ML2R Coding Nuggets
Solving the Single Unit Oja Flow

Christian Bauckhage*
Machine Learning Rhine-Ruhr
Fraunhofer IAIS
St. Augustin, Germany

Sebastian Müller
Machine Learning Rhine-Ruhr
University of Bonn
Bonn, Germany

Fabrice Beaumont
Computer Science
University of Bonn
Bonn, Germany

ABSTRACT

Ojas' rule for neural principal component learning has a continuous analog called the Oja flow. This is a gradient flow on the unit sphere whose equilibrium points indicate the principal eigenspace of the training data. We briefly discuss characteristics of this flow and show how to solve its differential equation using *SciPy*.

1 INTRODUCTION

Previously [3], we studied **Oja's rule** which extends **Hebb's rule** for how a single neuron adjusts its synaptic weights in an unsupervised manner. Oja's rule is celebrated for the fact that it causes the weight vector of a *linear neuron* to converge to the first principal component of its training data. In other words, if we consider a linear neuron with weights $\boldsymbol{w}$ which, for a given input $\boldsymbol{x}$, computes $y = \boldsymbol{w}^\intercal \boldsymbol{x}$ and let this neuron learn according to Oja's rule, we observe the following:

If we assume a *zero mean* sample of n training data points $\boldsymbol{x}_j$, we may think of them as the columns of a data matrix

$$X = \begin{bmatrix} \boldsymbol{x}_1 \, \boldsymbol{x}_2 \cdots \boldsymbol{x}_n \end{bmatrix} \tag{1}$$

and thus compute the sample covariance matrix as

$$C = \tfrac{1}{n} X X^\intercal \tag{2}$$

The *expected* weight update in the k-th iteration of learning under Oja's rule then amounts to

$$\Delta \boldsymbol{w}_k = C \boldsymbol{w}_k - \boldsymbol{w}_k \boldsymbol{w}_k^\intercal C \boldsymbol{w}_k \tag{3}$$

and, if the initial weight vector $\boldsymbol{w}_0$ was of unit length, i.e. $\|\boldsymbol{w}_0\| = 1$, the discrete iterative process

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \eta \cdot \Delta \boldsymbol{w}_k \tag{4}$$

with learning rate $\eta > 0$ will converge to the dominant eigenvector $\boldsymbol{u}_1$ of matrix C which spans the principal eigenspace of the data in matrix X.

In this note, we are concerned with a continuous generalization of the iteration in (3), the so called single-unit Oja flow [20].

In section 2, we derive the corresponding ordinary differential equation from the above finite difference scheme and briefly discuss important properties of the Oja flow. Then, in section 3, we numerically solve the corresponding initial value problem using methods in *SciPy*'s `integrate` package. Readers who would like to experiment with our exemplary code should be familiar with *NumPy* and *SciPy* [15] and only need to

```
import numpy as np
import numpy.linalg as la
from scipy.integrate import odeint
```

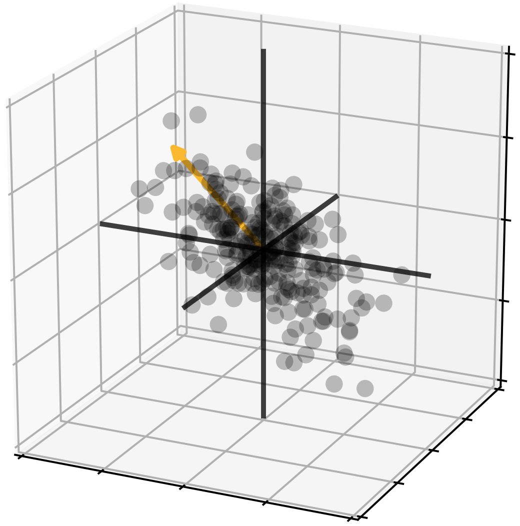

Figure 1: A sample of 3D data points $X = [\boldsymbol{x}_1 \, \boldsymbol{x}_2 \cdots \boldsymbol{x}_n]$ (grey dots) and the dominant eigenvector $\boldsymbol{u}_1$ (orange arrow) of the sample covariance matrix C. Observe that $\boldsymbol{u}_1$ spans the major principal axis of the data in X.

