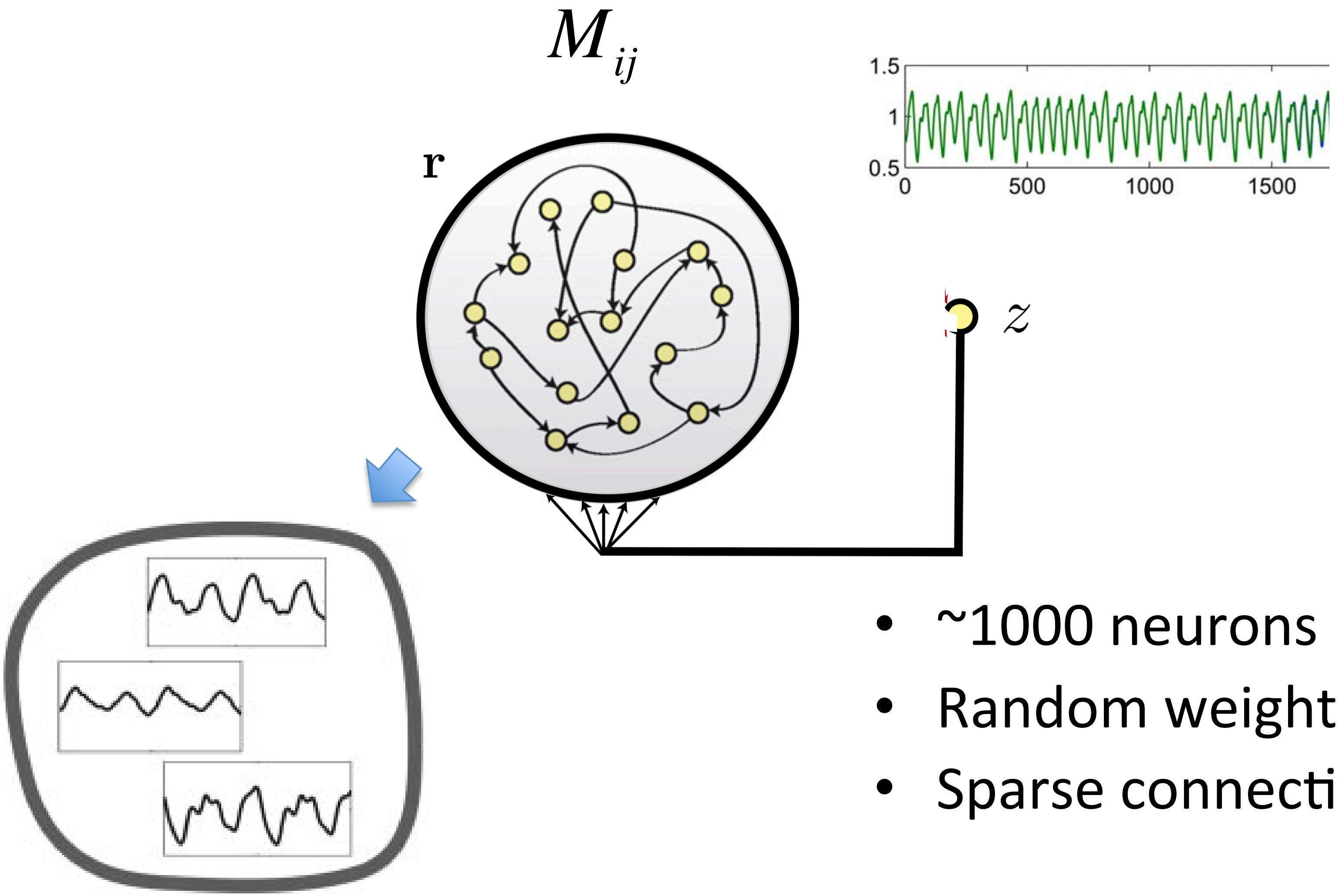
Recurrent Network Decoding

Problems with Recurrent Networks

- Training takes forever, doesn’t always work
- Can get suboptimal, unstable solutions
- Tend towards seizures, or quiet fixed points
- Stuck with small neural networks

