Notes on convolutional attractor nets

Michael C. Mozer

Energy functions and activation dynamics

Consider an attractor net consisting of n units whose state at iteration i is denoted x_i. The state is divided into n_v *visible* units and n_h *hidden* units. An attractor net is characterized by an energy or Lyapunov function (Koiran, 1994):

$$E_i = -\frac{1}{2} x_i W x_i^{\mathrm{T}} - b x_i^{\mathrm{T}} + \sum_j \int_0^{x_{ij}} f^{-1}(\xi) d\xi, \tag{1}$$

where x_i is the network state at some iteration i, j is an index over units in the net, and $f(.)$ is the activation function. With $f \equiv \tanh$, we have

$$E_i = -\frac{1}{2} x_i W x_i^{\mathrm{T}} - b x_i^{\mathrm{T}} + \sum_j x_{ij} \operatorname{arctanh}(x_{ij}) + \frac{1}{2} \ln(1 - x_{ij}^2). \tag{2}$$

The last two terms act as a barrier to prevent activation going outside of $[-1, +1]$: Energy increases rapidly as activations approach ± 1.

Suppose we update the net one neuron at a time—i.e., select a single neuron j at random and update its state according to the standard activation rule,

$$x_{i+1,j} = f\left(x_i w_j + b_j\right), \tag{3}$$

where w_j is the weight vector into unit j and b_j is its bias. Under this asynchronous update rule, we are guaranteed to find a local minimizer of Equation 2 as long as the weights are symmetric and the self-connections are 0. (The proof is trivial, found by solving for $\partial E / \partial x_j = 0$.)

Koiran (1994) proves that if all units in the net are updated *in parallel* according to the standard activation rule, the net is guaranteed to converge to either a fixed point or limit cycle of 2 as long as the weights are symmetric and the self-connections are non-negative. However, the net is not guaranteed to converge at the minimum energy state or even a local minimum of the energy, and it does not iterate strictly downhill in energy.

For a bipartite architecture—where we have layers of neurons, and neurons in a layer are connected only to the layers above and below—updating one *layer* at a time will ensure that a local energy minimum is reached. Thus, this *asynchronous* update dominates synchronous update in terms of the energy of the final state. We can be even more efficient than layerwise updating: if we alternate between updating units in even and odd layers (e.g., first update units in layers 1, 3, 5 and then update units in layers 2, 4), we will reach the same local minimum of energy.

Typically, we will probe the net by *clamping* a subset of visible units and then observing the activation of the remaining visible units. With c a vector of length n indicating whether each unit is clamped or not (1 or 0), the activation rule (Equation 3) becomes:

$$x_{i+1,j} = \begin{cases} f\left(x_i w_j + b_j\right) & \text{if } c_j = 0 \\ x_{i,j} & \text{otherwise.} \end{cases} \tag{4}$$

Clamping units does not change the convergence guarantees mentioned earlier.

Sometimes we wish to break symmetry in a network: there may be multiple minimum energy states consistent with the clamped visible units, and we would like to attain each such solution. To reach different asymptotic states from the same set of clamped values, we have several choices: (1) initialize unclamped units to random initial values; (2) initialize unclamped units to 0, but chose a random layerwise update order; and (3) don't update all units in a layer when a layer is chosen for updating: drop out certain units and keep their values from the previous iteration.

Training an attractor net

By definition, the dynamics of an attractor net lead to convergence—ideally to a fixed point, but possibly to a limit cycle of 2 steps. Given an initial state with certain visible units clamped and all other units reset (typically to 0), we can run the net and it will gradually converge over iterations. With a bipartite architecture and layerwise updating, we will always reach a fixed point. However, our code should still detect when a limit cycle of 2 is reached, in case we switch away from layerwise updating.

As with any recurrent net, we can train an attractor net by unfolding it in time to obtain a feedforward net in which each slab is the full network state at some iteration i. (The full state will consist of the activation of all n units in the attractor net, even if it has a layered structure, both visible and hidden units.)

Let's specify some arbitrary threshold that determines whether the net has converged: if x_i denotes the complete state vector at iteration i, we'll define converge to have (approximately) occurred when

$$||x_i \ - \ x_{i-1}||_\infty < \theta$$

(the largest change in any unit's activation is less than θ). We can define t_* to be the first step at which convergence is reached. Then, for all $i > t_*$, $x_i \approx x_{t_*}$. To simplify notation, we will denote the converged state as x_*.

