functon of $\dot{x}$, and possibly noncontinuous as well. For instance, consider the strategy to only apply force just before the cart comes to a halt on either slope, i.e. just before $\dot{x}$ reaches 0, and in the direction that will prolong the cart's ascent for one extra timestep, giving it a little extra potential energy. The plot of force as a function of velocity then resembles that of Figure 2: a positive force is applied when velocity approaches zero from the positive direction, and a negative force is applied when it approaches from the negative direction. Let us call this the *climbing* approach.

2. The proper action is often a function of several dimensions, not just one. Basing the control signal purely upon the location seems ineffective, since the decision to coast or apply energy should take into account both location and velocity. Although velocity seems to be the main factor, ignoring x completely could easily cause problems, as when the cart is on the far left, near location -1.2, a height that should allow it to coast downhill and up to the goal. Applying extra force near -1.2 could be unecessary (thus inefficient), or, worse, send the cart out of bounds on the left. Also, consider the case of an alternate strategy that favors *pumping* the accelerator exactly when the cart reaches its maximum speed of a given pass, i.e. at the low-point of the valley. In this case, the timestep of the action is determined by a single factor, x, but the direction of the force requires velocity information, since this strategy dictates applying a force in the same direction as the current motion, but only on the valley floor. Once again, both variables are needed to make the proper decision, and only in a few restricted cases could we expect to base that decision on a simple linear combination of the two variables: the relationship between the two variables and the action will almost certainly be nonlinear.

# 3 Specialization

In theory, a simple neural network should be able to learn the proper mapping from state to action, but the nonlinearities mentioned above complicate the design. In Figure 3, the network on the left has an impossible task of learning weights

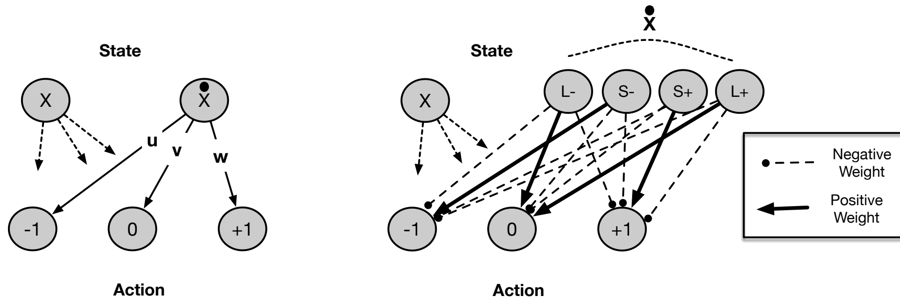


Figure 3: Neural networks for mapping mountain-car states to three control actions (-1,0,+1). Both represent attempts to manifest the *climbing* strategy whereby the force is exerted just before the cart comes to a halt along either slope, and in the same direction as the small remaining velocity. (Left) Both location and velocity are encoded as individual values. (Right) The velocity value is binned such that only 1 of 4 neurons is active: large negative (L-), small (near zero) negative (S-), small (near zero) positive (S+), and large positive (L+).

u, v and w to manifest the climbing strategy. As in all neural networks, these weights encode relationships between the nodes at their endpoints, but each weight can only capture one influence. Hence, weight w can represent a large or small, positive or negative correlation between velocity and the tendency to choose the +1 action. But Figure 2 shows that no simple correlation exists between velocity and any of the three actions.

However, the right side of Figure 3 provides more hope. There, the velocity value is pre-processed by a binary binning operation, using bins (ranges) of varying sizes such that only those velocity values very close to zero fit into the S- and S+ bins. The encoding for velocity thus becomes *one-hot*. This higher resolution encoding (in terms of the number of neurons) provides a useful semantic separation such that different velocity values can be *treated* differently, i.e. they can be multiplied by different weights reflecting different correlations / relationships with the action nodes.

At first glance, this network appears to capture the essence of the climbing strategy: small negative and small positive velocities have a positive relationship to actions -1 and +1, respectively, while large negative and large positive values both increase the positive influence upon the null action (0). Thus, higher input resolution enables greater nuance inside the network. Also, by sparsifying the representation with the one-hot encoding, the network can learn very specific responses for each of the 4 velocity regimes.