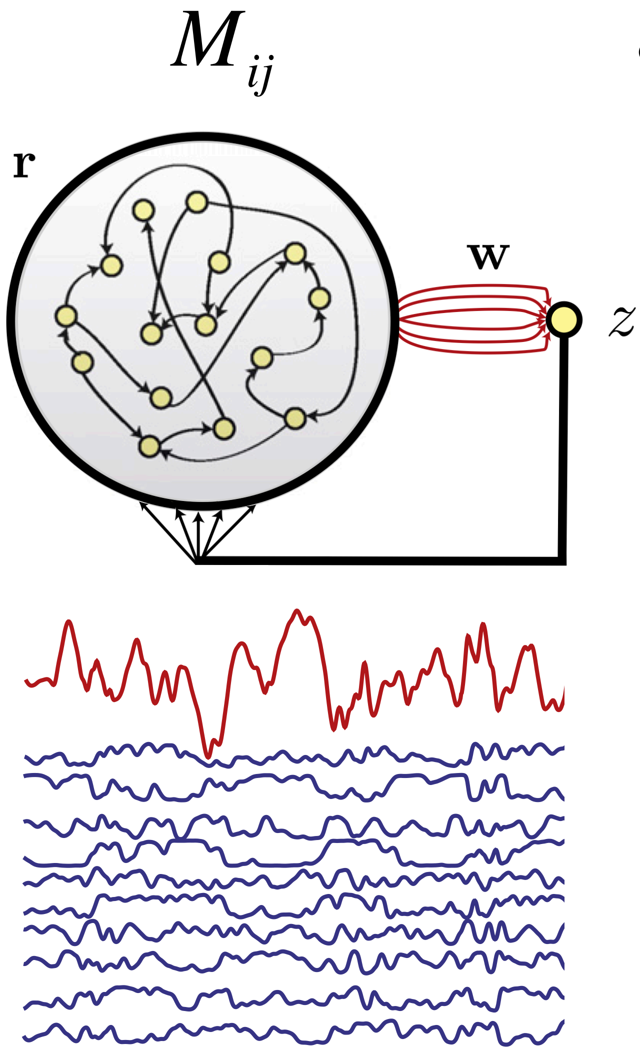
Echo-State Network



- ~1000 neurons
- Random weights Mij
- Sparse connectivity

(Jaeger and Haas, 2004)

Spontaneously Chaotic Network



- Assemble random network

N Neurons

M_{ij} Weight of connection from i to j is normally distributed with :

Zero mean:

$$\mu = 0$$

Variance:

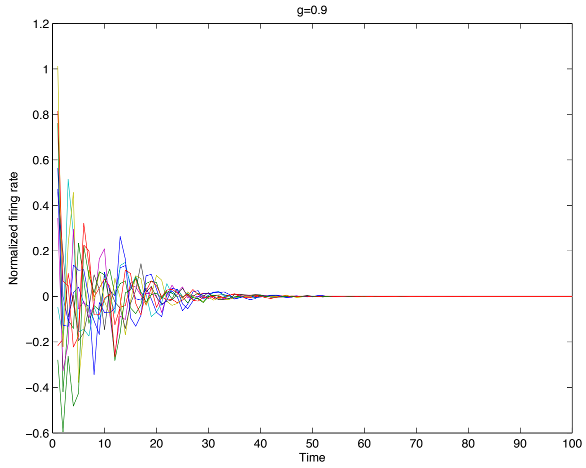
$$\sigma^2 = \frac{g^2}{N}$$

$g > 1 \longrightarrow$ Chaotic firing

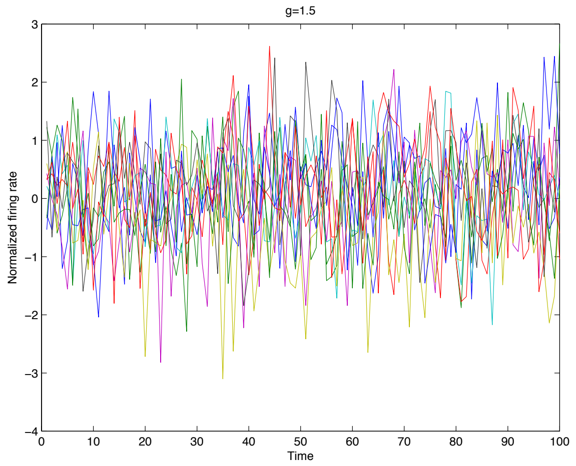
(Sussillo et al., 2009, Sompolinsky 1988)

