AT2 – Final Project: Coherent Patterns of Activity from Chaotic Neural Networks

Younesse Kaddar

From "Generating Coherent Patterns of Activity from Chaotic Neural Networks" by D. Sussillo and L.F.Abbott (2009)

IPYTHON NOTEBOOK **DOCUMENTATION** **GITHUB** **REPORT**

1. Networks Architectures

We will consider (sparse) **recurrent neural networks** of N_G neurons, that can be thought of as simple models actual biological neural networks:

- the neurons are inter-connected by synapses (which may be excitatory or inhibitory)
- each neuron's membrane potential follows a leaky integrate-and-fire-looking differential equation (which depends network architecture details)
- the network is a linear combination of the neurons' activites weighted by readout vector(s) (the latter will be modified during the learning process)
- there may be external inputs and/or the network output may be fed back to the network (directly or via a feedback network).

Network Architecture A

The first network architecture is the following recurrent generator network:

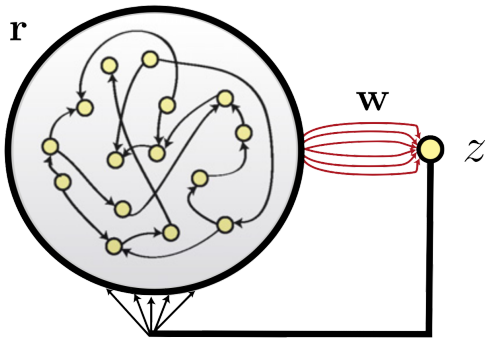


Figure 1.A - Network architecture A (image courtesy of David Sussilo)

where

- the large cicled network is the generator network
- r is the neurons' firing rates
- w is the readout vector: its weights are only ones prone to be modified during training (indicated by the red color, contrary to the black connections, that remain unchanged thoughout learning)

The only feedback provided to the generator network comes from the the readout unit.

Here, the membrane potential of neuron $i \in [1, N_G]$ is given by:

$$\tau \dot{x}_i = -x_i + g_{GG} \sum_{j=1}^{N_G} J_{ij}^{GG} \underbrace{\tanh(x_j)}_{\stackrel{\text{def}}{=} r_j} + g_{G_z} J_i^{G_z} z$$

- x_i is the neuron's membrane potential
- $\tau = 10$ ms is the time constant of the units dynamics
- J^{GG} is the synaptic weight/strength matrix of the generator network
- g_{GG} is the scaling factor of the synaptic weight matrix of the generator network
- J^{Gz} is the readout unit weight matrix (applied when the readout unit is fed back in the generator network)
- g_{Gz} is the scaling factor of the feedback loop:
  increasing the feedback connections results in the network chaotic activity allowing the learning process.

When it comes to the implementation:

- Each element of J^{GG} is set to 0 with probability $1 - p_{GG}$. The nonzero elements thereof are drawn from a centered Gaussian distribution with variance $1/p_{GG}$
- The elements of J^{Gz} are drawn from a uniform distribution between -1 and 1
- Nonzero elements of $\mathbf{w}$ are set initially either to zero or to values generated by a centered Gaussian distribution with variance $1/(p_z N)$.
- The network integration time step dt and the time span between modifications of the readout weights Δt may not be equal:

$$\Delta t \geq dt$$

2. First-Order Reduced and Controlled Error (FORCE) algorithm

---

Sussilo and Abott's FORCE supervised learning algorithm makes it possible for chaotic recurrent network output to match a large array of pre-determined activity patterns.

It is different from other classic learning algorithms in that, instead of trying to nullify the error as qickly as possible, it:

- reduces the output error drastically right away
- but then keeps **maintaining it small** (instead of nullifying it), and rather focuses on decreasing the number of modifications needed to keep the error small

Feeding back an output close but different to the desired one has the following advantages, among others:

- it avoids over-fitting and stability issues (indeed, and over-fitted chaotic nework may have its activity diverge as soon as a non-zero error is fed back), making the whole procedure suited for chaotic neural networks, in which are highly intersting, insofar as many models of spontaneously active neural circuits exhibit chaotic behaviors.
- it enable us to modify synaptic strength without restricting ourselves to specific neurons (like ouput ones), which makes it all the more realistic, from a biological standpoint.

How and why does FORCE learning work?

How?

We initialize a matrix $P(t)$ - the estimate of the inverse of the network rates correlation matrix plus a regularization term - as follows:

$$P(0) = \frac{1}{\alpha}\mathbf{I}$$

where α is the inverse of a learning rate: so a sensible value of α

- depends on the target function
- ought to be chosen such that $\alpha << N$

Indeed:

- if α is too small, the learning is so fast it can cause unstability issues
- if α is too large, the learning is so slow that it may end up failing

Then, at each time-step Δt:

1. $$P(t) \longleftarrow \left(\mathbf{I} - \frac{1}{1 + \langle \mathbf{r}(t), P(t-\Delta t)\mathbf{r}(t) \rangle} \underbrace{P(t-\Delta t)\mathbf{r}(t)\mathbf{r}(t)^\intercal}_{\text{outer product}} \right) P(t-\Delta t)$$

2. One compute the error *before the readout vector update*:

$$e_{-}(t) = \mathbf{w}(t-\Delta t)^\intercal \mathbf{r}(t) - f(t)$$

3. The readout vector is updated:

$$\mathbf{w}(t) \longleftarrow \mathbf{w}(t-\Delta t) - e_{-}(t) P(t) \mathbf{r}(t)$$

If $\alpha << N_G$, then it can be shown that the error is remains small from the first update on, and $\mathbf{w}$ converges to a constant value, all thoughout P converging toward the pseudo-inverse of $\sum_t \mathbf{r}(t)\mathbf{r}(t)^\intercal + \frac{1}{\alpha}\mathbf{I}$ and the error being reduced.

Why?

Essentially, FORCE relies on a **regularized version** of the recursive least-squares (RLS) algorithm (that is, the *online/incremental* verison of the well-known least-squares algorithm).

Basically, what one attempts to do is an *online regression* (but with the contraints mentioned above), where we try to find a predictor $\hat{f}$ such that:

$$f(t) = \hat{f}\left(\mathbf{x}(t)\right)$$

of the form

$$\hat{f}\left(\mathbf{x}(t)\right) = \sum_{i=1}^{N_G} \tanh(\mathbf{x}_i(t))\mathbf{w}_i = \langle \underbrace{\tanh(\mathbf{x}(t))}_{=\mathbf{r}(t)}, \mathbf{w} \rangle$$

So in a *batch* fashion, where we consider several observations for several consecutive timesteps (the $\mathbf{r}(t)$ are the lines of a matrix $\mathbf{R} = \tanh(\mathbf{X})$):

$$\hat{f}\left(\mathbf{X}\right) = \tanh(\mathbf{X})\mathbf{w}$$

In an *online* fashion: at each step t, one has the input/desired output pair:

$$\big(\mathbf{x}(t),\ f(t)\big)$$

The squared prediction error thereof is:

$$e\_(t) = \Big(f(t) - \underbrace{\tanh(\mathbf{x}(t+1))^\intercal \mathbf{w}(t-\Delta t)}_{=\langle \mathbf{r}(t+1), \mathbf{w}(t-\Delta t)\rangle}\Big)^2$$