2 THEORY

This section explains what it means to say that Oja's rule has a continuous analog. That is, we show that (4) is but a finite difference approximation of a vector-valued ordinary differential equation.

To begin with, we note that plugging (3) into (4) followed by some simple algebra yields

$$\frac{\boldsymbol{w}_{k+1} - \boldsymbol{w}_k}{\eta} = C \boldsymbol{w}_k - \boldsymbol{w}_k \boldsymbol{w}_k^\intercal C \boldsymbol{w}_k \tag{5}$$

Given this expression, we next introduce a new parameter $t = k\,\eta$ so that $t + \eta = k\,\eta + \eta = (k+1)\,\eta$. If we then assume that $\boldsymbol{w}(t) = \boldsymbol{w}_k$, we have $\boldsymbol{w}(t+\eta) = \boldsymbol{w}_{k+1}$ and can write (5) as

$$\frac{\boldsymbol{w}(t+\eta) - \boldsymbol{w}(t)}{\eta} = C\,\boldsymbol{w}(t) - \boldsymbol{w}(t)\,\boldsymbol{w}(t)^\intercal C\,\boldsymbol{w}(t) \tag{6}$$

At this point, it is pretty obvious where we are headed, because, in the limit $\eta \longrightarrow 0$, the expression in (6) becomes

$$\tfrac{d}{dt} \boldsymbol{w}(t) = C\,\boldsymbol{w}(t) - \boldsymbol{w}(t)\,\boldsymbol{w}(t)^\intercal C\,\boldsymbol{w}(t) \tag{7}$$

This is indeed an ordinary differential equation or a continuous time dynamical system and we can think of Oja's rule in (4) as the **forward Euler method** for solving (7).

[*] 0000-0001-6615-2128

NOTE: in order to reduce notational clutter, we will henceforth (mostly) drop the dependency on parameter t and simply write

$$\frac{d\boldsymbol{w}}{dt} = \boldsymbol{C}\boldsymbol{w} - \boldsymbol{w}\boldsymbol{w}^\intercal\boldsymbol{C}\boldsymbol{w} \tag{8}$$

For reasons that will become apparent soon, the process in (8) is called the (single unit) Oja flow and we emphasize that there are different ways of writing it down. In particular, we have

$$\begin{aligned}\frac{d\boldsymbol{w}}{dt} &= \boldsymbol{C}\boldsymbol{w} - \big(\boldsymbol{w}\boldsymbol{w}^\intercal\big)\boldsymbol{C}\boldsymbol{w} &(9)\\ &= \big(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\intercal\big)\,\boldsymbol{C}\boldsymbol{w} &(10)\end{aligned}$$

as well as

$$\begin{aligned}\frac{d\boldsymbol{w}}{dt} &= \boldsymbol{C}\boldsymbol{w} - \boldsymbol{w}\big(\boldsymbol{w}^\intercal\boldsymbol{C}\boldsymbol{w}\big) &(11)\\ &= \boldsymbol{C}\boldsymbol{w} - \big(\boldsymbol{w}^\intercal\boldsymbol{C}\boldsymbol{w}\big)\,\boldsymbol{w} &(12)\\ &= \big(\boldsymbol{C} - \boldsymbol{w}^\intercal\boldsymbol{C}\boldsymbol{w}\boldsymbol{I}\big)\,\boldsymbol{w} &(13)\end{aligned}$$

Unfortunately, it is a bit involved to prove the existence of asymptotically stable equilibrium points of (8) or the (speed of) convergence of $\boldsymbol{w}(t)$ to one of these equlibria. Readers interested in a rigorous treatment of these issues are referred to a marvelous paper by Yan, Helmke, and Moore. They show that, the Oja flow converges exponentially fast to an equilibrium from any initial value $\boldsymbol{w}(0)$ and that this equilibrium indicates the dominant eigenvector of $\boldsymbol{C}$ for *almost* all initial values [20].