Chaotic Regime

g=0.9



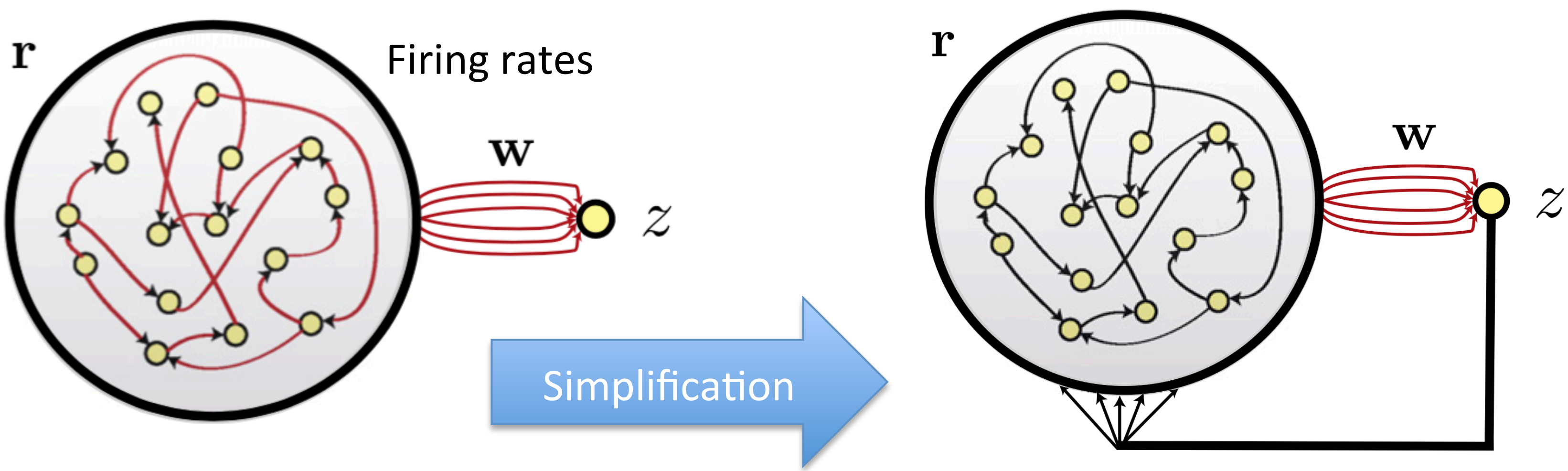
g=1.5



Training Chaotic Networks - FORCE

- FORCE Learning

$$\vec{r}(t)$$



Readout $\vec{z}(t) = \vec{w}^T \vec{r}(t) = w_1 r_1(t) + w_2 r_2(t) \ldots$

Goal: $\vec{z}(t) \longrightarrow f(t)$ (Arbitrary pattern generation)

(Sussillo et al., 2009)

FORCE Learning Rule

1. Create random, recurrent network

$$y_j(t+\Delta t) = M_{ij}\vec{r}_i(t) = M_{ij}\tanh(y_i(t))$$

2. Construct random output readout

$$z(t) = \vec{w}^T\vec{r}(t)$$

3. Simulate timestep and calculate error

$$e(t) = z(t) - f(t)$$

4. Adjust weights based on the error

$$w(t+\Delta t) = \vec{w}(t) - e(t)\vec{P}(t)\vec{r}(t)$$

(Sussillo et al., 2009)