In a *batch* way, given n input/desired output pairs:

$$\big(\mathbf{x}(t),\ f(t)\big), \dots, \big(\mathbf{x}(t+(n-1)\Delta t),\ f(t+(n-1)\Delta t)\big)$$

the squared error is (where $\mathbf{t} \stackrel{\text{def}}{=} (t, t+\Delta t, \cdots, t+(n-1)\Delta t)^\intercal$):

$$\begin{aligned} e_{batch} &= \frac{1}{2n}\|f(\mathbf{t}) - \hat{f}\,(\mathbf{X})\|_2^2 = \frac{1}{2n}\|f(\mathbf{t}) - \mathbf{R}\mathbf{w}\|_2^2 \\ &= \frac{1}{2n}\sum_{i=0}^{n}\Big(f(t+i\Delta t) - \langle \mathbf{r}(t+i\Delta t), \mathbf{w}\rangle\Big)^2 \end{aligned}$$

It is convex with respect to $\mathbf{w}$, so to minimize it we set the gradient to zero:

$$\mathbf{0} = \nabla_{\mathbf{w}} e_{batch} = -\frac{1}{n}\sum_{i=1}^{n}\mathbf{r}(t+i\Delta t)\Big(f(t+i\Delta t) - \langle \mathbf{r}(t+i\Delta t), \mathbf{w}^*\rangle\Big)$$

i.e.

$$\underbrace{\left(\sum_{i=1}^{n}\mathbf{r}(t+i\Delta t)\mathbf{r}(t+i\Delta t)^\intercal\right)}_{\stackrel{\text{def}}{=} A}\mathbf{w}^* = \underbrace{\sum_{i=1}^{N}\mathbf{r}(t+i\Delta t)f(t+i\Delta t)}_{\stackrel{\text{def}}{=} b}$$

Therefore:

$$\mathbf{w}^* = A^\sharp b$$

where $A^\sharp$ is the pseudo-inverse of A.

So we are beginning to see where do this $(A + \frac{1}{\alpha}\mathbf{I})^\sharp$ come from, in the FORCE algorithm.

It's even more blatant in the *online* version of the least squares algorithm: we see that A and b can be computed *incrementally* at each time-iteration:

$$A(t+\Delta t) = A(t) + \mathbf{r}(t+\Delta t)\mathbf{r}(t+\Delta t)^\intercal$$
$$b(t+\Delta t) = b(t) + \mathbf{r}(t+\Delta t)f(t+\Delta t)$$

Then, $\mathbf{w}^*$ can be estimated as:

$$\mathbf{w}(t+\Delta t) = \left(A^{(t+\Delta t)}\right)^\sharp b^{(t+\Delta t)}$$

The key point is that the pseudo-inverse $\left(A^{(t+\Delta t)}\right)^\sharp$ can be estimated with resort to the

> ***Sherman-Morrison lemma*** *(provided $A(0)$ is non-zero, which is what happens in our case):*
>
> $$\left(A + \mathbf{r}(t+\Delta t)\mathbf{r}(t+\Delta t)^\intercal\right)^\sharp = A^\sharp - \frac{A^\sharp \mathbf{r}(t+\Delta t)\mathbf{r}(t+\Delta t)^\intercal A^\sharp}{1 + \mathbf{r}(t+\Delta t)^\intercal A^\sharp \mathbf{r}(t+\Delta t)}$$

which is what gives the P's update-rule.

Implementation and Results

Our implementation makes use of *object-oriented programming*:

- it can be found here as a **python package**
    - to import it with `pip` :

```
!pip install git+https://github.com/youqad/Chaotic_Neural_Networks.gi
from chaotic_neural_networks import utils, networkA
```

- the documentation is here

The package structure is as follows:

```
 Chaotic_Neural_Networks
|
└──chaotic_neural_networks
|   |   __init__.py
|   |   utils.py
|   |   networkA.py
|
```

```
└──docs
   |   ...
```

- `utils.py` contains utility functions, among which target function such as a sum of sinusoids, a triangle-wave, etc...
- `networkA.py` is the module related to the first architecture: the class `NetworkA` is a way to instantiate such a network (which can be fully parametrized with the optional arguments). The three most important methods are:
    - `error` which computes the average train/test error of the network
    - `step` , which executes one step of length `dt` of the network dynamics
    - `FORCE_sequence` , which plots (returns a matplotlib figure to be precise) a full training sequence of the FORCE algorithm: showing the evolution of the network ouput(s), a handful of neurons membrane potential, and the time-derivative of the readout vector $\dot{\mathbf{w}}$ before training (spontaneous activity), throughout training, and after training (test phase).

For instance: the following code

```
import matplotlib.pyplot as plt
from chaotic_neural_networks import utils, networkA

t_max = 2400 # in ms: duration of each phase (pre-training, training, and test)

# Target function f: Sum of sinusoids
network1 = networkA.NetworkA(f=utils.periodic)

network1.FORCE_sequence(2400*3)

plt.show()
```

outputs:

