Hebbian learning
Part Two

NSC3270 / NSC5270
Computational Neuroscience

Tu/Th 9:35-10:50am
Featheringill 129

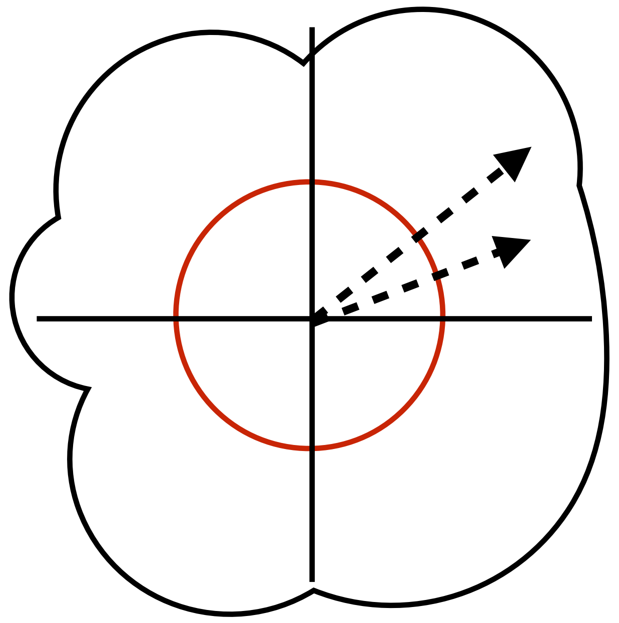
Professor Thomas Palmeri
Professor Sean Polyn

Neural network theory

Pattern recognition

Decision making

Similarity

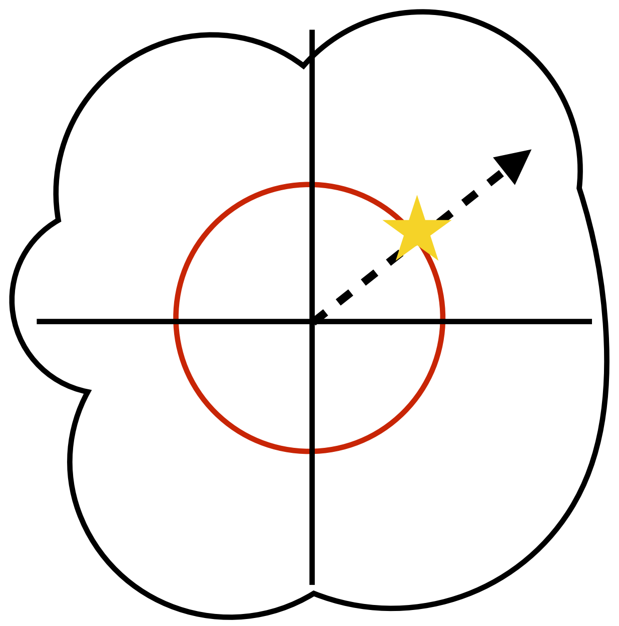


Neurons making decisions based on similarity in a geometric representational space

Why vector normalization?

Last time we introduced the idea of divisive normalization

Often the magnitude of a signal is not as important as the multivariate pattern of the signal



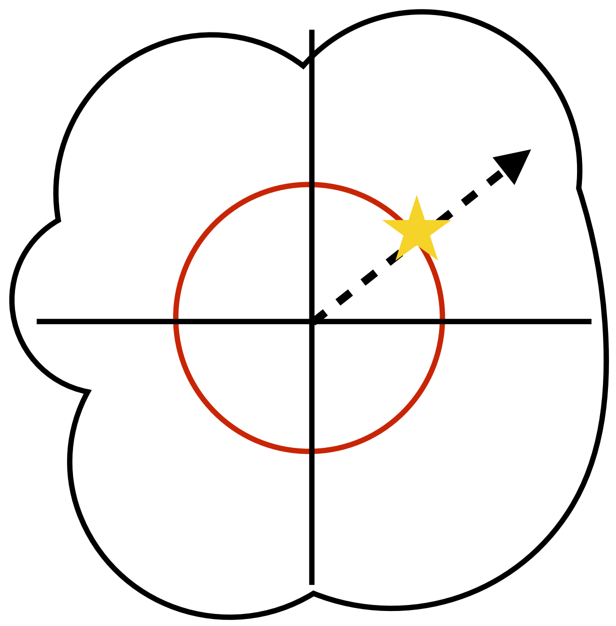
This unit circle is a 1-d manifold embedded in a 2-d plane

Mathematical operation: Normalization of a vector to the unit hypersphere

Why vector normalization?

Normalization of activation patterns across a population of neurons

Feasible in a neural system; inhibitory interneurons with a broad reach



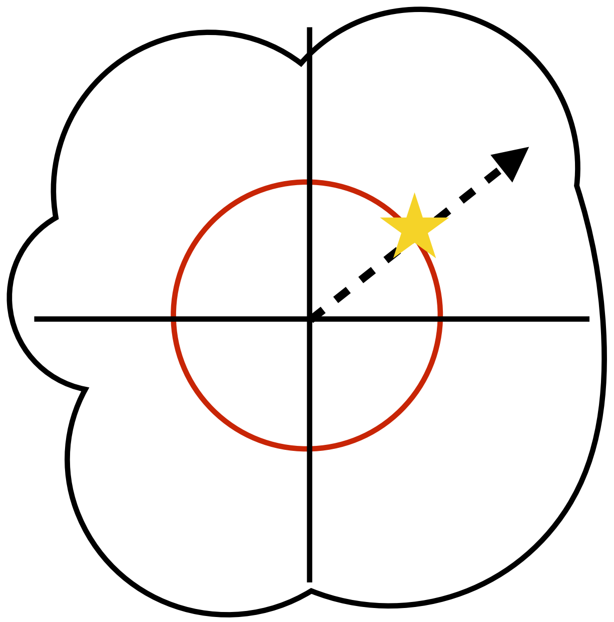
This unit circle is a 1-d manifold embedded in a 2-d plane

Mathematical operation: Normalization of a vector to the unit hypersphere

Why vector normalization?

Normalization of synaptic strengths across all the synapses of a neuron

Weights are neurophysiological structures that must be maintained, & resources are limited



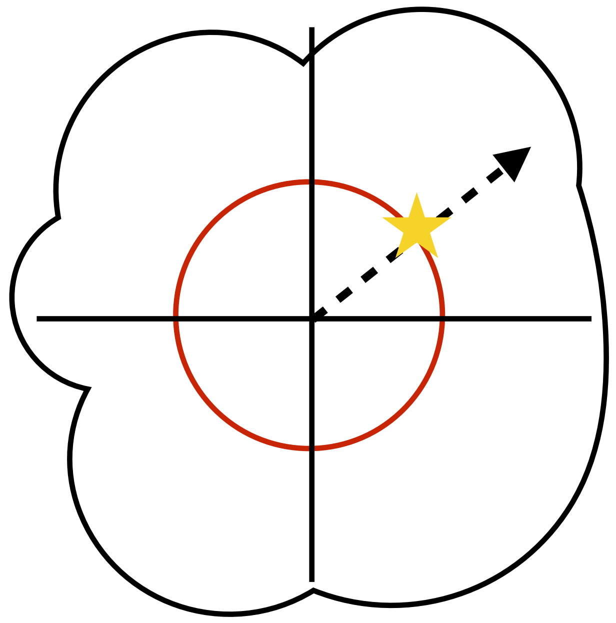
This unit circle is a 1-d manifold embedded in a 2-d plane

Mathematical operation: Normalization of a vector to the unit hypersphere

Why vector normalization?

“Excitation of the postsynaptic neuron is greatest when the input vector matches the weight vector.”

This is true when inputs and weights are normalized to be of unit length



This unit circle is a 1-d manifold embedded in a 2-d plane

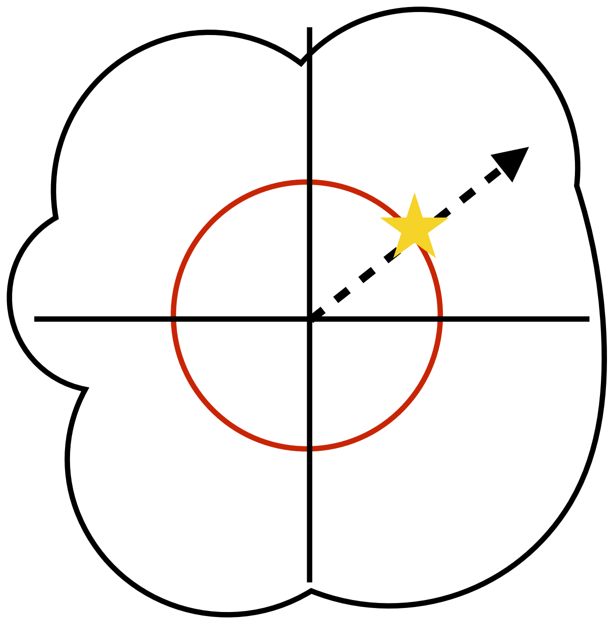
This idea is at the heart of pattern recognition

& Hebbian learning allows weights to approach the average input pattern

Normalization puts points onto the unit hypersphere

The operation is very simple in Python

```
x = r.rand(3,);
x = x / np.linalg.norm(x);
```



This unit circle is a 1-d manifold embedded in a 2-d plane

This idea is at the heart of pattern recognition

& Hebbian learning allows weights to approach the average input pattern

Constructing a model world

In order to explore the properties of these models, we need to construct a world for them to live in, an environment for them to experience

Using random variables and probability distributions to create model worlds with known statistical properties

Constructing a model world

With very few assumptions (just activation normalization and weight normalization) the simple Hebbian learning rule allows a neuron to learn the average pattern of excitation across its synapses, and tune itself to respond maximally to this pattern

Constructing a model world

For HW#4 you’ll be creating a model world in which input patterns are points on a unit circle. The network you create will have 2 input units, one output unit, and will use a Hebbian learning rule with weight normalization.