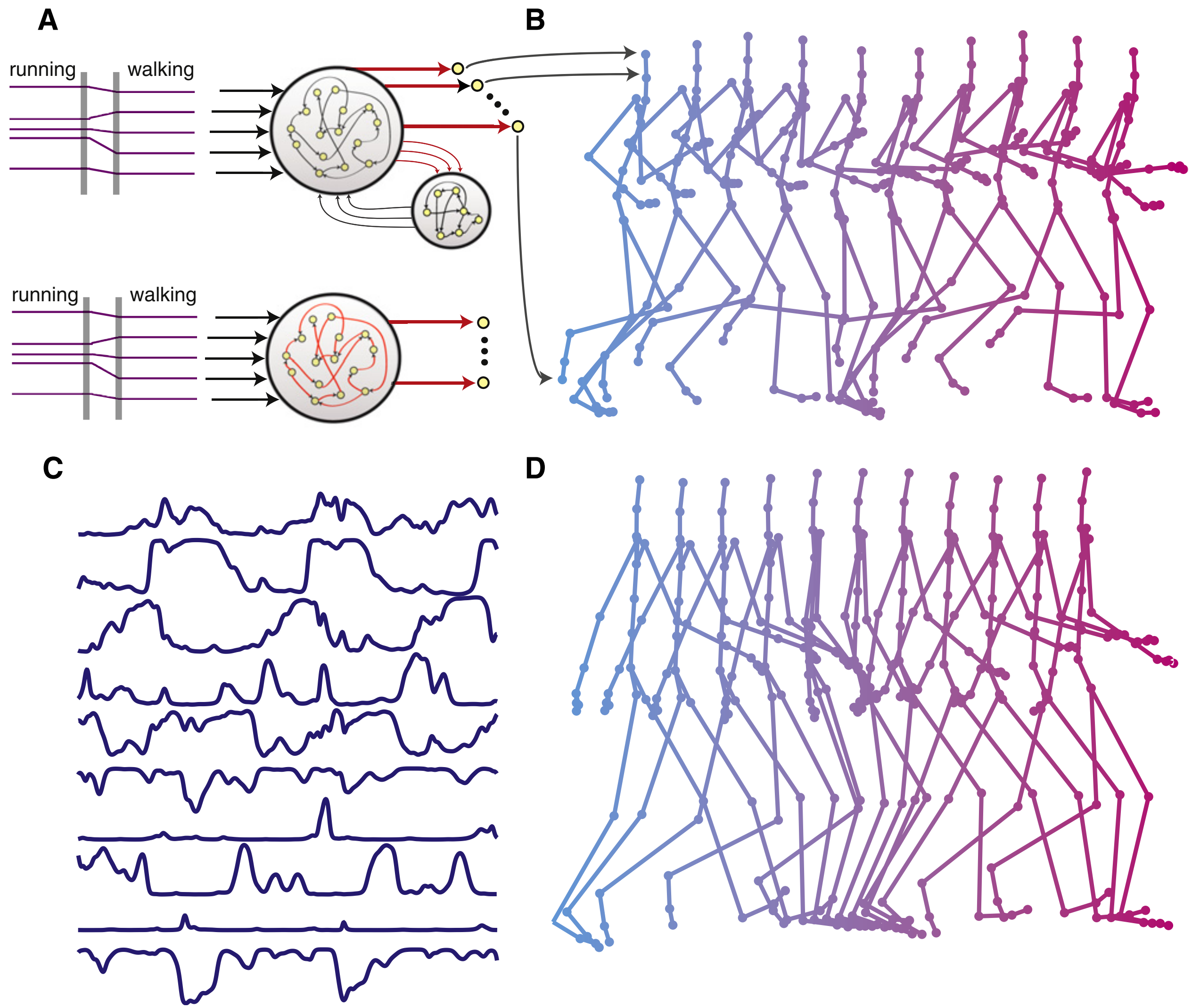
Learning Rule – On Wikipedia

- Recursive Least Squares (Adaptive Filter) is a faster version of gradient descent
- Jumps right to the best guess of the minimum
- Solve for change in weights that will minimize the squared error
- **P**(t) is N x N matrix of learning rates
- Calculate it recursively on each timestep

$$\vec{P}(t+\Delta t)=\vec{P}(t)-\frac{\vec{P}(t)\vec{r}(t)\vec{r}^{T}(t)\vec{P}(t)}{1+\vec{r}^{T}(t)\vec{P}(t)\vec{r}(t)}$$