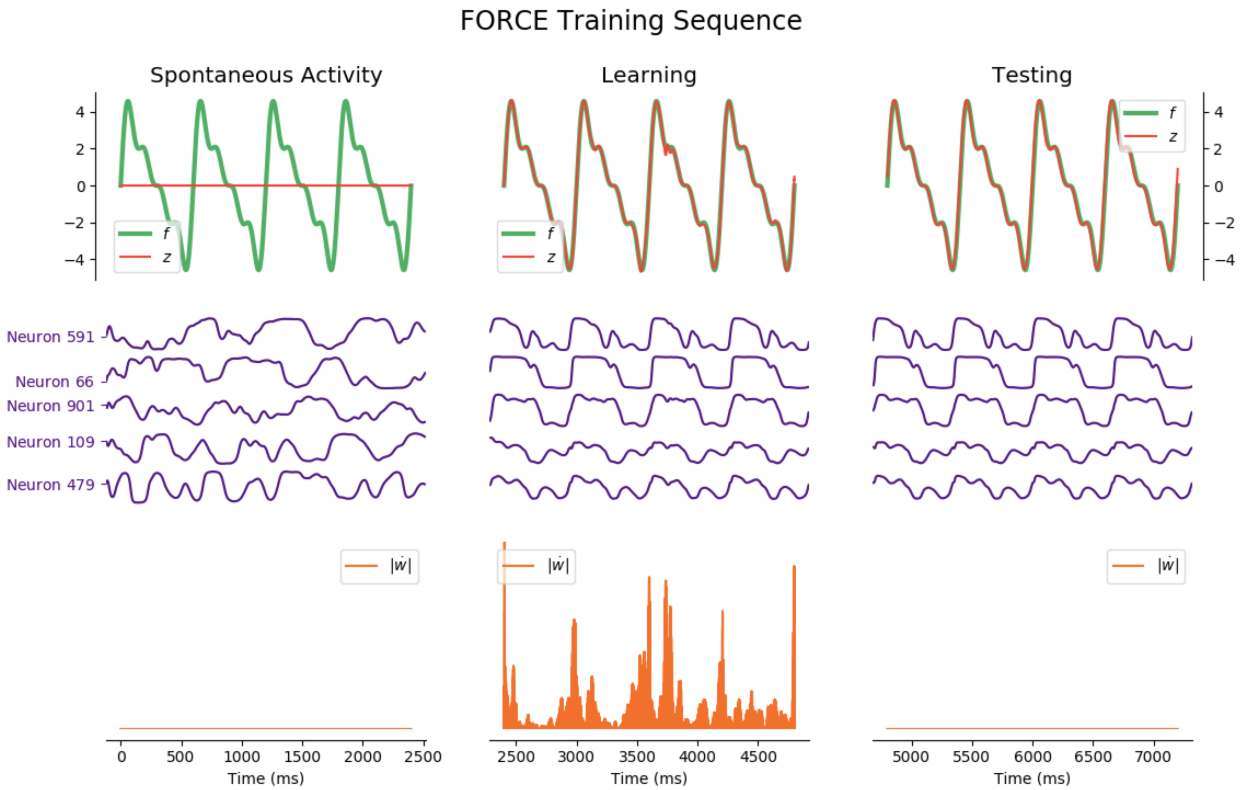


Figure 3.A.1 - FORCE training sequence (similar to Sussilo's figure 2), for a sum-of-sinusoids target function

The code to generate the following training sequence plots is in the `training_sequence_plots.py` file of the github repository (here).

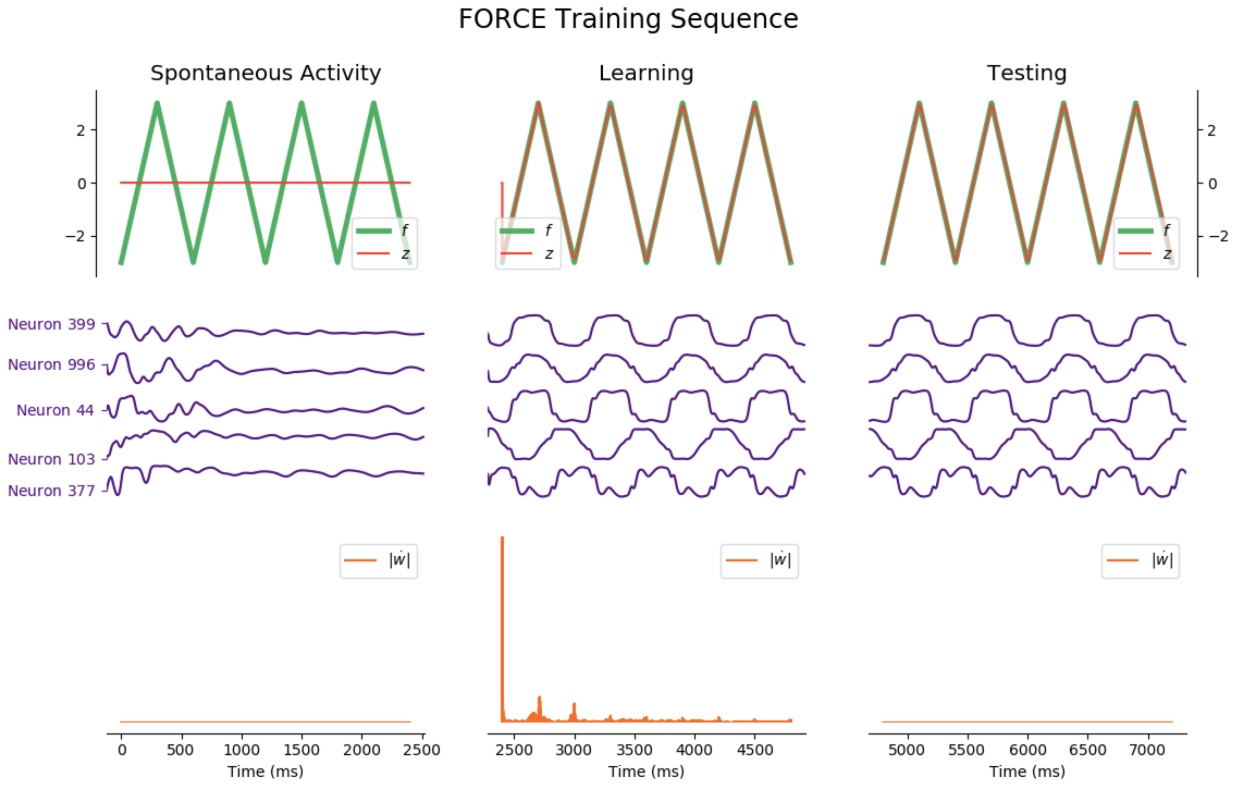


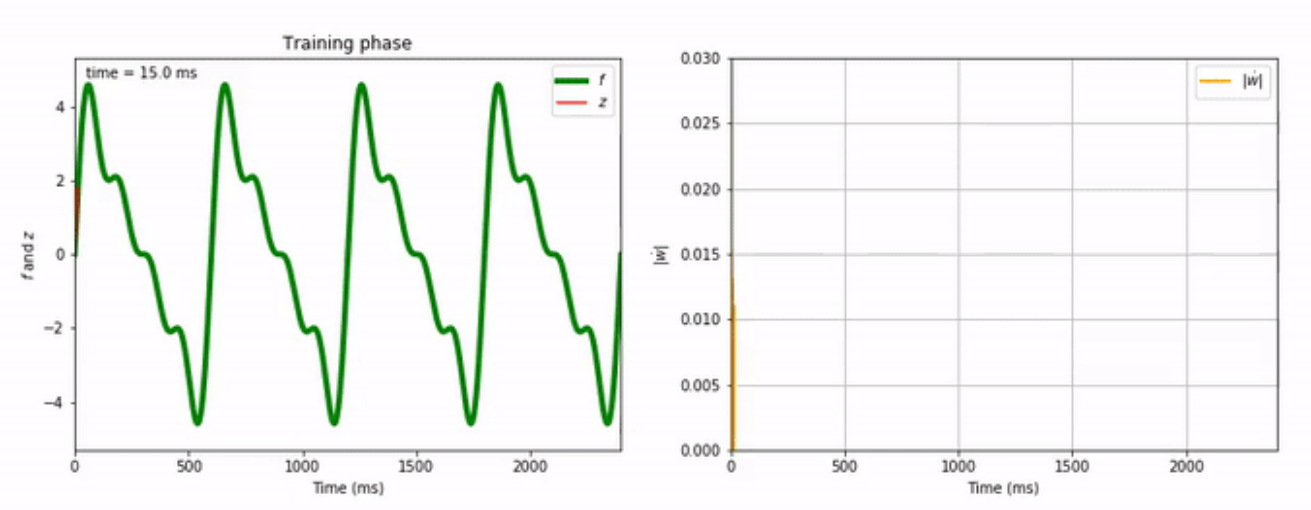
Figure 3.A.2 - FORCE training sequence, for a triangle-wave target function