We can now run the net for t iterations, with $t >> t_*$. This divides the dynamics into a *transient* phase for $1 \leq i < t_*$ and a *stationary* phase for $t_* \leq i \leq t$.

At step t we can define an error which is the difference between the target state y and the actual state x_t:

$$E_t = ||(x_t - y)M||^2 \tag{5}$$

where M is a diagonal array of dimension $n \times n$ for which $m_{jj} = 1$ only if j is a visible neuron, and y has arbitrary values for non-visible elements.

We will use backpropagation to compute $\partial E_t / \partial W$ where W is the weight matrix mapping upward, i.e., the net input to iteration $i + 1$ is

$$z_{i+1} = x_i W + b$$

and we will assume a tanh activation function:

$$x_{i+1} = \tanh(z_{i+1}).$$

Because the weights have been copied when we unfold, we need to consider the contribution of the weights at each iteration $i < t$ to E_t:

$$\frac{\partial E_t}{\partial W} = \sum_{i=1}^{t} \frac{\partial E}{\partial W_i}, \tag{6}$$

where W_i is the instantiation of the weights at iteration i:

$$\frac{\partial E_t}{\partial W_i} = x_{i-1}^{\mathrm{T}} \frac{\partial E_t}{\partial z_i}. \tag{7}$$

(This says that the derivative of a particular weight w_{ab} instantiated at iteration i is the product of the derivative of the error with respect to unit b at iteration i and the activation of unit a at iteration $i-1$.)

So now the problem is to compute $\partial E_t/\partial z_i$. We know that for $i = t$, we have

$$\frac{\partial E_t}{\partial z_t} = 2(x_t - y)MD_t, \tag{8}$$

where D_t is the diagonal derivative matrix with element (j,k) defined as:

$$d_{tjk} \equiv \frac{\partial x_t}{\partial z_t} = \begin{cases} 0 & \text{if } j \neq k \text{ or } c_j = 1 \\ 1 - x_{tj}^2 & \text{otherwise.} \end{cases} \tag{9}$$

(The derivative for a unit goes to zero as its activation approaches the limits of $+1$ or -1.) Then we can define derivatives recursively:

$$\frac{\partial E_t}{\partial z_{i-1}} = \frac{\partial E_t}{\partial z_i}\frac{\partial z_i}{\partial x_{i-1}}\frac{\partial x_{i-1}}{\partial z_{i-1}} = \frac{\partial E_t}{\partial z_i}W^{\mathrm{T}}D_i.$$

With $i > t_*$, we can replace x_i with x_*, leading to the general formulation:

$$\frac{\partial E_t}{\partial z_i} = \frac{\partial E_t}{\partial z_t}\left[W^{\mathrm{T}}D_*\right]^{t-i} \tag{10}$$

Combining Equations 6, 7, and 10, we can express the error derivative over the stable period as a power series:

$$\frac{\partial E_t}{\partial W} = x_*^{\mathrm{T}}\frac{\partial E_t}{\partial z_t}\sum_{i=c}^{t}\left[W^{\mathrm{T}}D_*\right]^{t-i} \tag{11}$$

As $t \to \infty$, we can ignore the settling time t_* and the summation becomes a power series,

$$\Omega_\infty = \lim_{t\to\infty}\sum_{k=0}^{t-1}\left[W^{\mathrm{T}}D_*\right]^k, \tag{12}$$

which can be computed iteratively via:

$$\Omega_i = \begin{cases} I & \text{for } i = 1 \\ I + \Omega_{i-1}W^{\mathrm{T}}D_* & \text{for } i > 1. \end{cases} \tag{13}$$

Note that in our case of symmetric weights, $W^{\mathrm{T}} = W$.

The Ω_i series will converge, $\lim_{i\to\infty}|\Omega_i - \Omega_{i-1}| = 0$, only if the spectral radius of the Jacobian, $W^{\mathrm{T}}D_*$, is less than 1. (All eigenvalues are between -1 and $+1$.) In my current implementation, I toss out any trial in which the gradient explodes. It happens only infrequently.