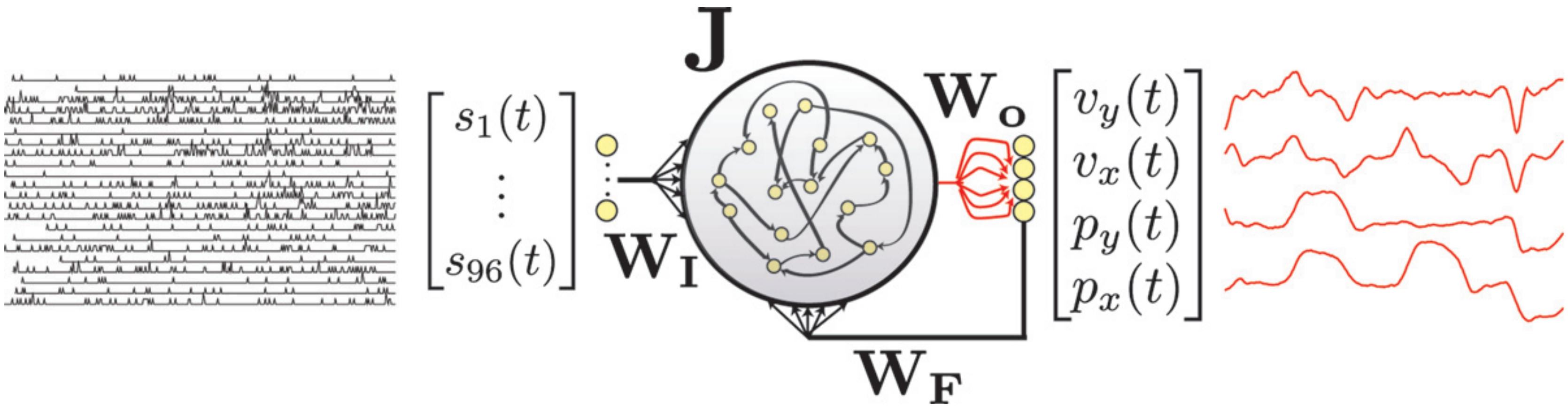
(Sussillo et al., 2009)