However, other important properties of the Oja flow are more easy to establish and we will discuss them next.

First of all, *if* we assume that the dynamical system in (8) has converged to an equilibrium $\boldsymbol{w}$ for which

$$\frac{d\boldsymbol{w}}{dt} = 0 \tag{14}$$

we then have for this equilibrium that

$$\boldsymbol{C}\boldsymbol{w} = \big(\boldsymbol{w}^\intercal\boldsymbol{C}\boldsymbol{w}\big)\,\boldsymbol{w} \tag{15}$$

Now, since $\boldsymbol{w}^\intercal\boldsymbol{C}\boldsymbol{w}$ is but a scalar, we may give it a name, say, λ. This way, we find $\boldsymbol{C}\boldsymbol{w} = \lambda\boldsymbol{w}$ which we immediately recognize as an eigenvalue / eigenvector equation. But this establishes

Lemma 2.1. If *the Oja flow converges to an equilibrium point, this equilibrium point corresponds to an eigenvector of matrix* $\boldsymbol{C}$*.*

Second of all, one can show that, if the Oja flow starts in an arbitrary point on the unit sphere, it will evolve on the unit sphere. Indeed, in the appendix, we prove the following

Lemma 2.2. *If the Oja flow starts in* $\boldsymbol{w}(0)$ *with* $\|\boldsymbol{w}(0)\| = 1$*, the flow is* isometric*, i.e.* $\|\boldsymbol{w}(t)\| = 1$ *for all* $t \geq 0$*.*

In light of Lemma 2.1 this is to say that, if the Oja flow starts with a unit vector, the vector it converges to will be a unit eigenvector of matrix $\boldsymbol{C}$.

Third of all, it is easy to see why the Oja flow is said to be a flow because, as shown in the appendix, we have

Lemma 2.3. *The Oja flow is a gradient flow.*

In order to unpack this statement, we briefly recall the notion of a **gradient flow**: Given a vector space V and a smooth function $f : V \to \mathbb{R}$, a gradient flow is a smooth curve $\boldsymbol{x} : \mathbb{R} \to V$, $t \mapsto \boldsymbol{x}(t)$ such that $\frac{d}{dt}\,\boldsymbol{x}(t) = -\nabla f\big(\boldsymbol{x}(t)\big)$.

Listing 1: Numerically integrating, i.e. solving, the Oja flow

```
 1  def integrateOjaFlow(matX, tmax=4, nsteps=101):
 2
 3      def derivative(w, t, C, I):
 4          return (I - np.outer(w, w)) @ C @ w
 5
 6      m, n = matX.shape
 7
 8      matI = np.eye(m)
 9      matC = np.cov(matX)
10
11      vecW0 = np.ones(m) / np.sqrt(m)
12      steps = np.linspace(0, tmax, nsteps)
13
14      matW = odeint(derivative, vecW0, steps, args=(matC, matI))
15
16      return matW
```

With respect to the claim in Lemma 2.3, this is to say that there must exist a function $f\big(\boldsymbol{w}(t)\big)$ such that

$$\frac{d\boldsymbol{w}}{dt} = -\nabla f(\boldsymbol{w}) \tag{16}$$

which is interesting because it tells us that Oja's rule in (4) is nothing but a gradient descent scheme.

3 PRACTICAL COMPUTATION

Having discussed theoretical properties of the Oja flow, the obvious question is, if we could actually use it to compute principal components of a data sample?