Going further along this path of specialization, we could also bin the values of x to resolve nonlinearities in the relationship between location and action. This would make more sense for the *pumping* strategy, where force should only be applied near the valley floor, i.e., where $x \approx 0$. The direction of the force at this point equals the direction of the velocity, so the action choice depends upon both location and velocity for this strategy, as reflected in the networks of Figure 4. The top network employs two one-hot encodings, thus enabling specialized treatment of both location and velocity. In the bottom network, the input states represent mutually-exclusive combinations of location and velocity such that a single one-hot encoding covers the entire input state. This enables extreme specialization of the network weights for each of the possible combinations.

For any number of input variables and any number of bins per variable, these conjunctive states are always possible to generate, thus producing large input layers with one-hot encoding and essentially forcing the network to devise special treatments to each possible scenario. Alternatively, one can bin only the individual variables and then add one or more intermediate layers of neurons between the input and output / action layer. Then, if training data exists, the network can learn to produce one-hot (or at least sparse) representations in these middle layers. These intermediate activation patterns can then receive *special treatment* via specialized weights between themselves and the output layer. The beauty of backpropagation for supervised learning is that it will often learn only those sparse representations needed to perform the given task. Hence, it might produce unique intermediate patterns for the conjuctive states (S,N) and (S,P) but map all other combinations to a third pattern that triggers the null action. In essence, backpropagation learns

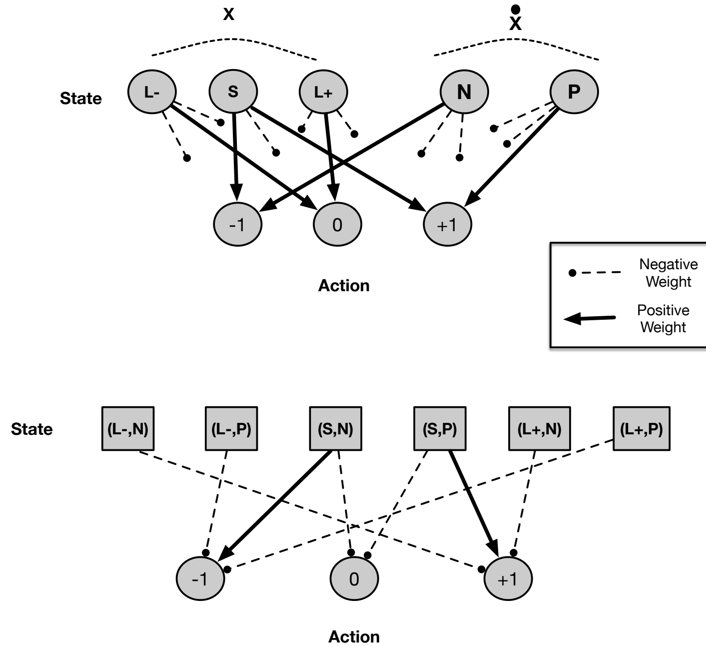


Figure 4: Neural networks for mapping mountain-car states to three control actions (-1,0,+1). Both represent attempts to manifest the *pumping* strategy, whereby the force is exerted when the cart reaches its maximum velocity near the bottom of the valley ($x \approx 0$), and in the same direction as that velocity. (Top) Both location and velocity values are binned to yield two one-hot representations. Bin ranges for location: large negative (L-), small positive or negative (S), large positive (L+). Bin ranges for velocity: negative (N) and positive (P). Here, action nodes require two positive inputs to fire. (Bottom) Conjunctive binning yields six mutually exclusive input states (rectangles) such that only one input node is active for any mountain-car state. Here, action nodes require a single positive input to fire.

the proper combination of specialization and generalization.

So far, we have only discussed specialization. The more bins per variable, and the more conjunctive input factors, the greater support for specialization within the network. However, many input situations should be treated similarly. In many cases, these situations engender similar encodings, and thus it should be relatively straightforward to map them to similar outputs. Resemblance of input patterns implies overlap, and one-hot encodings have no such overlap: the intersection of active bits in two different one-hot encodings is the null set.

One-hots support specialization but make generalization much harder. Clearly, the balance between specialization and generalization requires special attention to the representation. Can we get all the advantages of binning and one-hot encodings while also supporting generalization?

# 4 Generalization

Go back to Figure 2 and note that state variables that have not been expanded by binary binning are represented by real scalars. Thus, if the state, $(x,\dot{x})$ is S = (0.2,0.03), then the vector (V) of outputs for the 3 action nodes produced by S should be very similar to the vector V* produced by S* = (0.24,0.03). In short, the network should *generalize* over a whole region of states around S, essentially lumping them all into the same category of *states that produce A(V)*, where A(V) is the winning action in vector V, i.e., the action node with the highest activation level. So we would

expect A(V) = A(V*). This implicit aggregation of similar states into clusters that elicit the same output is the essence of generalization and abstraction ... and it arises naturally in networks that employ direct, real-valued coding of inputs. These networks get generalization *for free*, but, as discussed above, have trouble specializing.