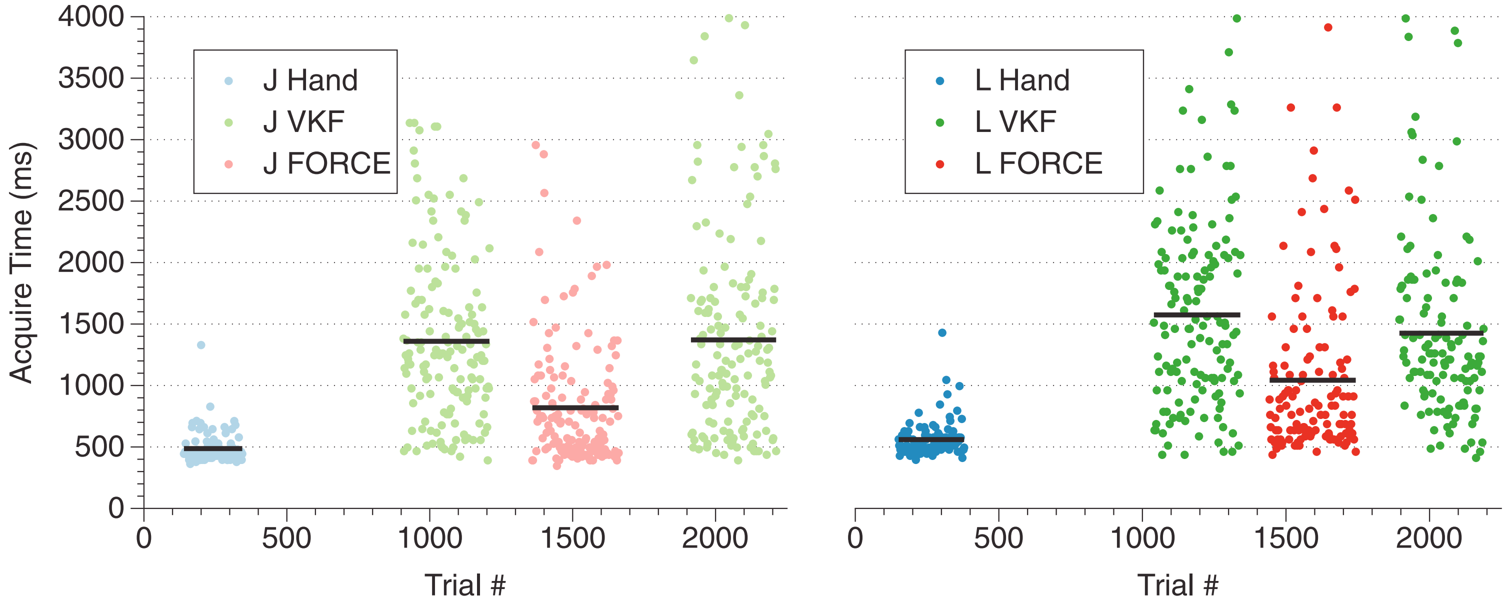
Neural Network Capabilities



(Sussillo et al., 2009)