The weights of the network will approach the prototype of the noisy input patterns.

(Open HW4 detail slides)

Random number generation

Uniform random from 0 to 1 (`rand`)

How to scale and shift this to get points on a different interval, -1 to 1

Creating random points on the unit circle

(v1) generate random points in 2-d and project each one to the unit circle using vector normalization

(v2) generate random points in radians and use cos and sin to get the coordinates in 2-d space.

(you’ll do v2 for the homework)

Cosine similarity

Using cosine to quantify similarity of two points

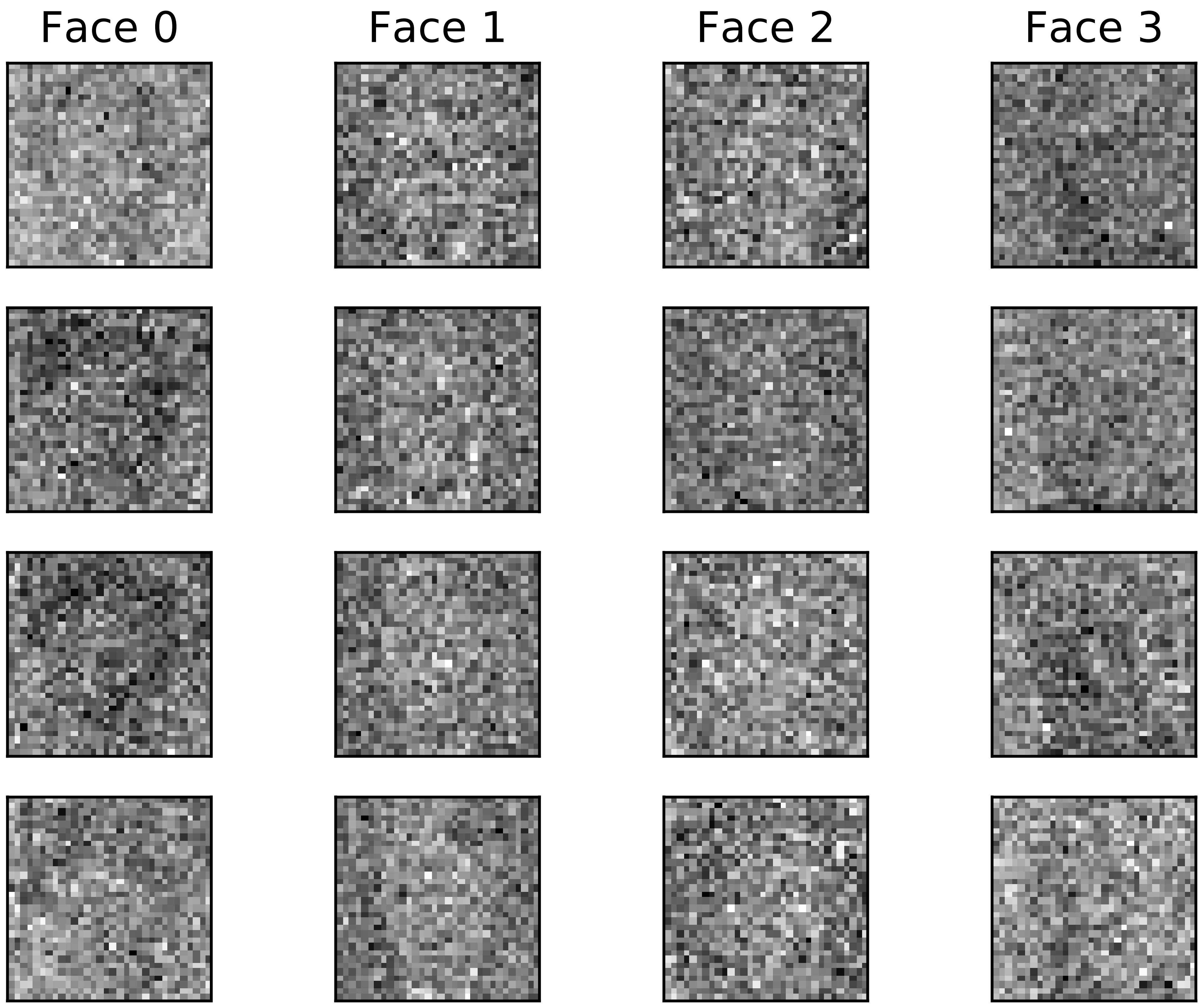
2-d example on board, 3-d example with `rand(3,1)`

```
>> dot(x1,x2) / (norm(x1) * norm(x2))
```

It doesn’t matter how high-dimensional your patterns are, cosine similarity just involves 3 points (the two points being compared, and the origin), and 3 points define a 2-d plane

Cosine similarity for two points on the unit circle is particularly straightforward, can you see why?

Learning a running average



Learning a running average

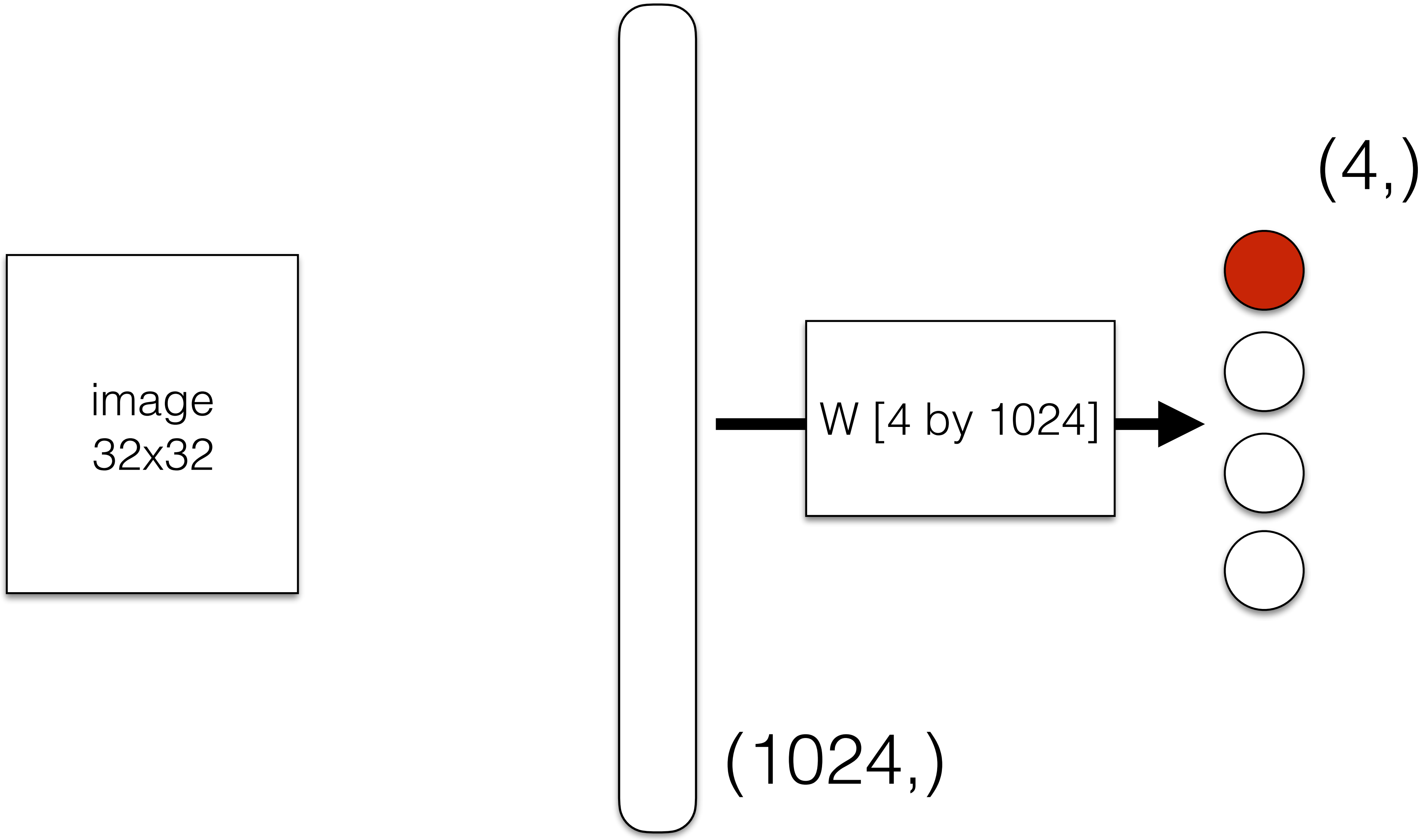
32x32 grayscale images of 4 different celebrity faces (1024 input units), 4 output units, one for each face.

