Notes on convolutional attractor nets

Michael C. Mozer

1 Training an attractor net

An attractor net is a recurrent net that will converge to a fixed point (or a limit cycle of 2 steps). The network state is decomposed into two sets of units: *visible* units and *hidden* units, denoted x_{vis} and x_{hid}, respectively. Let $\hat{y}$ denote x_{vis}. **N.B.** The state x consists of the activation of all units in the complete attractor net ($x = \hat{y} \cup x_{hid}$), even if the network has a layered structure.

To train any recurrent net, we can unfold the net in time to obtain a feedforward net in which each slab is the full network state at some iteration i (Figure **??**).

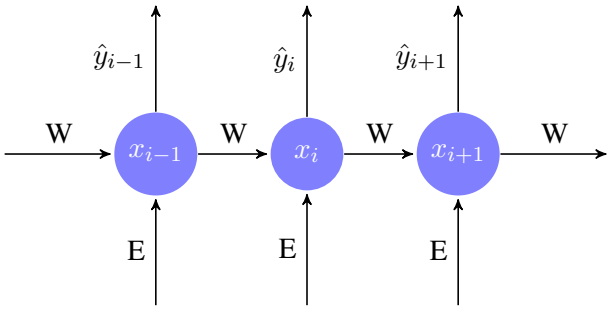

Figure 1: Unrolled Network

The network state, x, updates at each iteration i following the rule:

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

where W is a constrained weight matrix that maps between iterative states and whose constraints ensure attractor dynamics. Specifically, we impose symmetry constraints on W, detailed below in §**??**. Further, we use $\tanh$ such that $x \in [-1, 1]$ as necessary for attractor dynamics (Koiran, 1994).

When the network is run, its state will change at each iteration, but gradually converge on the fixed points of the system. We specify some arbitrary threshold, θ, that determines whether the net has converged. x_i denotes the complete state vector at iteration i. We define convergence to have (approximately) occurred when

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

We look at the difference between steps of size 2 to ensure limit cycles of 2 are captured, and $||\cdot||_\infty$ ensures the largest change in any unit's activation is less than θ. We define c to be the first step as which convergence is reached. Then, for all $i > c$, $x_i \approx x_c$. We will denote the converged state, x_c, as x_*.

After state convergence at $i = c$, we can now run the net for t iterations, with $t >> c$. This divides the network's training dynamics into a *transient* phase for $1 \leq i < c$ and a *stationary* phase for $c \leq i \leq t$. For the *transient* phase, we calculate the updates to W utilizing a TD(λ) approach, discussed below. Now, we will discuss how we calculate updates to W for the *stationary* phase.

Stationary Phase Training

We will use backpropagation to compute $\partial E_t / \partial W$ for the stationary phase of training; i.e., the network's state has stabilized at step c and $t >> c$. First, we define the error of the network, E_t. At step t we can define an error which is the difference between the target state y and the actual state x_t:

$$E_t = ||(x_t - y)\ \mathrm{diag}(m)||^2 \tag{1}$$

where m is a mask that has 1's for the elements of x that are visible, and y has arbitrary values for non-visible elements.

The effect of the weights, W, on the error, E_j (Equation **??**) at iteration j is given by:

$$\frac{\partial E_j}{\partial W} = \frac{\partial E_j}{\partial x_j} \frac{\partial x_j}{\partial z_j} \frac{\partial z_j}{\partial W}$$

Condensing $\frac{\partial E_j}{\partial x_j} \frac{\partial x_j}{\partial z_j} = \frac{\partial E_j}{\partial z_j}$, and given $\frac{\partial z_j}{\partial W} = x_{j-1}$, we obtain the compact form

$$\frac{\partial E_j}{\partial W} = x_{j-1}^{\mathrm{T}} \frac{\partial E_j}{\partial z_j}$$

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

$$\frac{\partial E_t}{\partial W} = \sum_{j=1}^{t} x_{j-1}^{\mathrm{T}} \frac{\partial E_t}{\partial z_j}, \tag{2}$$

This tells us that the derivative of the total error at timestep t with respect to W_{ab} is the sum of products at iteration j between the derivative of the error with respect to unit b at iteration j and the activation of unit a at iteration $j - 1$.

Now we must compute $\partial E_t / \partial z_j$. We know that for $j = t$, we have

$$\frac{\partial E_t}{\partial z_t} = 2(x_t - y)\ \mathrm{diag}(1 - x_t^2)\ \mathrm{diag}(m), \tag{3}$$

where $\mathrm{diag}(1 - x_t^2)$ is the derivative of $\tanh(x_t)$; that is, the diagonal matrix is the derivative of a layer's output with respect to its net input.

We can now define derivatives recursively:

$$\begin{aligned}
\frac{\partial E_t}{\partial z_{j-1}} &= \frac{\partial E_t}{\partial z_j} \frac{\partial z_j}{\partial z_{j-1}} \\
&= \frac{\partial E_t}{\partial z_j} \frac{\partial z_j}{\partial x_{j-1}} \frac{\partial x_{j-1}}{\partial z_{j-1}} \\
&= \frac{\partial E_t}{\partial z_j} W^{\mathrm{T}} \frac{\partial x_{j-1}}{\partial z_{j-1}} \\
&= \frac{\partial E_t}{\partial z_j} W^{\mathrm{T}}\ \mathrm{diag}(1 - x_{j-1}^2)
\end{aligned}$$

Given that the net has converged $\forall j \geq c$, we replace x_j with x_*, leading to the general formulation:

$$\frac{\partial E_t}{\partial z_j} = \frac{\partial E_t}{\partial z_t} \left[W^{\mathrm{T}} \operatorname{diag}(1 - x_*^2) \right]^{t-j} \qquad c \leq j < t \tag{4}$$