> ***Average Train Error:*** 0.016
> ***Average Test Error:*** 0.055

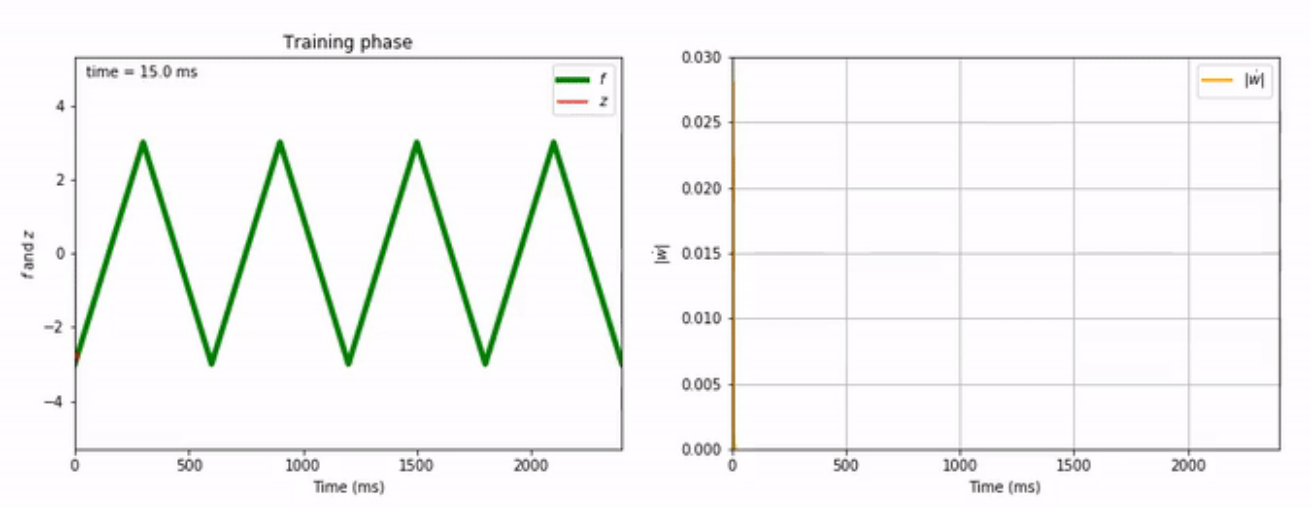
Considering the decrease of $|\dot{\mathbf{w}}|$, we see that the learning process is far quicker for the triangle-wave function than for the sum-of-sinusoids one (which is not surprising, as the triangle-wave is piecewise linear): for the former, it takes between 3 and 4 periods for the learning to be complete.

Here are animated gifs of the FORCE learning phase for these target functions (the code to generate these animations is in the Jupyter notebook, the function used is `utils.animation_training`):

- Evolution of FORCE learning for a sum of four sinusoids as target:



- Evolution of FORCE learning for triangle-wave as target:



Now, let us investigate more complicated patterns: we reiterate the FORCE learning procedure, but this time for a significantly more complicated target:

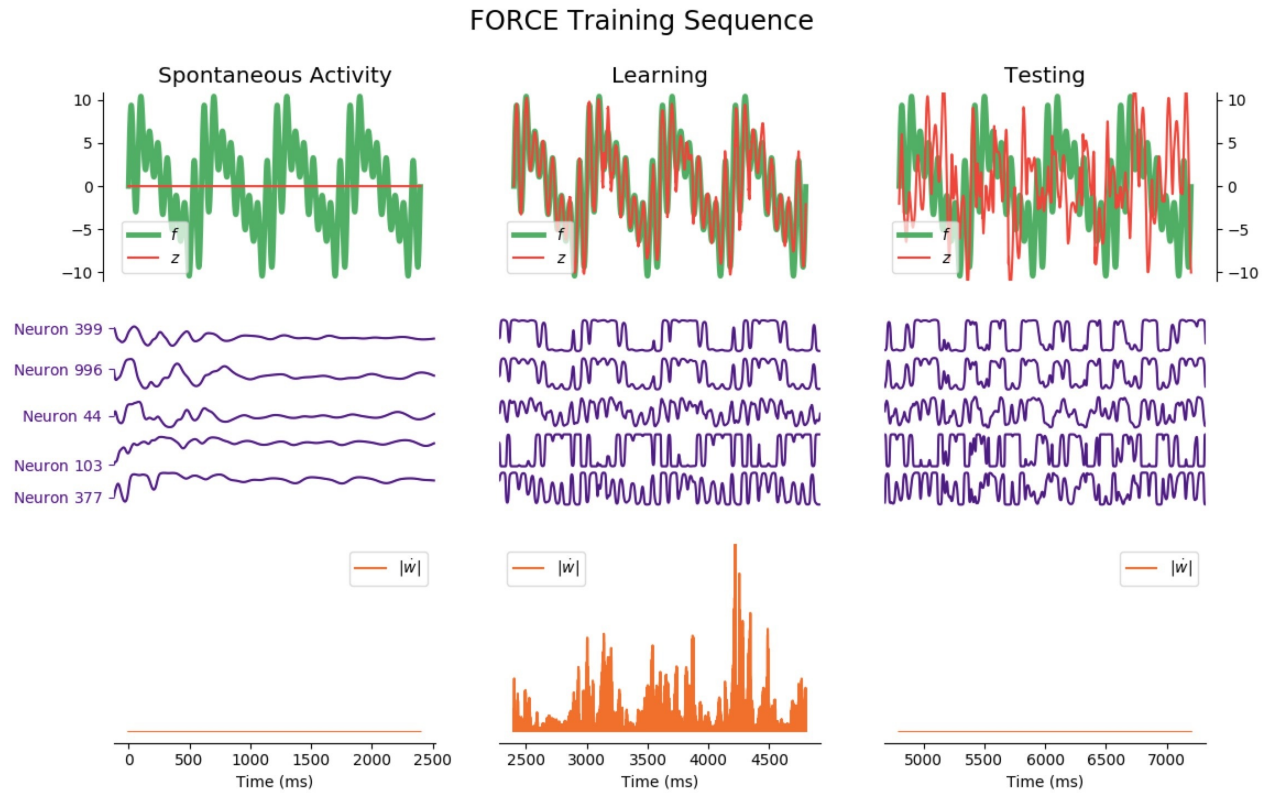


Figure 3.A.3.1. - FORCE training sequence: each phase lasting 2400 ms, for a complicated sum-of-sinusoids target function