In what follows, we will work with the expression in (10) which, including the the dependency on time t, reads

$$\tfrac{d}{dt}\,\boldsymbol{w}(t) = \big(\boldsymbol{I} - \boldsymbol{w}(t)\boldsymbol{w}(t)^\intercal\big)\,\boldsymbol{C}\boldsymbol{w}(t) \tag{17}$$

Scrutinizing this expression, we realize that solving the Oja flow means to compute

$$\boldsymbol{w}(t) = \int_0^t \big(\boldsymbol{I} - \boldsymbol{w}(\tau)\boldsymbol{w}(\tau)^\intercal\big)\,\boldsymbol{C}\boldsymbol{w}(\tau)\,d\tau \tag{18}$$

which begs the question of how to solve the integral on the right?

The strategy we adhere to in the following is to use numerical integration. To this end, we will resort to function `odeint` which is available in *SciPy*'s `integrate` module.[1]

Listing 1 shows a function `integrateOjaFlow` which illustrates the use of `odeint` for our purpose. Function `integrateOjaFlow` is called with three parameters `matX`, `tmax`, and `nsteps`.

Parameter `matX` is a 2D *NumPy* array that represents the data matrix $\boldsymbol{X}$ in (1); the roles of the other two parameters will become clear shortly.

At the beginning of `integrateOjaFlow` (in lines 3 and 4), we define a function `derivative` whose role, too, will become clear shortly. For now, we already note that it suspiciously looks like an implementation of the equation of the Oja flow in (10).

In line 6, we determine the size of $\boldsymbol{X}$. The number m of its rows is then used to initialize an array `matI` which represents the $m \times m$

[1] **NOTE:** `odeint` is now considered a legacy function and users of the latest versions of *SciPy* are encouraged to work with `solve_ivp` instead. This function, too, is found in the `integrate` module. However, its API has undergone some changes over the past couple of *SciPy* releases so that discussing its use would entail the risk that readers working with slightly older *SciPy* versions could not run our code. Hence, we stick with "good old" `odeint`.

identity matrix $\boldsymbol{I}$ and line 9 computes an array matC which represents the sample covariance matrix $\boldsymbol{C}$ in (2).

Next, in line 11, we initialize a 1D array vecW0 which represents an m-dimensional unit vector $\boldsymbol{w}(0) = \boldsymbol{w}_0$ that indicates (an arbitrary choice of) the initial value of the Oja flow at time $t = 0$.

If we make use of odeint to numerically integrate a differential equation, we need to specify a sequence of time steps for which to solve the equation. Hence, in line 12, we initialize an array steps of such time steps. The initial point of the sequence is 0, the last point of the sequence is given by the parameter tmax and the number of point in between is given by the parameter nsteps. These are the two additional parameters passed to integrateOjaFlow and we opted to set their default values to 4 and 101, respectively.

Given all these preparations, we can now invoke odeint to solve the Oja flow. This happens in line 14 and we note that, out of the many parameters of odeint, the following ones are of major interest for our current setting:

- the 1st required parameter is a callable object, i.e. a function that computes the differential equation we wish to solve; here we set it to derivative, i.e. the function we defined in lines 3 and 4; we already said that it is an immediate *NumPy* implementation of the differential equation in (10); we also point out that its parameter t does not occur in its body but odeint requires it to be present; finally, we note that the order in which parameters w and t occur in the definition of derivative may seem strange but is another requirement of odeint
- the 2nd required parameter represent the initial condition of the differential equation to be solved; hence, we pass the array vecW0 which we initialized above
- the 3rd mandatory parameter represents the time points at which to solve the differential equation under consideration; here, we therefore pass the array steps
- args is an optional parameter that is only required if the function passed in the first argument has additional parameters (other than w and t); in our case it has, namely matC and matI and so we pass them in a tuple

Used like this, odeint produces an array of shape (nsteps, m) which we store in matW and finally return it as the result of function integrateOjaFlow.