As also shown above, nonlinearities in the mapping from states to actions call for the use of binning (and other techniques) to **expand** the representations of input values such that different sub-ranges of the value can be treated differently, via different weights. However, when that expansion is the (very popular) one-hot binary vector, the network loses its innate ability to generalize and must **learn** proper abstractions instead. For instance, if the one-hots (0100) and (0010) should map to the same outputs, then the network must learn that the weights emanating from the second input node are very similar to those from the third input node. This can be difficult for backpropagation, especially when there are many such outgoing weights. It is more likely that backpropagation will learn two, very different, weight vectors that do perform similar functions but, to the human eye, appear totally dissimilar.

The trick is to employ an expansion code that still provides some degree of *gratuitous generalization*, i.e., one that does not force backpropagation to learn some of the abstractions that are obvious to the human user of the system. We still want similar input states to have similar representations, but we need to insure that the format of the input encoding allows backpropagation to easily detect and exploit those similarities. Those likenesses easily detected by humans are not always so straightforward for a machine, and vice versa.

This calls for representations that support overlap between encodings for different states, but that also allow significant separation to enable *special treatment* of diverse states. For example, the binary encodings of real values do constitute representations with these two basic properties. However, strict binary codes have so many unnatural discontinuities that they only confound learning: although 63 and 64 are nearly identical values for a variable with range [0,100], their 7-bit binary representations have a confounding dissonance: 0011111 versus 0100000. A neural network would struggle mightily to learn that 0011111 and 0100000 should map to similar outputs. Gray codes remedy this problem, since they guarantee that all adjacent integers have very similar encodings, but they can also produce unnatural overlap: the Gray codes for 5 and 13 are 01111 and 10111, respectively, thus burdening the network with learning that these two similar inputs actually represent completely different values that (in many cases) should map to divergent outputs. Unary codes (where 5 is 1111100000000000 and 13 is 1111111111111000 for integers in the range [0,16]) handle both of these problems but suffer inefficient bloating for large numbers.

There is no perfect solution to this representation problem, but many *reasonable* solutions exist due to the special ability of backpropagation to implicitly *figure out* which inputs should be separated and which should be lumped into the same category when determining appropriate outputs. As long as the initial representation does not use oversized bins (that aggregate too many raw states into exactly the same encoding), the neural network may have a chance of attaining the proper combination of abstraction and specialization for a given problem. However, there are situations when the user can assist the neural network via a judicious choice of input representation, and coarse codes serve just that purpose.

# 5 Coarse Coding

A coarse code permits extreme expansion coding via conjunctive binning, but it also supports overlaps between shifted bin groups such that each raw state has a one-hot encoding for each group. Thus, the expansion code is sparse (few 1's) but still permits overlapping codes for similar input states. As a result, both specialization and generalization become straightforward for the neural network.

Returning to the mountain-car problem, the lower network of Figure 4 illustrates one key aspect of many coarse-coding schemes: input variables are binned and then combined into a conjunctive state. The coarse coder works in an n-dimensional space, where n is the number of state variables. It divides that space into regions, that may or may not overlap (and may or may not be of equal size), and the encoding of any state (typically a single point in that space) is