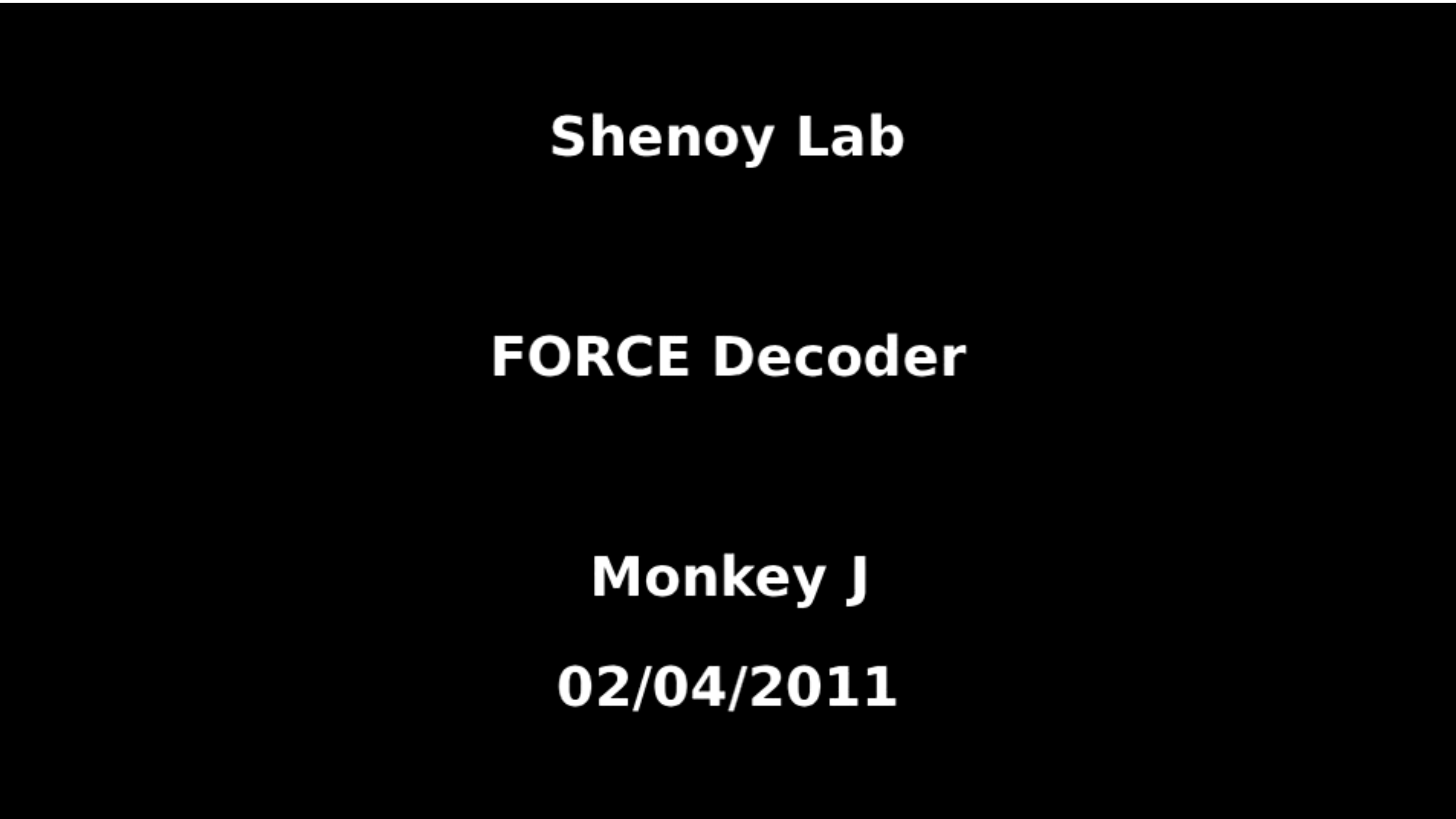
Decoding Neurons with Neurons



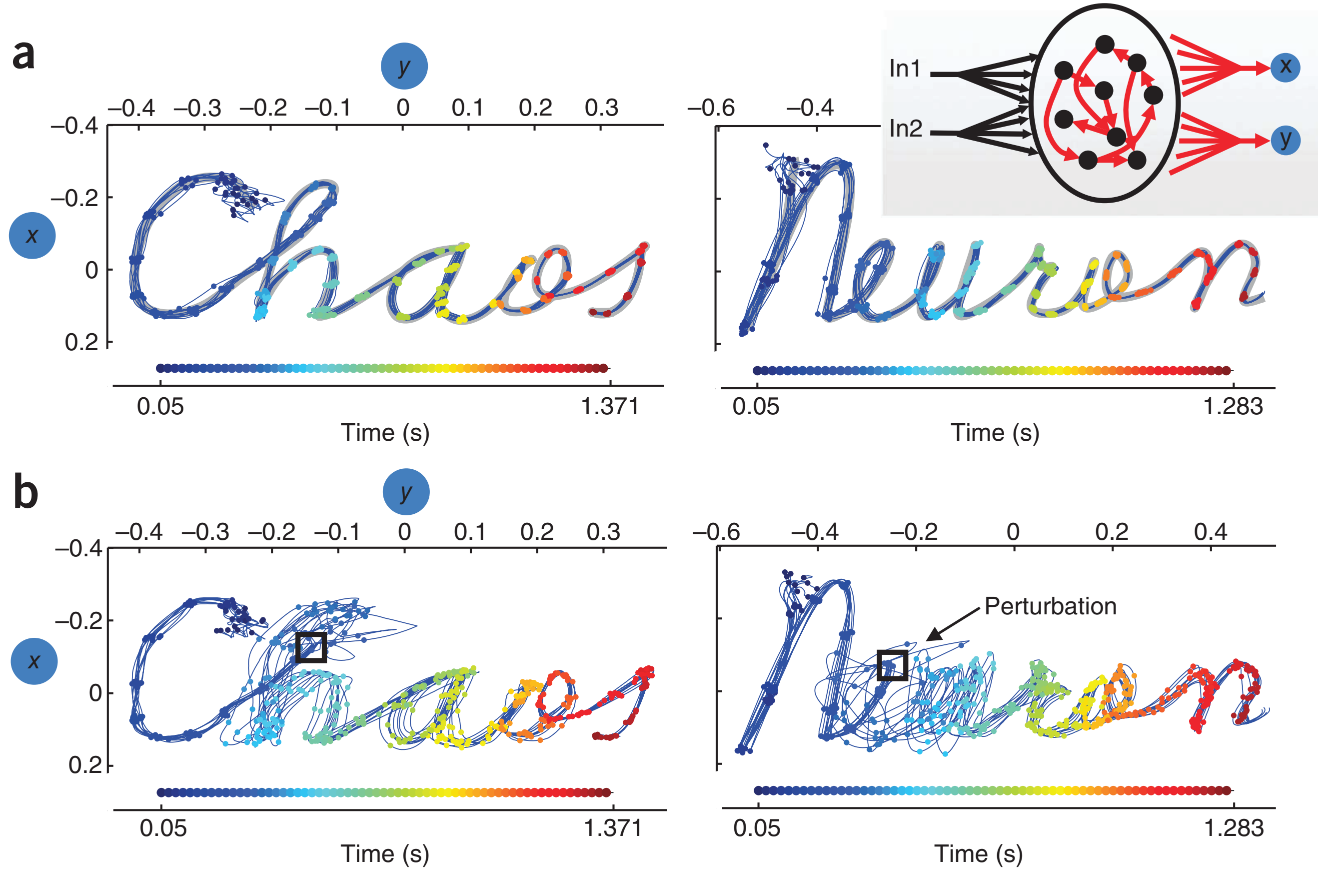


(Sussillo et al., 2012)