Adding noise to a source image to create each noisy image: Sample 1024 values from a Gaussian distribution and add them to each pixel (`randn`), multiplied/scaled by 0.07.

After noising, activation patterns are normalized to the unit hypersphere.

Model uses simple Hebbian learning with weight normalization. During training, the output unit corresponding to the face identity of the current pattern is clamped at 1, the other output units are clamped at 0.

Hebb_face_noise.py

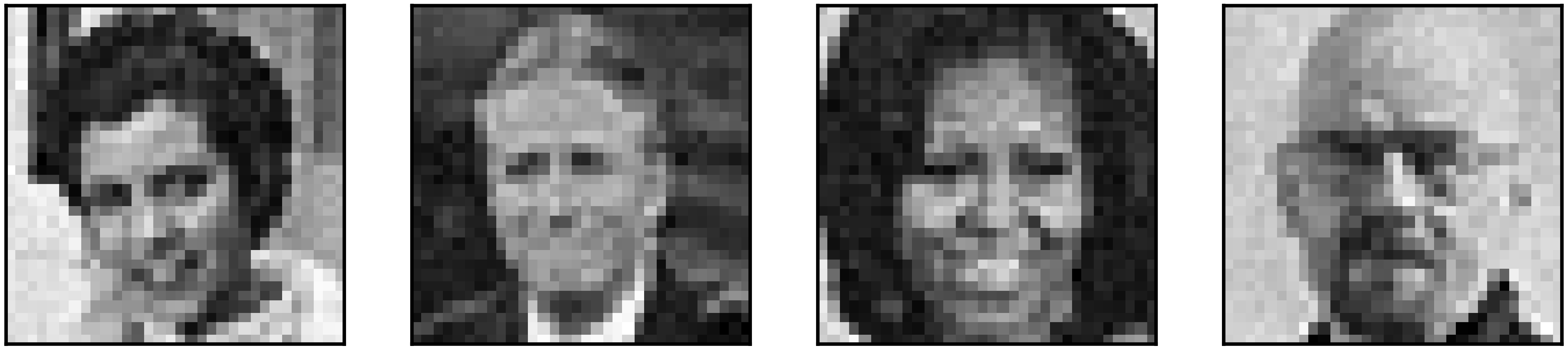


Input is different noisy versions of the face picture. Correct output unit is clamped at 1. The output unit is trying to extract what face is present from the noise.

Hebb_face_noise.py

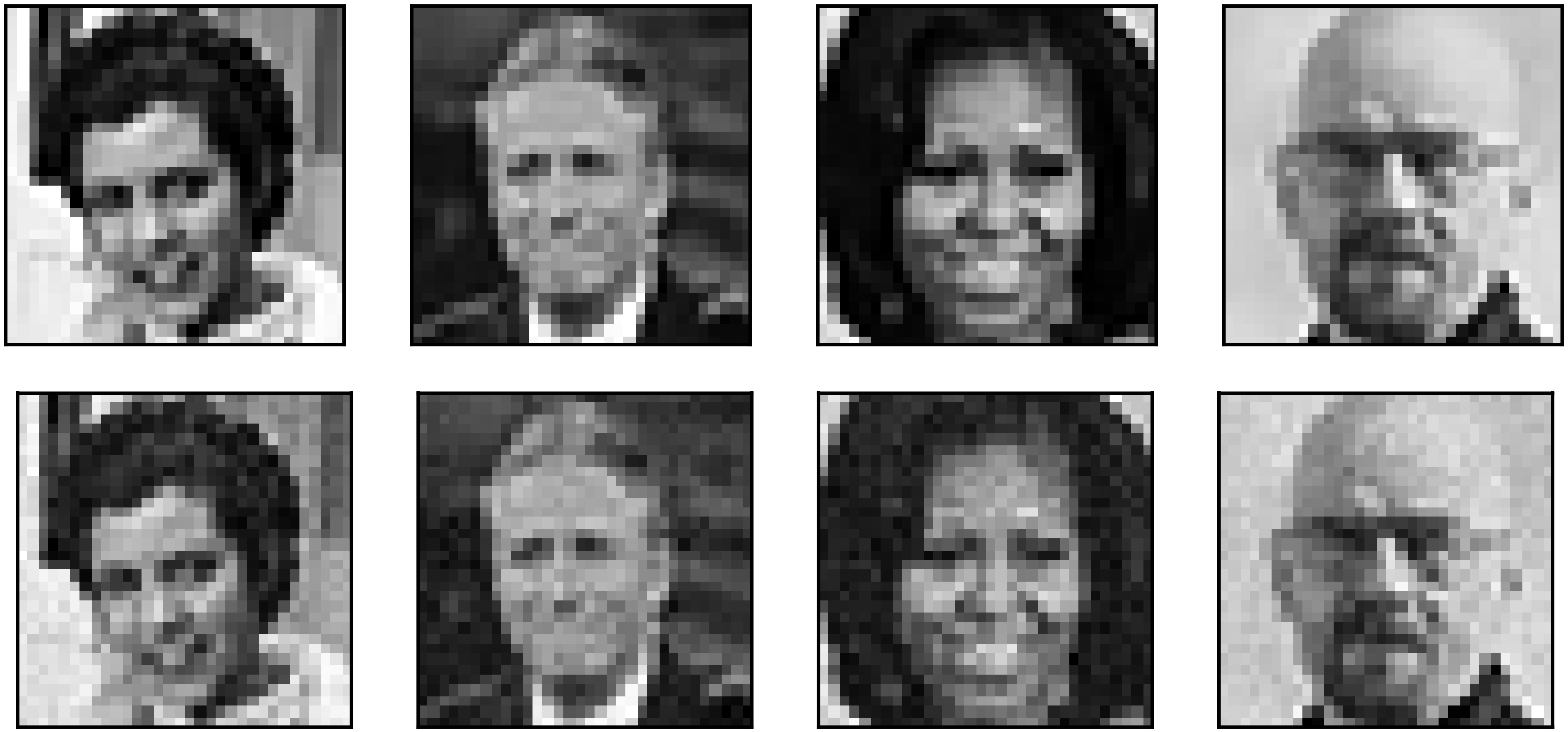
Run it in WING to see the 4 faces embedded in the weights

Hebb_face_noise.py



Images of the network weights post-training. The network can recover the original images even though it only ever sees the highly corrupted noisy images.