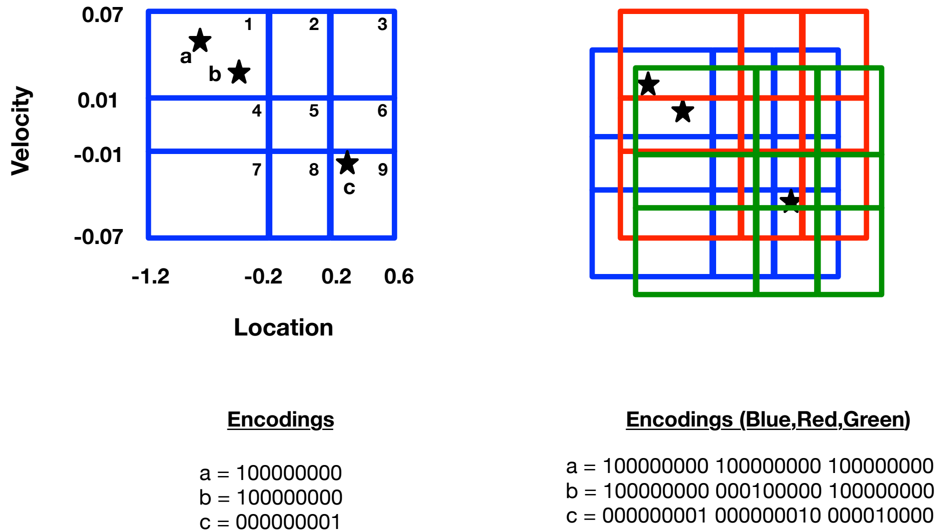


Figure 5: Tile coding for the mountain-car problem. (Left) Indices for each tile / region appear in the upper right-hand corner; stars denote the locations of 3 states (a,b,c) in this 2-dimensional space. One-hot, binary encodings based on region membership appear for each state. (Right) Three overlapping grids yield three concatenated one-hot encodings for each state. Note that the similar states (a and b) have similar, but not identical, encodings.

then given by the indices of the region(s) in which that point lies.

Although many coarse-coding schemes exist (as covered by Sutton and Barto [2] (pages 215-220)), I will focus on tile coding in the remainder of this document.

## 5.1 Tile Coding

In the lower network of Figure 4, the bins for the two variables are: (L-,S,L+) for the location and (N,P) for the velocity. To make this a little more realistic, use the same bin (L-,S,L+) for the velocity. We can then quantify the bin ranges as ([-1.2,-0.2],(-0.2,0.2],(0.2,0.6]) for location and ([-0.07,-0.01), [-0.01,0.01], (0,01.07]) for velocity, based on the original constraints of the mountain-car problem.

Figure 5 illustrates tile coding in terms of the 2-dimensional spaces and the encodings of individual system states. First of all, note that although most tile codings employ equal-sized regions, heterogeneous sizes, as in the figure, can be beneficial for introducing representational biases that the user believes will enhance performance. The left side of the figure displays a simple conjunctive binning operation, with each state having a one-hot vector representation. This fails to separate states a and b, thus forcing their generalization to a shared category. On the right, a combination of three overlapping grids engender a coarse encoding consisting of three one-hots per state. These codes enable states a, b and c to be handled differently, if desirable, but they also clearly reflect the close similarity of states a and b, thus allowing those two states to be mapped to similar outputs when desirable. Thus, the coarse (yet still sparse) encoding gives the neural network enough raw material to find the proper balance between specialization and abstraction.

The issue of coarse coding arises quite frequently in the control of systems whose states are conventionally expressed as vectors of real values, where *control* means mapping states to actions. As shown above, simply feeding those real values directly into a neural network can create a very difficult learning task due to the (often numerous) nonlinearities. In some cases, it may be impossible for backpropagation to discover an effective set of weights. By binning each

variable, and then creating conjunctive bins, and, finally, adding overlapping bin groups, the coarse coder enables backpropagation to both specialize and generalize, as best suits the task.

## 6 *Representation without Reasoning is an Idle Exercise*

This section's title is one of my favorite computer-science quotes from the renowned AI researcher and cognitive scientist Kenneth Forbus [1]. The quote is originally from the mid 1980's, but it is as relevant today as back then, and it applies equally well to AI as to any other area of computer science. Computational representations of system states do not exist in a vacuum: their form, and thereby their utility, depends intimately upon the problem-solving task. Due diligence in crafting representations can pay huge benefits. The (nearly inevitable) hours or days spent getting a bug-free AI system to properly perform a task can quickly balloon into weeks for an ill-conceived representation.

The current star of subsymbolic AI, deep learning, is often touted as *finding its own representation* for problems. Though a bit of an exaggeration, this refers to the ability of backpropagation to tune network weights so that the activation patterns formed in intermediate layers comprise very useful concepts that support an effective mapping from inputs to outputs. As discussed above, these internal representations can strike the proper balance between generalization and specialization.

However, that process strongly relies upon both the human-designed topology of the network and the format of the input data, i.e., the encoding of search states. This document will hopefully help you in making judicious choices for those all-important representations, the ones for which you, not backpropagation, have the primary design responsibility.

## References

[1] K. D. FORBUS, *Qualitative reasoning*, in The Computer Science and Engineering Handbook, Taylor and Francis CRC Press, Abingdon, England, 1997, pp. 715–733.

[2] R. S. SUTTON AND A. G. BARTO, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 2018.