Hence, if we are given a 2D array matX which represents an $m \times n$ data matrix $\boldsymbol{X}$, we may use

```
matW = integrateOjaFlow(matX)
```

to obtain an array whose rows represent the states of the Oja flow at the nsteps time points between 0 and tmax. If tmax is large enough, the last row matW[-1] of array matW represents a vector $\boldsymbol{w}(t_{\max})$ that corresponds to a stable equilibrium of the flow. For instance, for the 3D data points in Fig. 1, we find

```
print (matW[-1])
>>> [-0.79887118  0.50321141  0.32951953]
```

In order to compare this result to eigenvector $\boldsymbol{u}_1$ of the sample covariance matrix that would be computed by standard linear algebra routines, we recall our discussion in [2] and use

```
matC = np.cov(matX)
vecL, matU = la.eigh(matC)
```

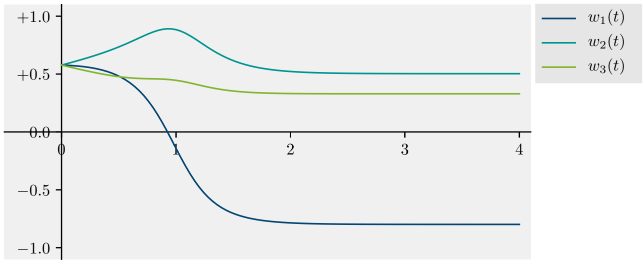

Figure 2: Visualization of an Oja flow computed for the 3D data in Fig. 1. The initial value was the unit vector $\boldsymbol{w}(0) = 1/\sqrt{3}$ and the figure shows how the components $w_1(t)$, $w_2(t)$, and $w_3(t)$ of $\boldsymbol{w}(t)$ evolve over time. In accordance with theoretical expectations [20], the flow quickly reaches a stable point.

in order to obtain an array

```
vecU1 = matU[:,-1]
```

which we then inspect as follows

```
print (vecU1)
>>> [-0.7988784   0.50319912   0.32952077]
```

The minute differences between this result and the outcome of our Oja flow computation can be attributed to numerical imprecision. All in all, our example corroborates the theoretical expectation that the Oja flow converges almost surely to the principal eigenspace of the given data sample [20].

4 CONCLUSION

In this note, we saw that the problem of computing the principal component of a set of data points can be cast as the problem of solving an ordinary differential equation called the Oja flow.

While this is an interesting theoretical result, one may wonder if it provides immediate practical benefits? At this point in time, the answer is a sounding NO, BUT …

On the one hand, we saw that numerical integration methods can solve the Oja flow. But when it comes to computational efficiency, these cannot compete with high performance linear algebra routines for eigenvalue / eigenvector computation. Such routines are implemented in the LAPACK library [1] which is linked by *NumPy* / *SciPy* and we already discussed how to apply their corresponding methods [2]. In this sense, dynamical system models for eigenvalue / eigenvector problems are of little use when working with conventional digital computers.

On the other hand, we are currently witnessing the (re)emergence of next generation computing devices which transcend certain limitations of digital computers. Indeed, it has been known for long that analog computers (as well as special purpose VLSI circuits) can solve differential equations [8, 17]. Since there have been interesting developments in this area [7, 9, 11, 18], the Oja flow may become of practical vlaue. There has also been substantial technical progress in quantum computing and quantum computers, too, can solve differential equations [6, 10, 12, 14]. As they offer exponential advantages over classical computers, this, too, may lead to practical applications of the Oja flow.

In short, since eigenvector computation plays a fundamental role in intelligent data analysis[2], and since the Oja flow allows for computing eigenvectors in a manner that suites next generation computing devices, it may soon take on a much bigger role than it has been playing so far.

A APPENDIX

In the following, we provide the still outstanding proofs of Lemma 2.2 and 2.3. To begin with, we reiterate that we are concerned with a zero mean sample gathered in a data matrix $\boldsymbol{X} = \left[\boldsymbol{x}_1\, \boldsymbol{x}_2 \cdots \boldsymbol{x}_n\right]$ and that the sample covariance matrix can be computed as $\boldsymbol{C} = \frac{1}{n}\boldsymbol{X}\boldsymbol{X}^\top$.