Finding Ω_∞ by iterating Equation 13 and combining Equations 8 and 11, we obtain

$$\frac{\partial E_t}{W} = x_*^{\mathrm{T}}(x_* - y)MD_*\Omega_\infty. \tag{14}$$

This formulation gets a lot of hype as the Pineda/Almeida algorithm, but it's nothing more than BPTT with a net that has reached a stationary state and stays there infinitely long.

Note that because $\frac{\partial E_t}{\partial W} = \frac{\partial E_{t-1}}{\partial W}$ as $t \to \infty$, the derivative direction doesn't change whether we inject error only at the final step t or at every step i, $t_* \leq i \leq t$.

By analogy to Equation 14, the derivative of the bias weights, which are just like any other weight with a fixed input of 1, is:

$$\frac{\partial E_t}{b} = (x_* - y) M D_* \Omega_\infty. \tag{15}$$

Equations 14 and 15 give us the error derivative *for the stationary phase* of the net. It's also the error derivative for the transient phase if error is injected *only at the final time step*. The reason for this latter claim is that assuming Equation 13 converges, the gradient is squashed out by the time it reaches the transient phase. However, if we inject error at every step, training the current state to the target—equivalent to TD(1)—then we must also compute an error for the transient phase of the activation dynamics. This error can be computed using pytorch's auto-differentiation over iterations $1 - t_*$. If examples are trained in batches, it will be important to identify t_*—the point of stability—for each example individually.

I believe there is room here for doing temporal difference learning, particularly TD(0). The stationary phase is already doing TD(0) by virtue of the fact the state is no longer changing. But we can do TD(0) during the transient phase *as long as treat each step $i < t_*$ as a TD series*, with an external error injected at i and a TD training signal produced for the earlier time steps.

Symmetry constraints

In an attractor net, if there is a connection from some unit α to some unit β, denoted $w_{\alpha,\beta}$, we must have symmetry:

$$w_{\alpha,\beta} = w_{\beta,\alpha}.$$

In a fully connected pool of units, this constraint simply implies symmetry of the full weight matrix W.

Our challenge is to determine a set of symmetry constraints on an architecture with units arranged in a lattice and with restricted connectivity, specifically a convolutional layer. We address two cases: (2) we have a single convolutional layer with internal recurrent connectivity that follows the local connectivity constraints of a convolutional net. (1) we have two convolutional layers which have connections between layers but no connectivity within a layer.

Case 1: Interconnectivity within a recurrent convolutional layer

Our convolutional layer is a lattice of dimensions $n_f \times n_x \times n_y$, where n_f is the number of filters, n_x is the horizontal spatial extent, and n_y is the vertical spatial extent. We will build a recurrent network which, when unfolded in time, will map from this layer to a layer with the same dimensionality via a convolutional operator with patch size $n_f \times (2n_r + 1) \times (2n_r + 1)$, where n_r is the radius of the patch. The recurrent mapping will be determined by a set of weights, $W = \{w_{efab}\}$, where w_{efab} refers to the connection feeding into unit (x, y, f) from unit $(x + a, y + b, e)$, for every x and y on the lattice. W has dimensionality $n_f \times n_f \times (2n_r + 1) \times (2n_r + 1)$. To simplify notation without any loss of generality, we will assume $x = y = 0$.

If w_{efab} specifies the connection $(a, b, e) \to (0, 0, f)$, then we require this weight to be symmetric with the connection $(0, 0, f) \to (a, b, e)$, or by normalizing with a constant offset, $(-a, -b, f) \to (0, 0, e)$, i.e.,

$$w_{e,f,a,b} = w_{f,e,-a,-b}.$$

Our challenge is how to efficiently enforce this constraint, or how to express the underlying parameters and how they map to W.

Remember, we also need to enforce the self-connection constraint $w_{f,f,0,0} \geq 0$ for all f.