> ***Average Train Error:*** 0.725
> ***Average Test Error:*** 5.730

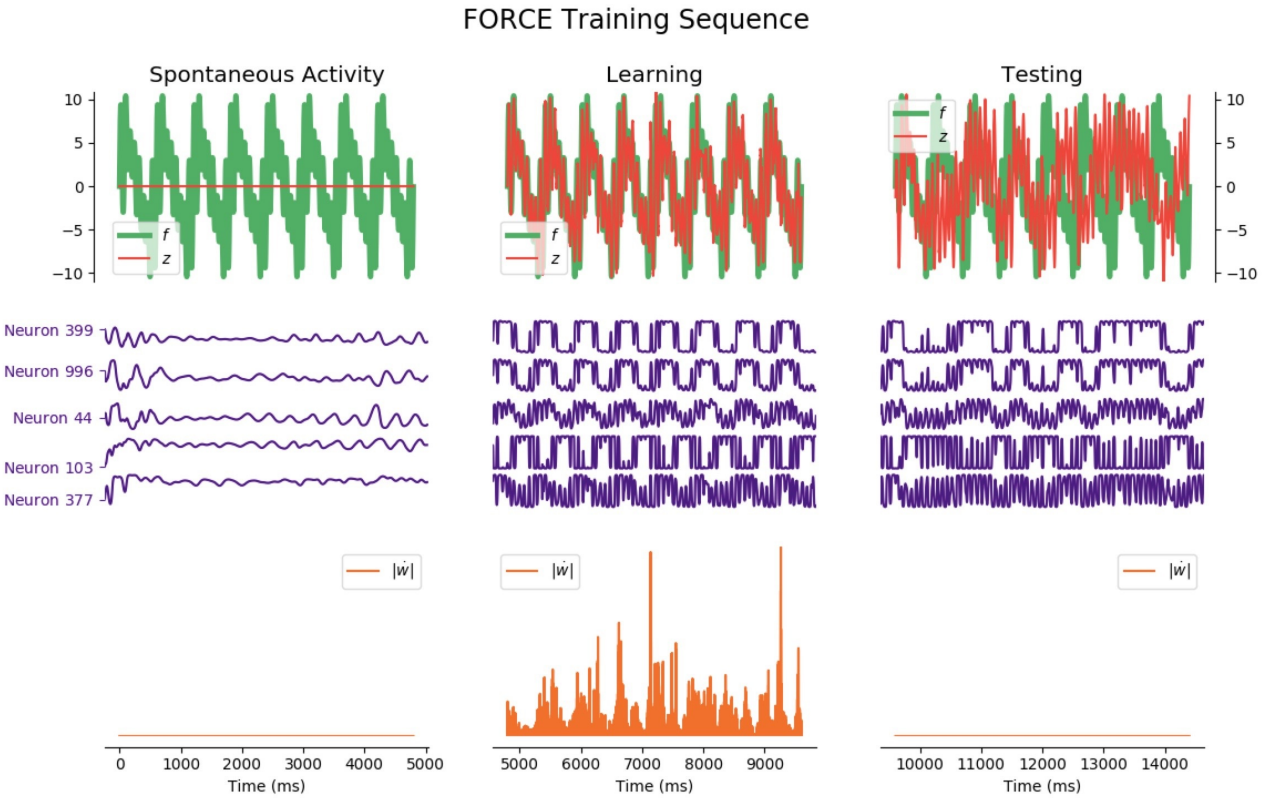


Figure 3.A.3.2. - FORCE training sequence: each phase lasting 4800 ms, for a complicated sum-of-sinusoids target function

> ***Average Train Error:*** 0.880
> ***Average Test Error:*** 5.789

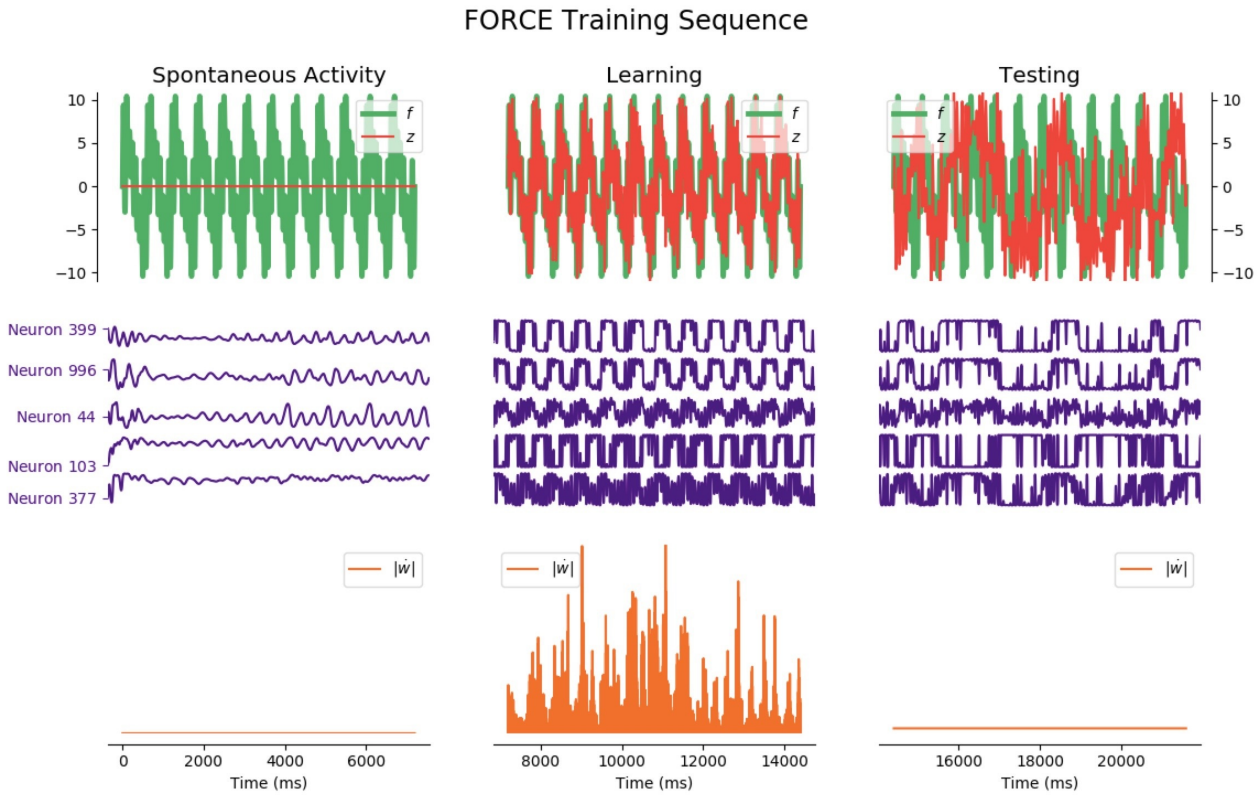


Figure 3.A.3.3. - FORCE training sequence: each phase lasting 7200 ms, for a complicated sum-of-sinusoids target function

> ***Average Train Error:*** 0.934
> ***Average Test Error:*** 5.993

Figure 3.A.3.4. - FORCE training sequence: each phase lasting 12000 ms, for a complicated sum-of-sinusoids target function

> ***Average Train Error:*** 0.893
> ***Average Test Error:*** 5.887

It appears that the high variability of this target makes matters more complicated: when it comes to learning, the network is still reasonably effective right away, but it has poor testing accuracy for phases lasting less than 12 seconds (the train and test errors even increase until 7 seconds, and then decreases for longer phases).