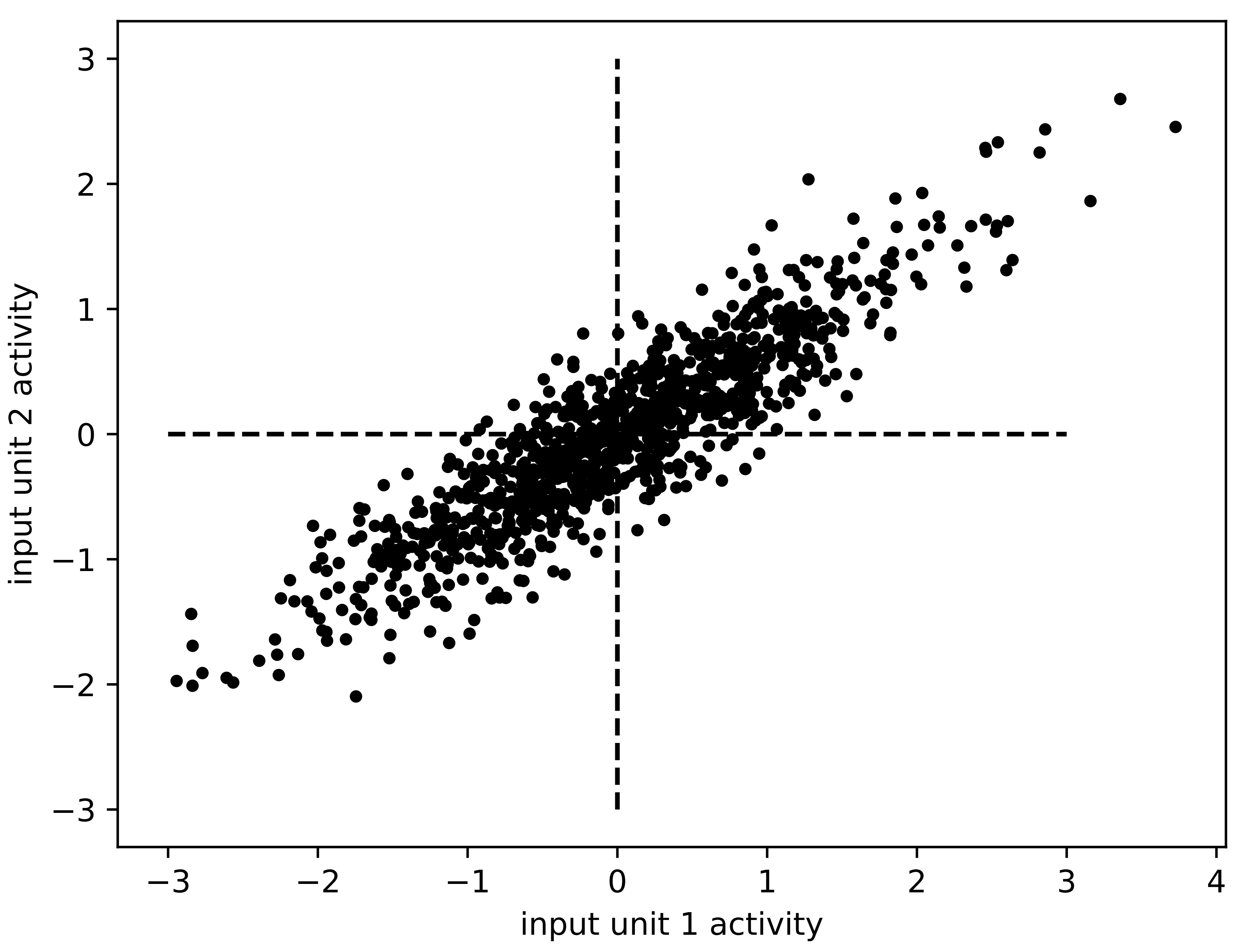
Hebb_face_noise.py



Top row: Original images

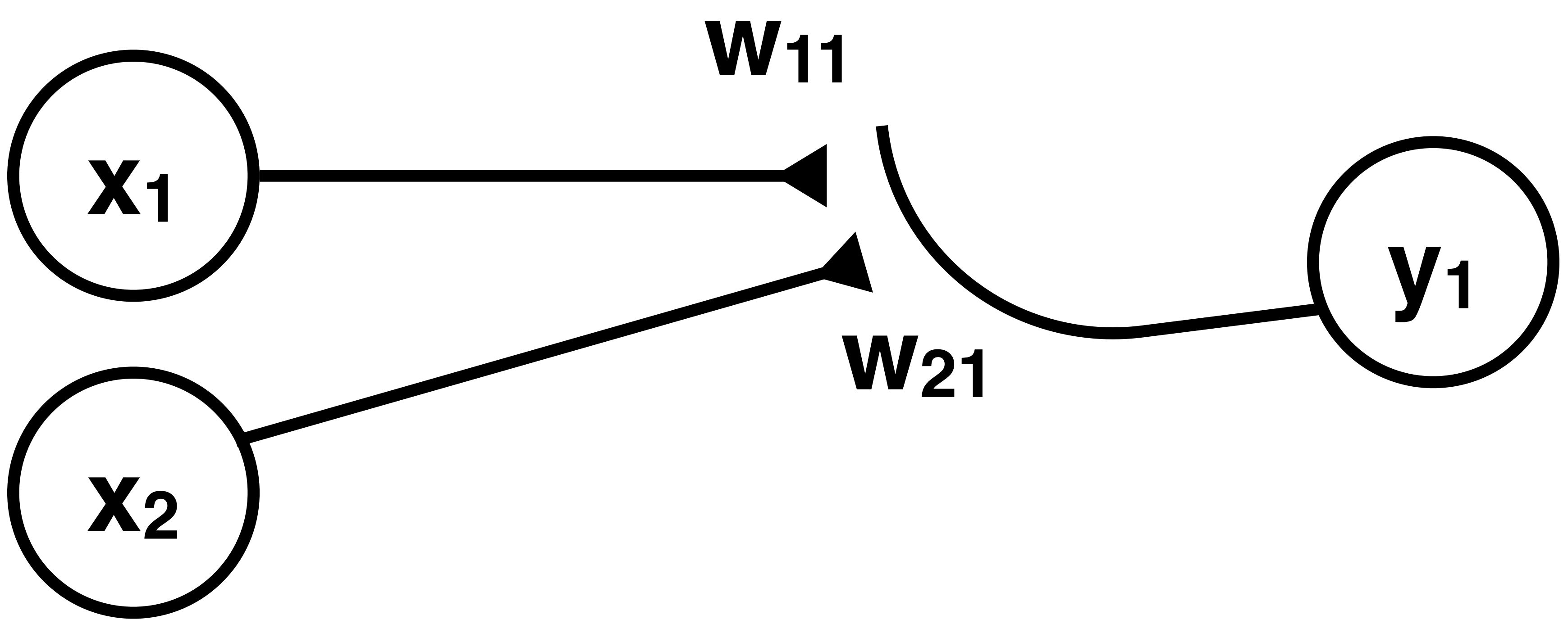
Bottom row: Recovered images

Oja’s rule and Principal Component Analysis



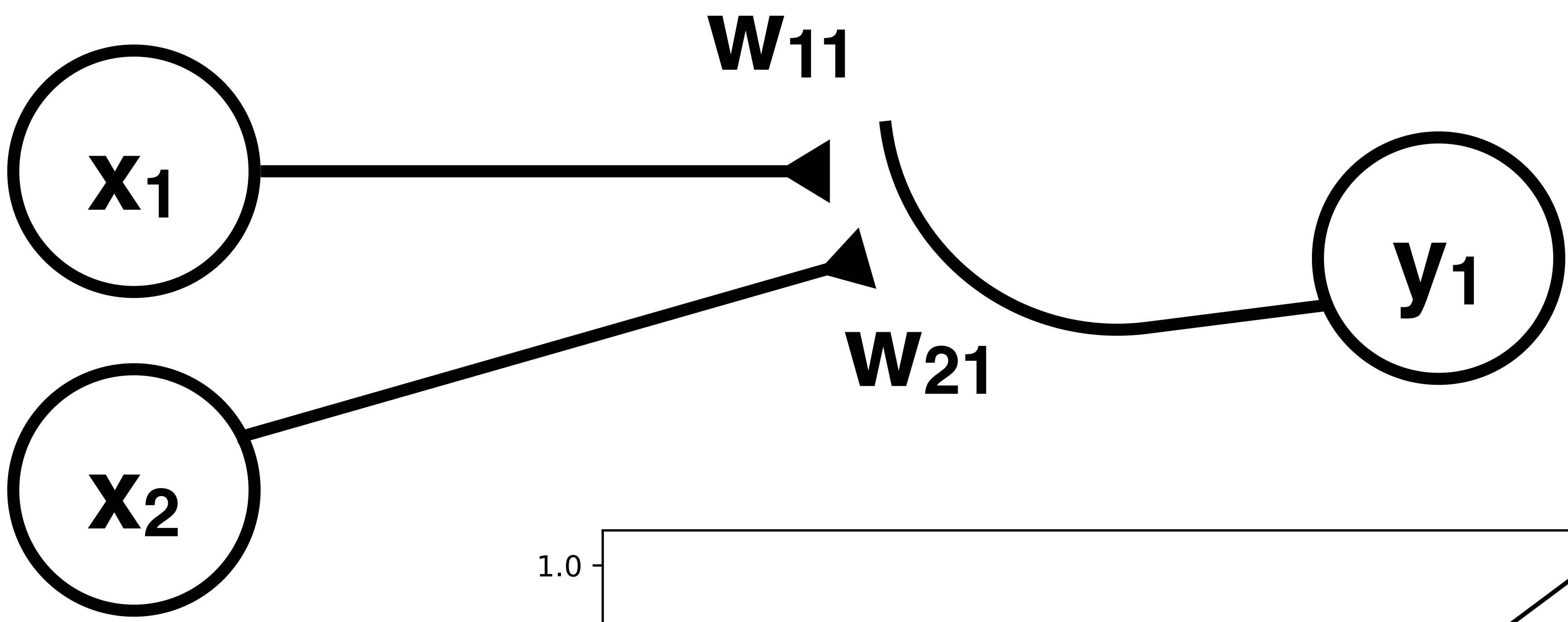
Input activation doesn’t have to get normalized. Consider this model world, 2 input units with positively correlated activation. Each point is an input pattern.