Substituting Equation **??** into Equation **??** and readjusting indices, we can express the error derivative over the stable period, $\partial E_t / \partial W$, as a power series:

$$\begin{aligned}
\frac{\partial E_t}{\partial W} &= \sum_{j=c}^{t} x_*^{\mathrm{T}} \frac{\partial E_t}{\partial z_j} \\
\frac{\partial E_t}{\partial W} &= \sum_{j=c}^{t} x_*^{\mathrm{T}} \frac{\partial E_t}{\partial z_t} \left[W^{\mathrm{T}} \operatorname{diag}(1 - x_*^2) \right]^{t-j} \\
\frac{\partial E_t}{\partial W} &= x_*^{\mathrm{T}} \frac{\partial E_t}{\partial z_t} \sum_{j=c}^{t} \left[W^{\mathrm{T}} \operatorname{diag}(1 - x_*^2) \right]^{t-j}
\end{aligned} \tag{5}$$

As $t \to \infty$, we can ignore the settling time c and the summation, $\sum_{j=c}^{t} \left[W^{\mathrm{T}} \operatorname{diag}(1 - x_*^2) \right]^{t-j}$, becomes a power series, Ω_∞,

$$\Omega_\infty = \lim_{i \to \infty} \sum_{k=0}^{i-1} \left[W^{\mathrm{T}} \operatorname{diag}(1 - x_*^2) \right]^{k}, \tag{6}$$

which can be computed iteratively via:

$$\Omega_i = \begin{cases} 0 & \text{for } i = 0 \\ I + \Omega_{i-1} W^{\mathrm{T}} \operatorname{diag}(1 - x_*^2) & \text{for } i > 0. \end{cases} \tag{7}$$

This formulation gets a lot of hype as the Pineda/Almeida algorithm, referred to at Recurrent Back Propagation (RBP), but RBP nothing more than BPTT with a net that has reached a stationary state.

Alternatively, an analytical solution for Ω_∞ may exist. Let $R = W^{\mathrm{T}} \operatorname{diag}(1 - x_*^2)$. Given sufficient properties of R, the following analytical solution to Ω_∞ is:

$$\Omega_\infty = \left[I - W^{\mathrm{T}} \operatorname{diag}(1 - x_*^2) \right]^{-1} \tag{8}$$

Specifically, the existence of Equation **??** depends on $\rho(R) < 1$, where $\rho(R)$ is the spectral radius of R. Appendix **??** points to a reference containing potential proofs on symmetric matrices and spectral radius constraints.

Plugging in Equation **??** and the result of Equation **??** into Equation **??** yields:

$$\begin{aligned}
\frac{\partial E_t}{\partial W} &= x_*^{\mathrm{T}} \frac{\partial E_t}{\partial z_t} \sum_{j=c}^{t} \left[W^{\mathrm{T}} \operatorname{diag}(1 - x_*^2) \right]^{t-j} \\
\frac{\partial E_t}{\partial W} &= x_*^{\mathrm{T}} 2(x_* - y) \operatorname{diag}(1 - x_*^2) \operatorname{diag}(m) \, \Omega_\infty \\
\frac{\partial E_t}{\partial W} &= 2 x_*^{\mathrm{T}} (x_* - y) \operatorname{diag}(1 - x_*^2) \operatorname{diag}(m) \, \Omega_\infty
\end{aligned} \tag{9}$$

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 j for $c \leq j \leq t$.

Equation **??** gives us the error derivative *for the stationary phase* of the net.

Transient Phase Training

Equation **??** also provides 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 because as Equation **??** 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 $i \in \{1, ..., c\}$. If examples are trained in batches, it will be important to identify c—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 $j < c$ as a TD series*, with an external error injected at j and a TD training signal produced for the earlier time steps.

2 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}.

A Proofs

All material from this section referenced from Zdeněk Dvořák's, "Spectral radius, symmetric and positive matrices", 2016.

Consider a square matrix, W. The geometric series of W, $\sum_{i=0}^{\infty} W^i$ is guaranteed to converge to $[I - W]^{-1}$ when $\rho(W) < 1$, where $\rho(W)$ is the spectral radius of W.

Definition A.1. Spectral Radius The *spectral radius* of a square matrix W is

$$\rho(W) = \max\{|\lambda| : \lambda \text{ is an eigenvalue of } W\}$$

For an $n \times n$ matrix W, let $||W|| = \max\{|W_{ij} : 1 \leq i, j \leq n\}$.

Lemma A.1. If $\rho(W) < 1$, then

$$\lim_{n\to\infty} ||W^n|| = 0.$$

If $\rho(W) > 1$, then

$$\lim_{n\to\infty} ||W^n|| = \infty.$$

Lemma A.2. If W is a *symmetric real matrix*, then $\max\left\{x^T W x : ||x|| = 1\right\}$ is the largest eigenvalue of W.

Proof. Let $W = QDQ^T$ for a diagonal matrix D and an orthogonal matrix Q. Note that W and D have the same eigenvalues and that $||Qx|| = ||x||$ for every x, and since Q is regular, it follows that $\max\left\{x^T W x : ||x|| = 1\right\} = \max\left\{x^T D x : ||x|| = 1\right\}$. Therefore, it suffices to show that $\max\left\{x^T W x : ||x|| = 1\right\}$ is the largest eigenvalue of D. Let $d_1 \geq d_2 \geq \ldots \geq d_n$ be the diagonal entries of D, which are also its eigenvalues. Then $x^T D x = \sum_{i=1}^{n} d_i x_i^2 \leq d_1 \sum_{i=1}^{n} x_i^2 = d_1||x|| = d1$ for every x such that $||x_1|| = 1$, and $e_1^T D e_1 = d_1$. □