An approach using numpy

Starting with a simple example, suppose we have a two-dimensional weight array Z, and we want to ensure that Z satisfies $z_{a,b} = z_{-a,-b}$. (I am using the indexing scheme from the previous section where the indices range from $-n_r$ to $+n_r$.) The constraint specifies that the $(+,+)$ quadrant mirrors the $(-,-)$ quadrant, and the $(+,-)$ quadrant mirrors the $(-,+)$ quadrant. One way to guarantee that Z has the desired property is to define $Z0$ as an unconstrained array, and then to define Z using the `numpy` expression:

```
Z = (Z0 + Z0[::-1,::-1])/2.0
```

For W, we want to do something similar: define an unconstrained $W0$ and then sum the $W_{f,e,a,b}$ and $W_{e,f,-a,-b}$ to symmetrize:

```
W = (W0 + np.swapaxes(W0[:,:,::-1,::-1],0,1))/2.0
```

Here is an example of how this works:

```
>>> W0=np.random.random([2,2,3,3])*2.0-1.0
array([[[[-0.50707783,  0.88680495,  0.99713658],
         [-0.61545061,  0.23018124, -0.44635288],
         [ 0.34563401,  0.64314837, -0.39077811]],

        [[ 0.51212478, -0.11878099,  0.70312206],
         [ 0.13932228,  0.34936946,  0.72935749],
         [ 0.16512938,  0.25552298,  0.41256676]]],


       [[[-0.78205692,  0.55854117,  0.62183324],
         [-0.16264486,  0.22754876, -0.74926053],
         [-0.8973259 ,  0.97267335,  0.71312325]],

        [[-0.82456146,  0.29473738, -0.66902053],
         [ 0.92127547, -0.64778669,  0.03054832],
         [-0.97555256,  0.62103307,  0.42283626]]]])

>>> W=(W0+np.swapaxes(W0[:,:,::-1,::-1],0,1))/2.0
array([[[[-0.44892797,  0.76497666,  0.67138529],
         [-0.53090174,  0.23018124, -0.53090174],
         [ 0.67138529,  0.76497666, -0.44892797]],

        [[ 0.61262402,  0.42694618, -0.09710192],
         [-0.30496912,  0.28845911,  0.28335632],
         [ 0.39348131,  0.40703208, -0.18474508]]],


       [[[-0.18474508,  0.40703208,  0.39348131],
         [ 0.28335632,  0.28845911, -0.30496912],
         [-0.09710192,  0.42694618,  0.61262402]],

        [[-0.2008626 ,  0.45788523, -0.82228654],
         [ 0.47591189, -0.64778669,  0.47591189],
         [-0.82228654,  0.45788523, -0.2008626 ]]]])
```

Note that the `W[0,1,:,:]` and `W[1,0,:,:]` matrices are up-down, left-right flipped versions of one another, and note that `W[0,0,:,:]` and `W[1,1,:,:]` are symmetric around both diagonals. The only constraint that is not guaranteed here is that the self-connections `W[0,0,1,1]` and `W[1,1,1,1]` are not guaranteed to be positive. In `numpy`, you can do this with

```
for i in range(0,2):
    W[i,i,1,1] = np.abs(W[i,i,1,1])
```

Case 2: Interconnectivity between convolutional layers

This case covers the bipartite nets we are currently building, with a series of layers within the net and connectivity between layers but not within a layer.

Consider two convolutional layers each with a horizontal spatial extent of n_x and a vertical spatial extent of n_y. The lower layer has n_f filters and the upper layer has n_g filters. We will build a recurrent network that connects the lower layer to the upper and vice versa. Let $W^{\text{up}} = \{w^{\text{up}}_{fgab}\}$ be the connectivity from from the lower layer to the upper layer, where w^{up}_{fgab} refers to the connection feeding into unit (x, y, g) in the upper layer from unit $(x + a, y + b, f)$ in the lower layer. We can also define the weight matrix from the upper to lower layer: $W^{\text{down}} = \{w^{\text{down}}_{gfab}\}$ be the connectivity from from the upper to lower layer, where w^{down}_{gfab} refers to the connection feeding into unit (x, y, f) in the lower layer from unit $(x + a, y + b, g)$ in the upper layer.

W^{up} is a fully unconstrained matrix of dimensions $n_f \times n_g \times (2n_r + 1) \times (2n_r + 1)$, where n_r is the radius of a patch. W^{down} has dimensions $n_g \times n_f \times (2n_r + 1) \times (2n_r + 1)$ and is fully specified by W^{up}:

$$w^{\text{down}}_{g,f,-a,-b} = w^{\text{up}}_{f,g,a,b}$$

As Michael Iuzzolino has determined, this constraint is easily implemented by transposing the first two dimensions of W^{up} and flipping the indices of the last two dimensions to obtain W^{down}.