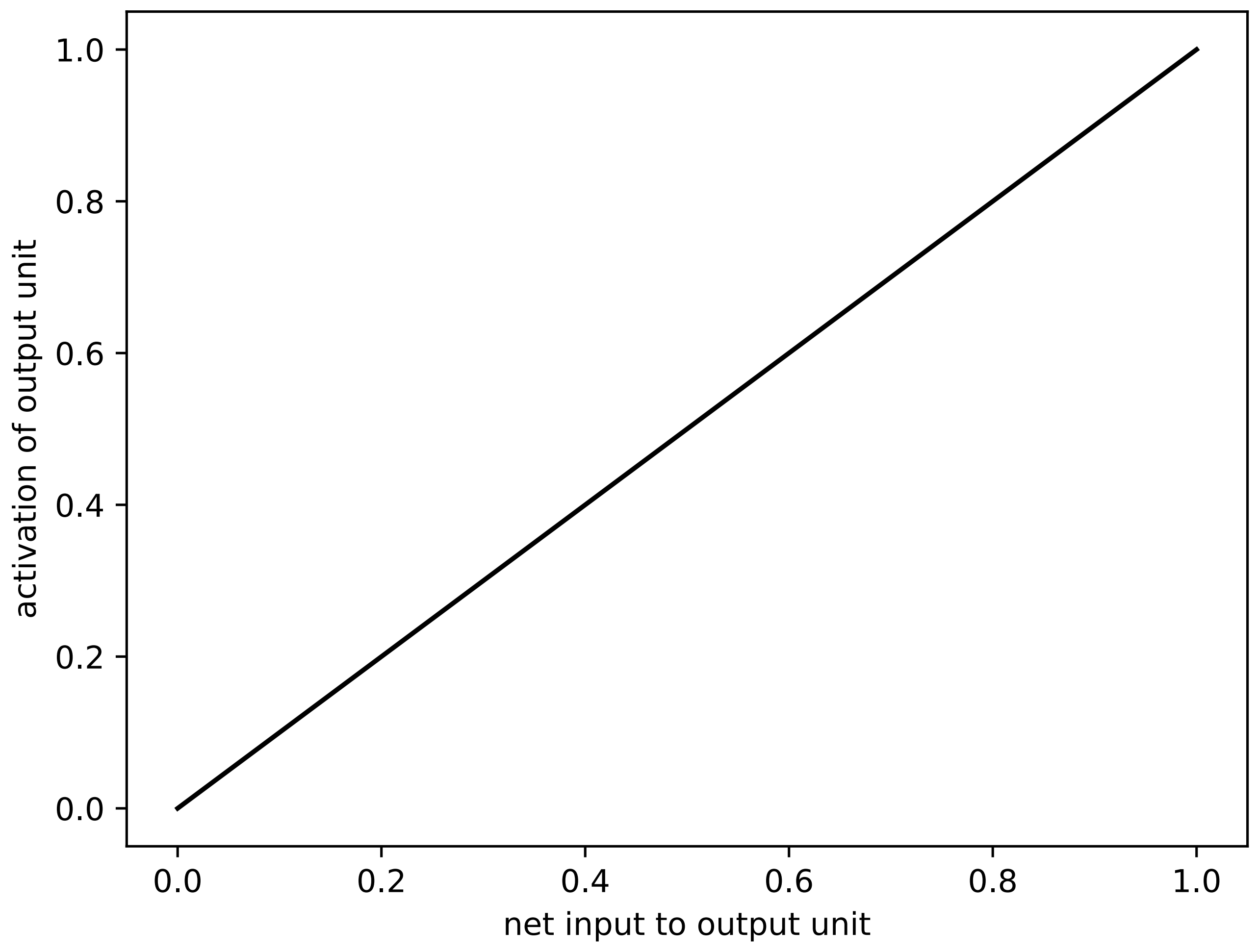
Oja's rule and Principal Component Analysis



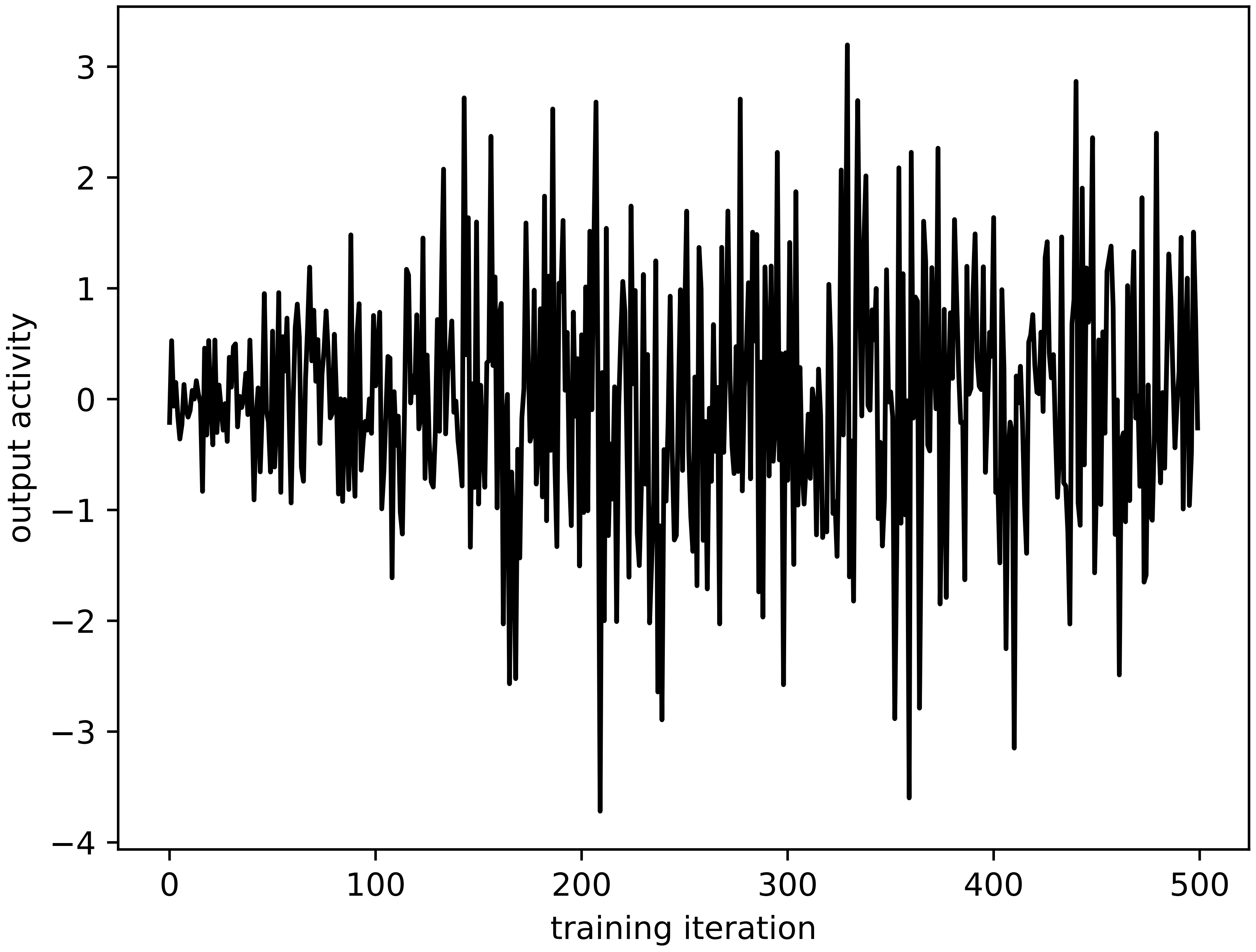
We create a simple network. Two input units, one output unit. The output unit has a linear activation function. In other words, the activation of y is literally the dot product of the activation vector and the weight vector.