The network may also have several outputs, corresponding to several readout units. Here are some instances of 2 and 3 output networks:

Two simultaneous outputs

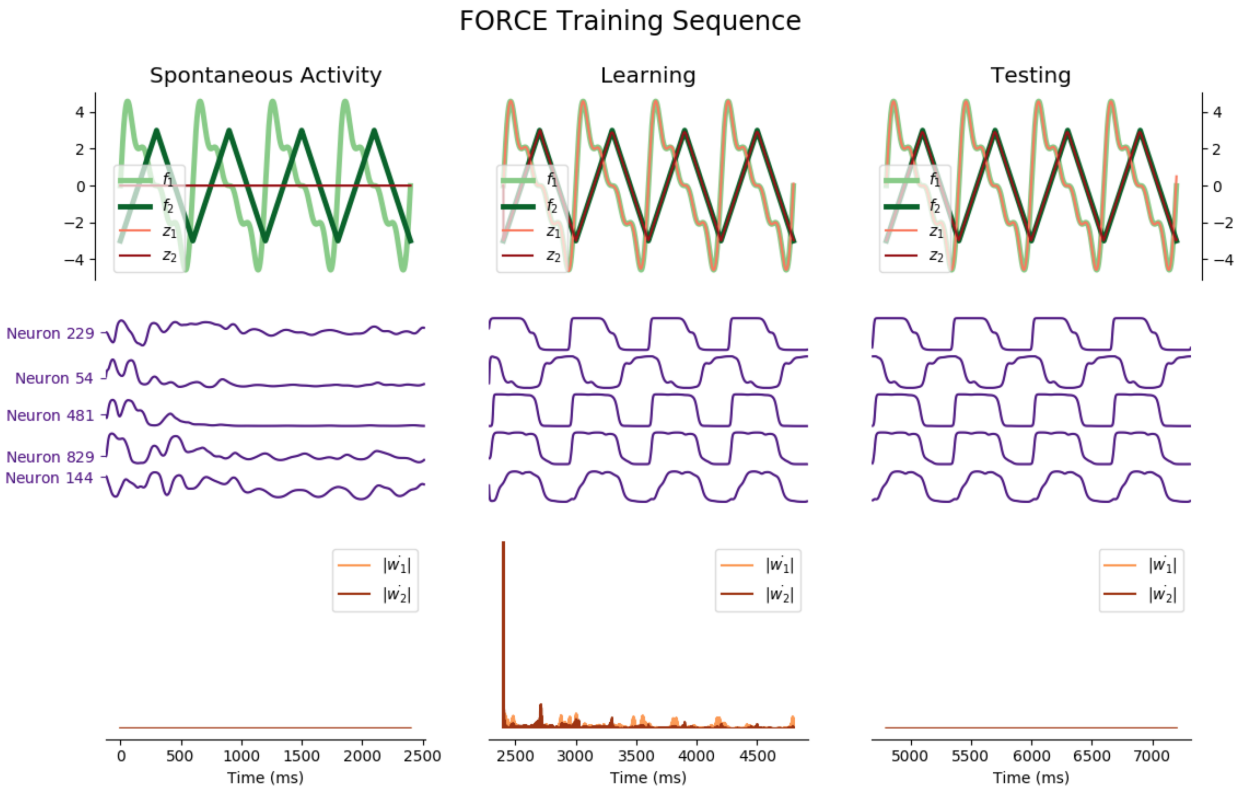


Figure 3.A.4. - FORCE training sequence (each phase lasting 2400 ms) for two simultaneous outputs (each one associated to one of the two readout units): a sum-of-sinusoids AND a triangle-wave target functions

> ***Average Train Errors:*** $(0.017, 0.010)$
> ***Average Test Errors:*** $(0.067, 0.040)$

Three simultaneous outputs

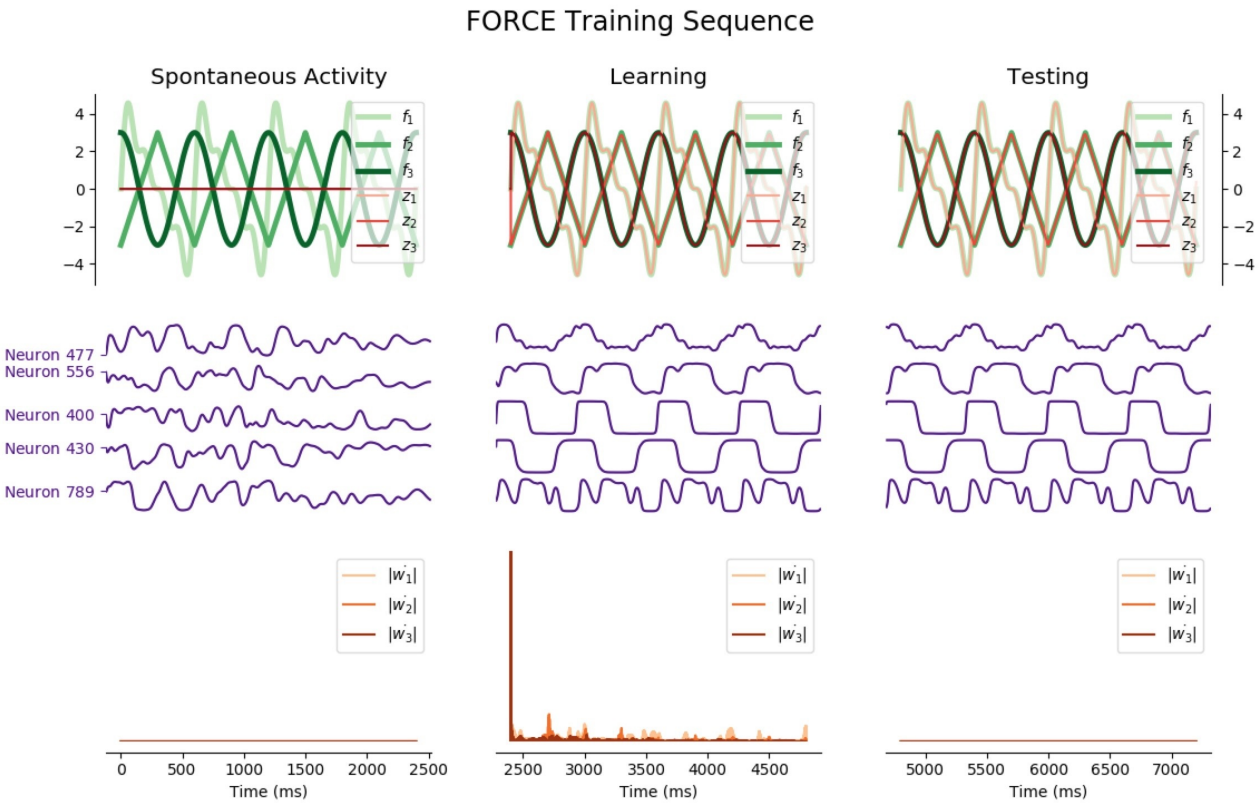


Figure 3.A.5. - FORCE training sequence (each phase lasting 2400 ms) for three simultaneous outputs (each one associated to one of the three readout units): a sum-of-sinusoids, a triangle-wave and a cosine target functions

> ***Average Train Errors:*** $(0.019, 0.012, 0.009)$
> ***Average Test Errors:*** $(0.073, 0.050, 0.050)$

Principal Component Analysis (PCA)

As a matter of fact, most of the network activity can be accounted for by a few leading principal components.

Indeed, after training the network so that it produces different target functions (for various numbers of outputs), one can apply principal component analysis (PCA) to project the network activity on a handful (8 is enough for the following examples) of principal components. Then, it appears that most of the target patterns can be obtained from these few projections: out the hundreds of degrees of freedom available (one thousand in our case, as $N_G = 1000$), only a dozens are actually necessary to produce the target functions we have considered.