Movie



Perturbations



(Laje, Buonomano, 2013)

Now have biological architecture…

…but no biological learning mechanism

Recursive Least Squares

- When you need a filter as fast as possible
- Don't train it to reduce the *mean* squared error (i. e. average error over time)
- Train it every time step to minimize the error *right now*
- Converges faster to an optimum filter at cost of higher computational complexity

Recursive Least Squares

- Minimize (non-mean) least squared error

$$e(i) = f(i) - \vec{w}_n^T \vec{y}(i)$$

$$E(n) = \frac{1}{2}\sum_{i=0}^{n} \lambda^{n-i} \left| e(i)^2 \right|$$

Desired output at time i

Filter at time n

Neural Firing Rates

Forgetting factor,
small means past is not
important, usually ~0.98

Derivation of RLS

- Take derivative of error wrt **w**, set to 0

1)

$$E(n) = \frac{1}{2}\sum_{i=0}^{n} \lambda^{n-i} \left| e(i)^2 \right| \qquad e(i) = f(i) - \sum_{k=0}^{p} w_n(k) y(i-k)$$

2)

$$\frac{\partial E(n)}{\partial w_n(k)} = \sum_{i=0}^{n} \lambda^{n-i} e(i) \frac{\partial e(i)}{\partial w_n(k)}$$

(Skipping a bit)

Intuitive Final Answer

- Multiplying an optimum gain by the error in the current estimate using the filter from the last timestep

$$\vec{w}_n - \vec{w}_{n-1} = \Delta\vec{w}_n = \vec{g}(n)\Big(f(n) - \vec{w}_{n-1}^T\vec{y}(n)\Big)$$

Optimum gain for least squared error

What you wanted

What the last timestep's filter gives you with current incoming data

In Matlab

```
% sim, so x(t) and r(t) are created.
   x = (1.0-dt)*x + M*(r*dt) + wf*(z*dt);
   r = tanh(x);
   z = wo'*r;

   if mod(ti, learn_every) == 0
   % update inverse correlation matrix
  %NOTE lambda = 1.0
   k = P*r;
    c = 1.0/(1.0 + r'*P*r);
   P = P - k*(k'*c);

   % update the error for the linear readout
   e = z-ft(ti);

   % update the output weights
   dw = -e*k*c;
   wo = wo + dw;
```

$$\vec{g}(n) = \frac{\lambda^{-1}\vec{P}(n-1)\vec{y}(n)}{1+\lambda^{-1}\vec{y}(n)^T\vec{P}(n-1)\vec{y}(n)}$$

$$\vec{g}(n) = \frac{k}{1+\vec{y}(n)^T\vec{P}(n-1)\vec{y}(n)}$$

$$\vec{g}(n) = kc$$

(Sussillo et al. 2012)

In Matlab

```
% sim, so x(t) and r(t) are created.
  x = (1.0-dt)*x + M*(r*dt) + wf*(z*dt);
  r = tanh(x);
  z = wo'*r;

  if mod(ti, learn_every) == 0
  % update inverse correlation matrix
 %NOTE lambda = 1.0
  k = P*r;
  c = 1.0/(1.0 + r'*P*r);
  P = P - k*(k'*c);

  % update the error for the linear readout
  e = z-ft(ti);

  % update the output weights
  dw = -e*k*c;
  wo = wo + dw;
```

$$\vec{P}(n) = \vec{P}(n-1) - \vec{g}(n)\vec{y}^T(n)\vec{P}(n-1)$$

$$\vec{P}(n) = \vec{P}(n-1) - kc\vec{y}^T(n)\vec{P}(n-1)$$

$$\vec{P}(n) = \vec{P}(n-1) - kck'$$

$$\vec{g}(n) = kc$$

(Sussillo et al. 2012)