Oja’s rule and Principal Component Analysis

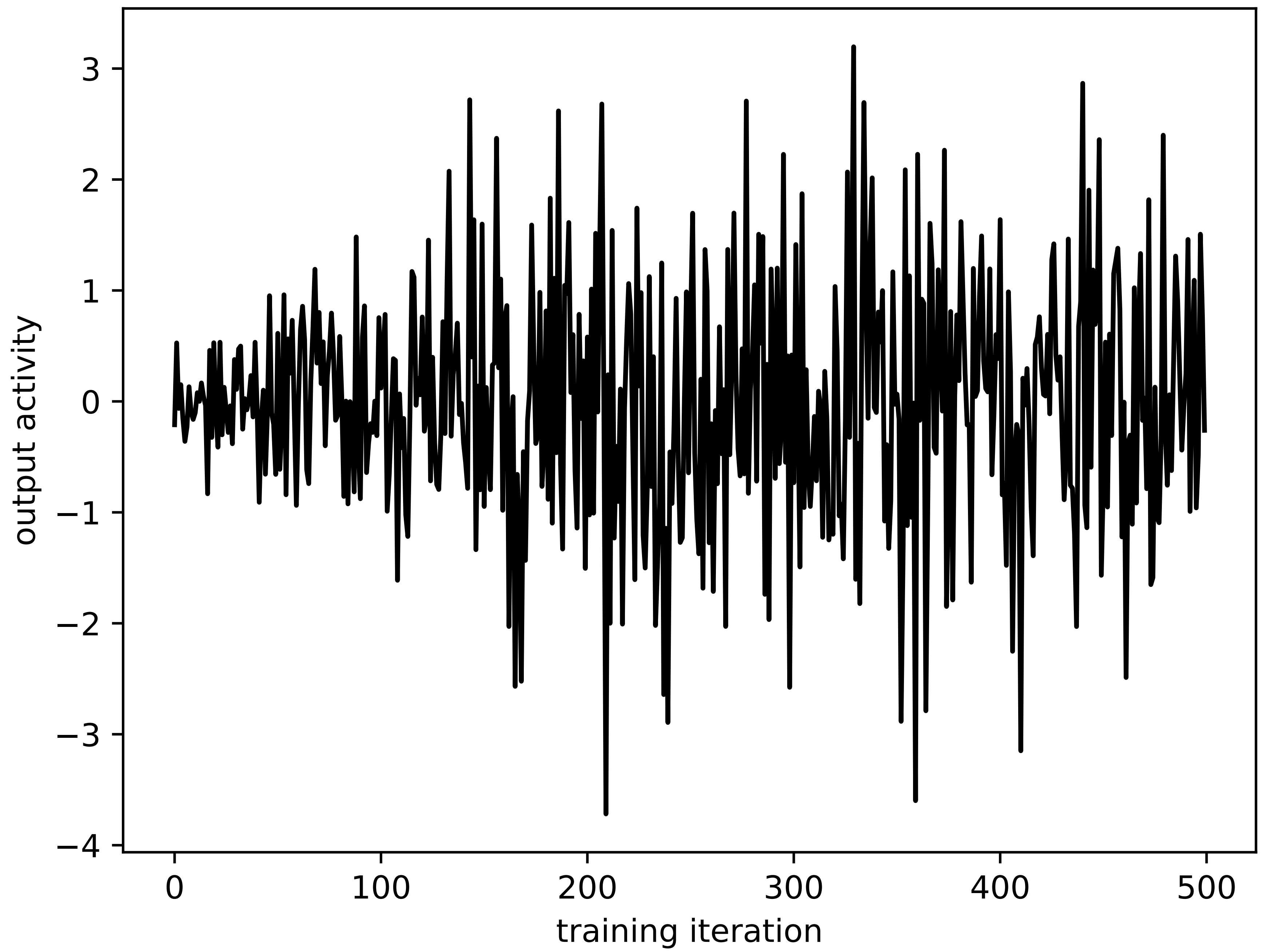




Oja’s rule and Principal Component Analysis

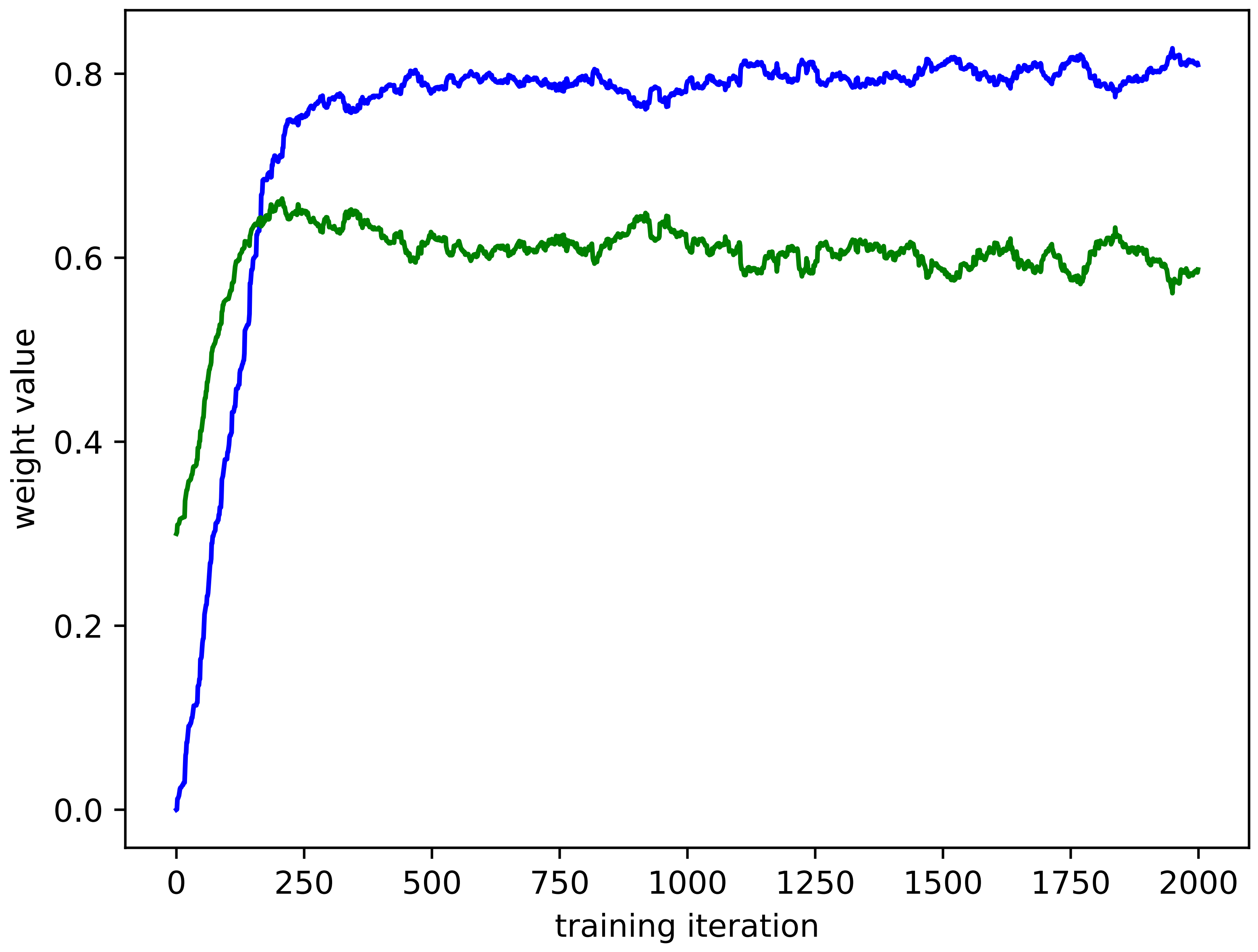


Oja’s rule and Principal Component Analysis



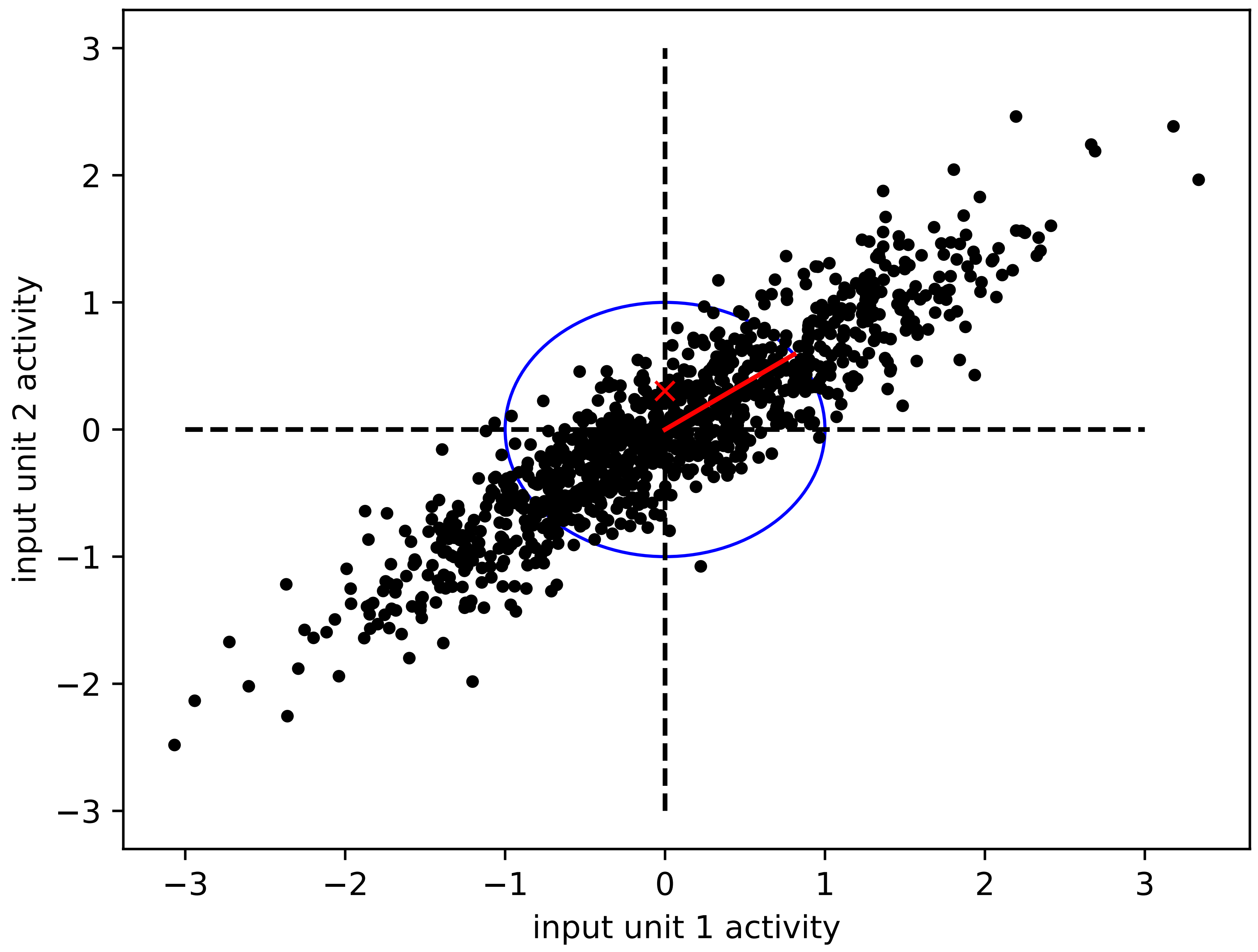
As learning progresses, the variance of the output unit activity increases.