NB: the code to generate the following figures is in the `principal_component_plots.py` file of the github repository.

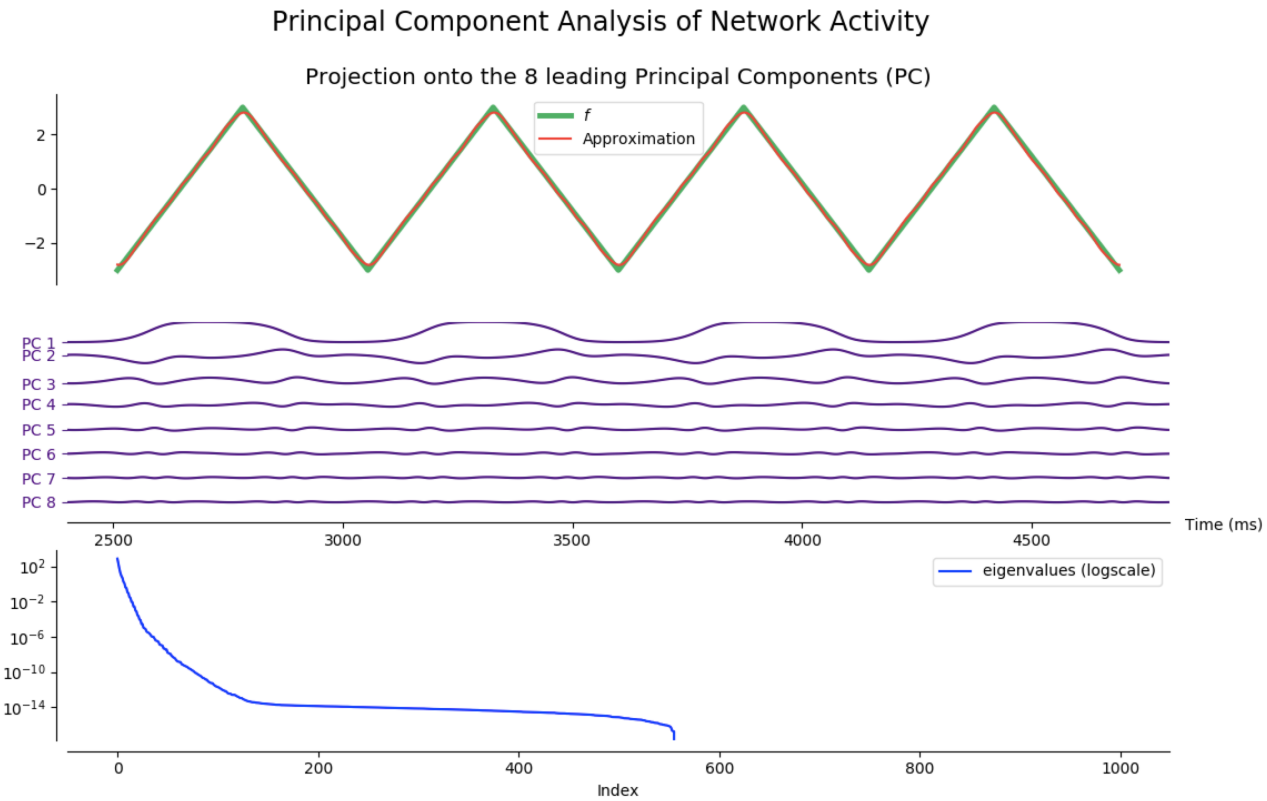


Figure 3.A.6. - Triangle-wave target function (one output): Approximation using activity projected onto the 8 Leading Principal Components (LPC), Projections of network activity onto the LPC, and (logscale) plot of PCA eigenvalues.

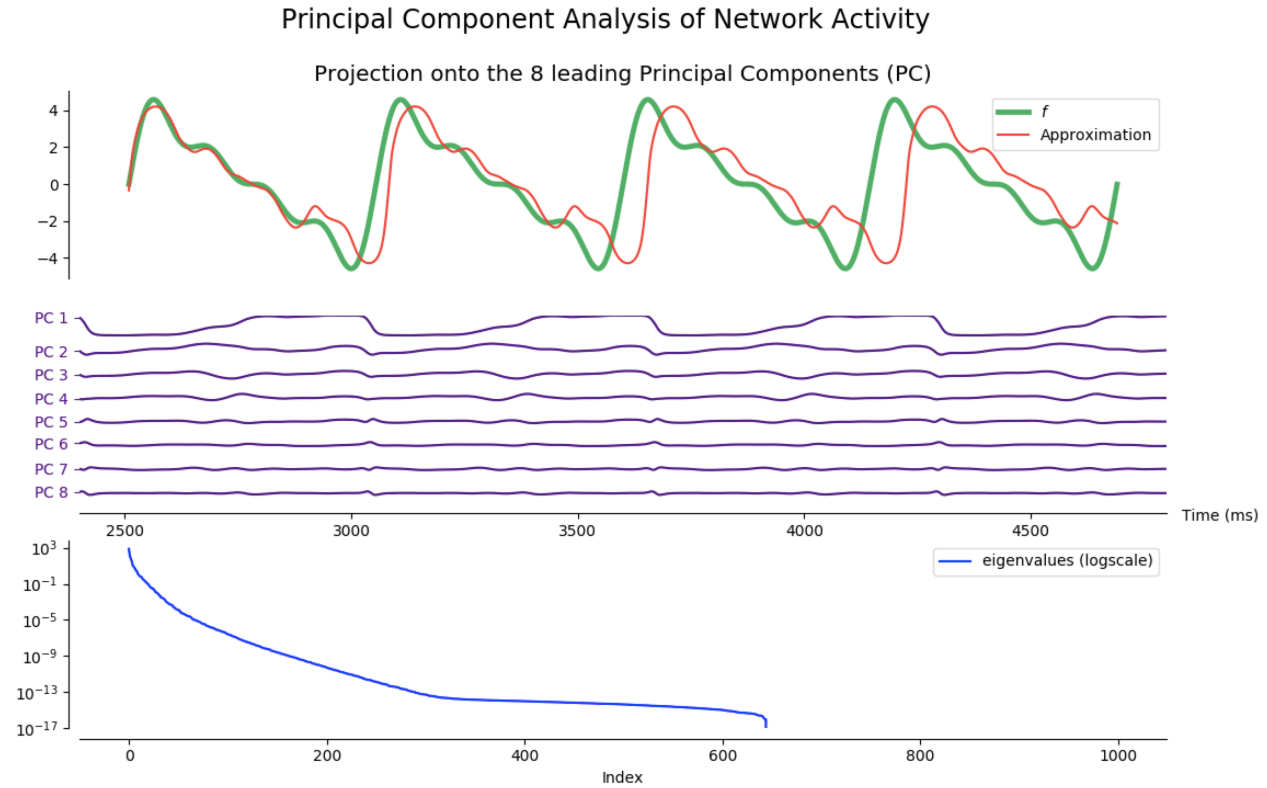


Figure 3.A.7. - Sum-of-sinusoids target function (one output): Approximation using activity projected onto the 8 LPC, Projections of network activity onto the LPC, and (logscale) plot of PCA eigenvalues.

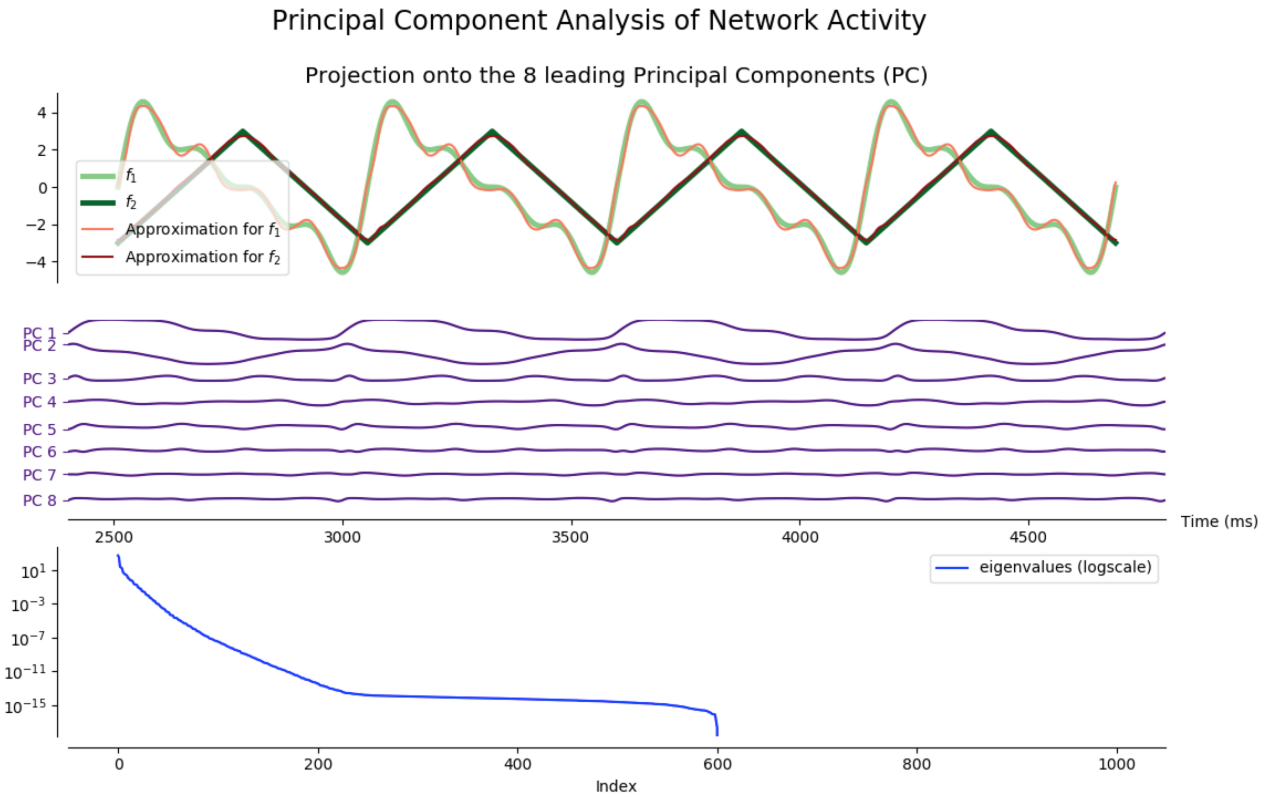


Figure 3.A.8. - Triangle-wave and sum-of-sinusoids target functions (two outputs): Approximation using activity projected onto the 8 LPC, Projections of network activity onto the LPC, and (logscale) plot of PCA eigenvalues.

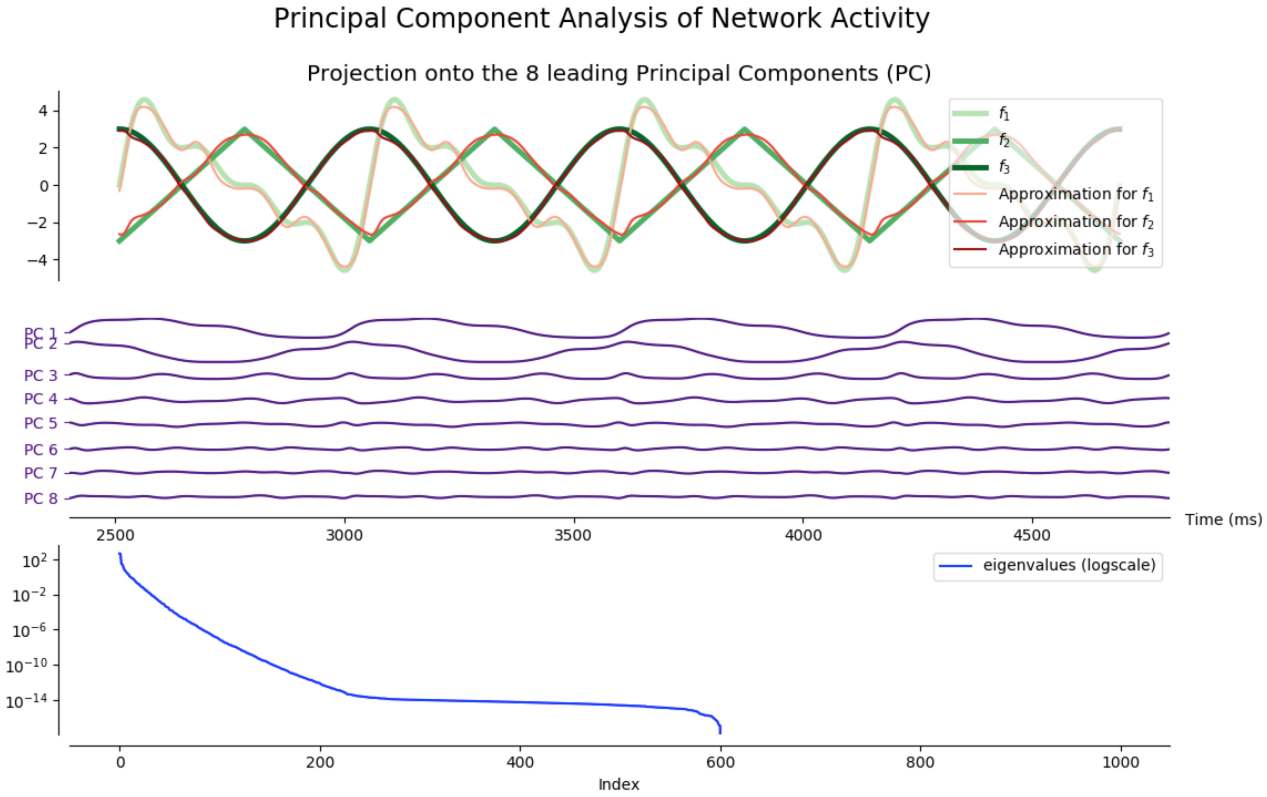


Figure 3.A.9. - Triangle-wave, sum-of-sinusoids and cosine target functions (three outputs): Approximation using activity projected onto the 8 LPC, Projections of network activity onto the LPC, and (logscale) plot of PCA eigenvalues.