Oja's rule and Principal Component Analysis



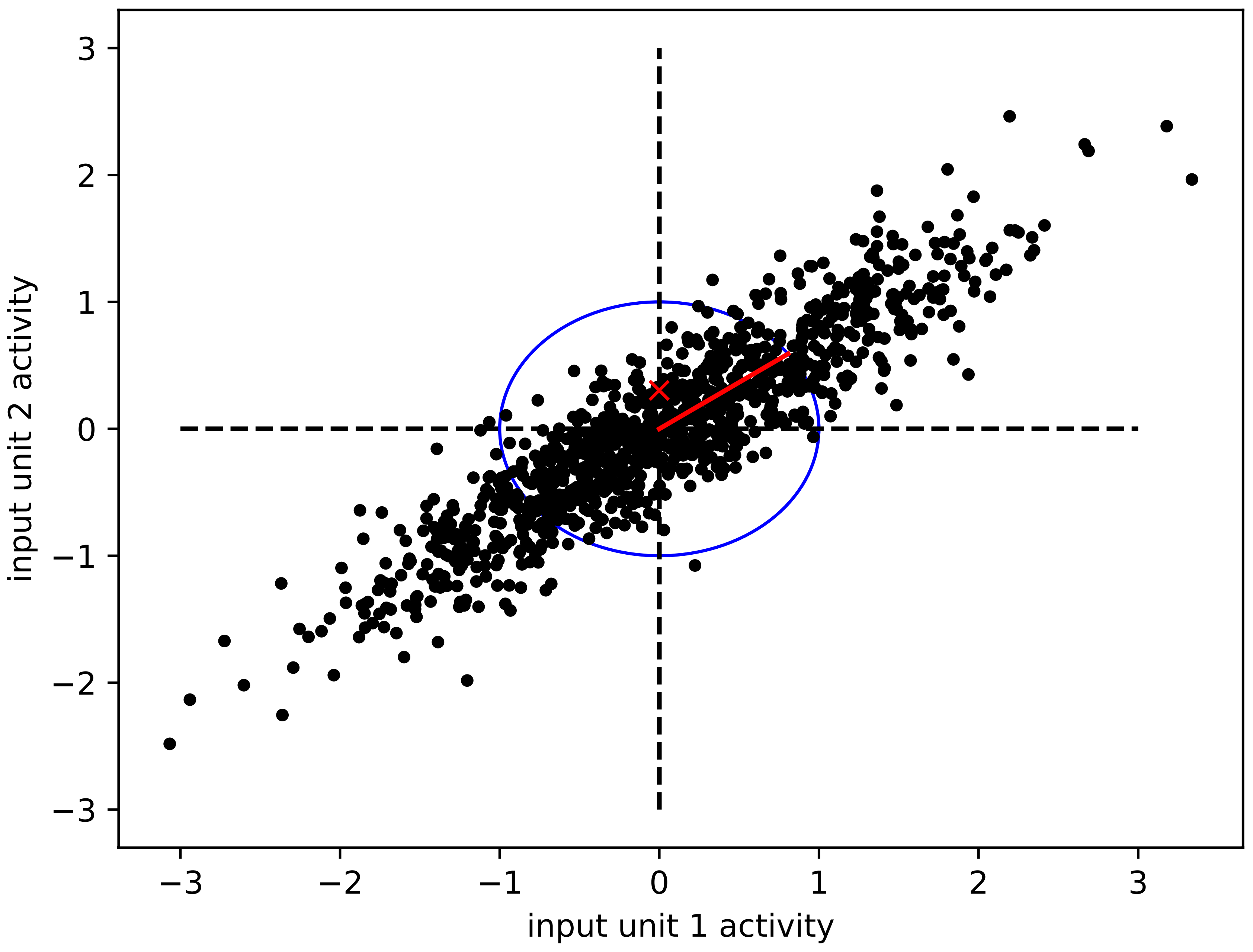
The weights converge fairly quickly (rate of convergence will depend on learning rate parameter). The weights don't start off at length 1, but they gradually converge to a vector of length 1.

Oja's rule and Principal Component Analysis



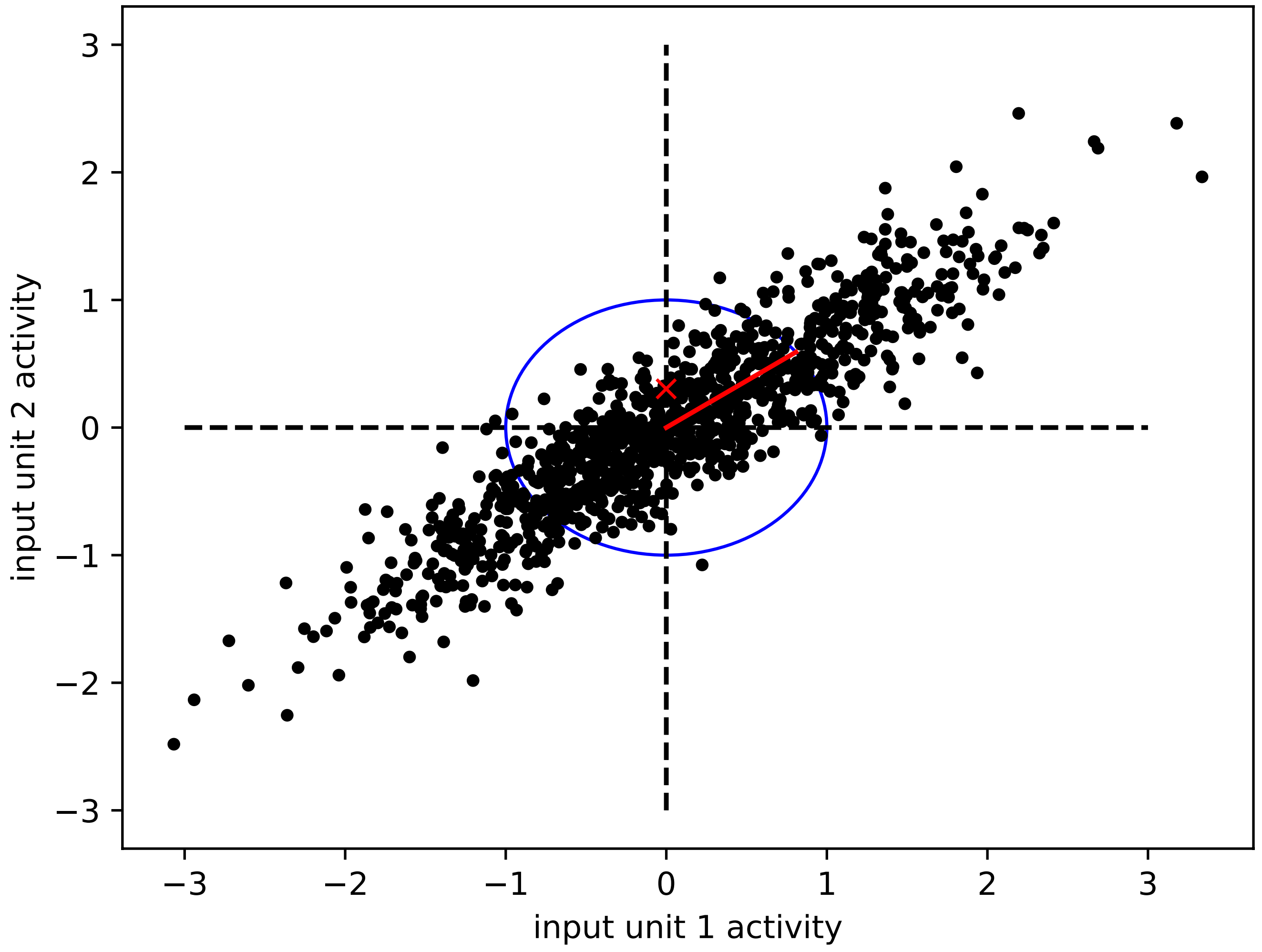
It isn't learning a running average of the training patterns, it is learning the axis of maximum variability of the signal. Why might this be useful?

Oja’s rule and Principal Component Analysis



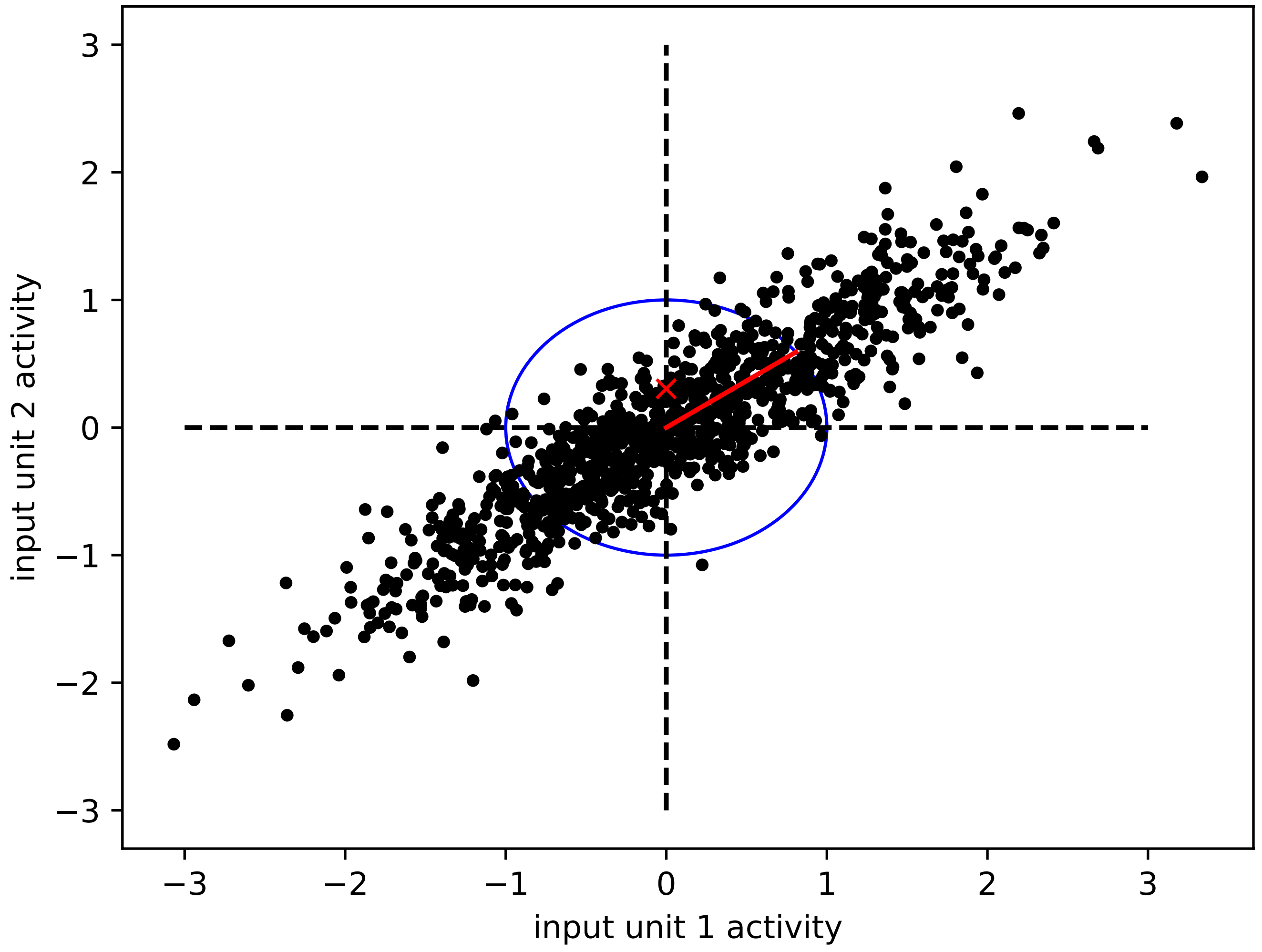
Can you see why the variability of the output unit activity increases as the weight vector approaches this final state?

Oja’s rule and Principal Component Analysis



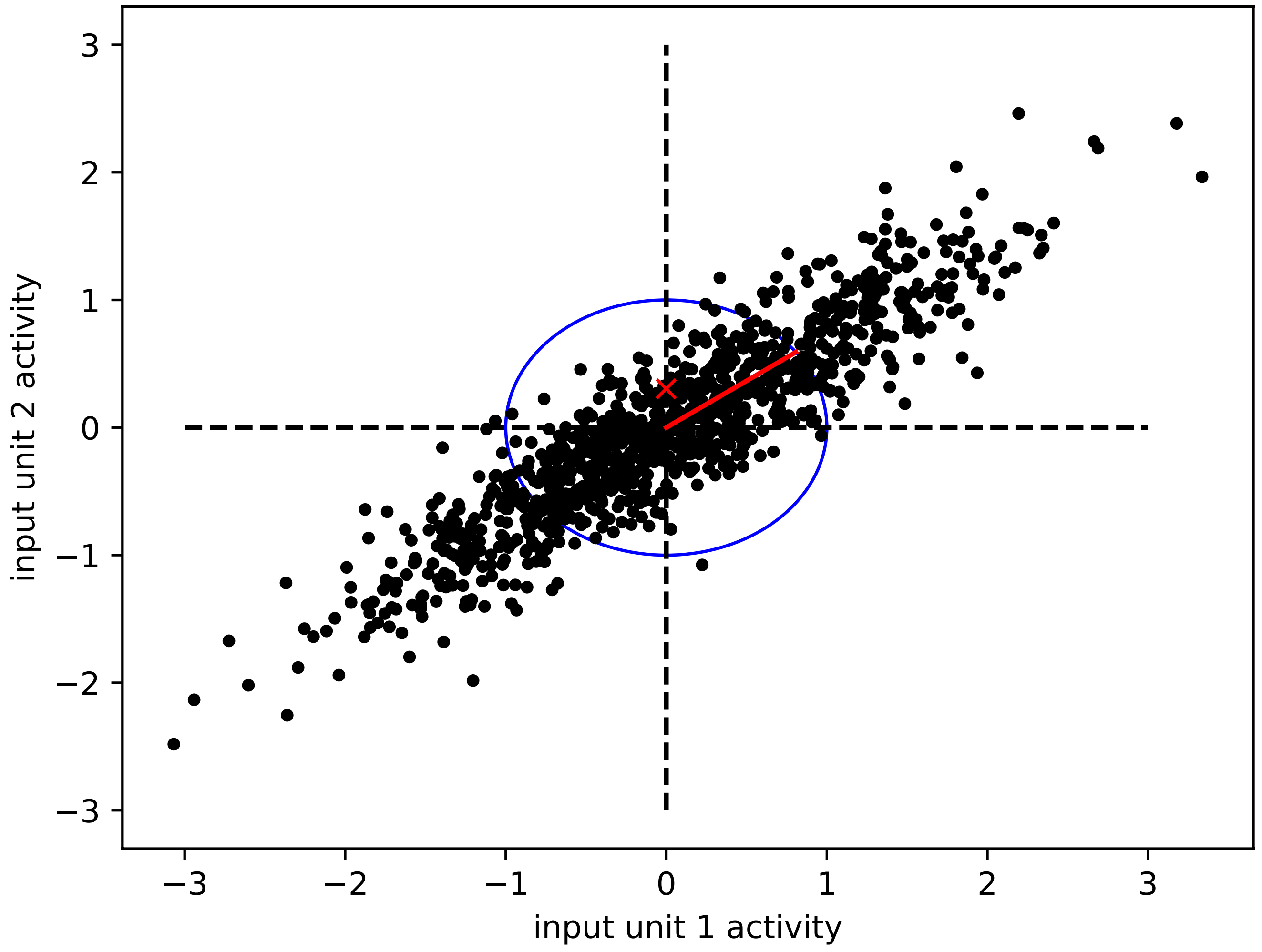
In a sense, the weights are still trying to learn the running average of the input patterns. The true average is zero, but the normalization factor won’t let the weight vector approach zero. Every time it takes a step towards zero it gets normalized back to the unit circle.

Oja's rule and Principal Component Analysis



There's another potential stable point that the weights could end up at. Do you know what it is?

Oja’s rule and Principal Component Analysis



It is possible to extend this system to identify other sources of variance in the signal by adding other output units and some mechanism to make sure they don’t all identify the first principal component. In this signal there’s a second component orthogonal to the first one, but it isn’t as strong.