Also, for notational convenience, we will switch from the Leibniz notation to Newton's notation for temporal derivatives and write the Oja flow as

$$\dot{\boldsymbol{w}} = \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{C}\, \boldsymbol{w}$$

Having recalled our basic assumptions and common notational conventions, we can proceed and provide the

Proof of Lemma 2.3. All we need to do to show that the Oja flow is a gradient flow, is to find a function $f(\boldsymbol{w})$ such that

$$\dot{\boldsymbol{w}} = \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{C}\, \boldsymbol{w} = -\nabla f(\boldsymbol{w})$$

To make a rather long story short, we will consider an inspired ansatz, namely

$$h(\boldsymbol{w}) = \left\| \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{X} \right\|^2 \tag{19}$$

Writing this squared Frobenius term in terms of a trace, we have

$$\begin{aligned} \left\| \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{X} \right\|^2 &= \operatorname{tr}\left[\Big(\left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{X} \Big) \Big(\left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{X} \Big)^\top \right] \\ &= \operatorname{tr}\left[\left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{X}\boldsymbol{X}^\top \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right)^\top \right] \\ &= \operatorname{tr}\left[\left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{X}\boldsymbol{X}^\top \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \right] \end{aligned}$$

where the last step is possible because matrix $\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top$ is symmetric.

Given this result, we next consider a slightly modified function, namely

$$\begin{aligned} g(\boldsymbol{w}) = \tfrac{1}{n}\, h(\boldsymbol{w}) &= \tfrac{1}{n} \operatorname{tr}\left[\left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{X}\boldsymbol{X}^\top \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \right] \\ &= \operatorname{tr}\left[\left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \tfrac{1}{n} \boldsymbol{X}\boldsymbol{X}^\top \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \right] \\ &= \operatorname{tr}\left[\left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{C} \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \right] \end{aligned}$$

Next, we consider the gradient of $g(\boldsymbol{w})$ and note that, if we want to compute

$$\nabla g = \frac{dg}{d\boldsymbol{w}}$$

we must pay attention to the fact that g is a function of the form $g(\boldsymbol{w}) = \operatorname{tr}\left[\boldsymbol{M}(\boldsymbol{w})\, \boldsymbol{C}\, \boldsymbol{M}(\boldsymbol{w})\right]$. In other words, we have to invoke the chain rule

$$\frac{dg}{d\boldsymbol{w}} = \frac{dg}{d\boldsymbol{M}} \cdot \frac{d\boldsymbol{M}}{d\boldsymbol{w}}$$

To make another long story short[3], the sought after gradient amounts to

$$\frac{dg}{d\boldsymbol{w}} = 2\left(\left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{C} \right) \cdot (-2)\, \boldsymbol{w} = -4 \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{C}\, \boldsymbol{w}$$

But this is finally to say that, if we introduce yet another slightly modified function, namely

$$f(\boldsymbol{w}) = \tfrac{1}{4}\, g(\boldsymbol{w})$$

there does indeed exist a function such that

$$\dot{\boldsymbol{w}} = \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{C}\, \boldsymbol{w} = -\nabla f(\boldsymbol{w})$$

□

Having proved the crucial Lemma 2.3 and seen the techniques this involved, it is now easy to provide the following simple

Proof of Lemma 2.2. To show that the Oja flow is isometric when started with a vector $\boldsymbol{w}$ where $\|\boldsymbol{w}\| = 1$, we need to show that the length of $\boldsymbol{w}$ is an invariant of the process

$$\dot{\boldsymbol{w}} = \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{C}\, \boldsymbol{w}$$

To this end, we consider the temporal derivative of the function

$$L(\boldsymbol{w}) = \|\boldsymbol{w}\|^2 = \boldsymbol{w}^\top \boldsymbol{w} \tag{20}$$

and find

$$\begin{aligned} \frac{dL}{dt} = \frac{dL}{d\boldsymbol{w}} \frac{d\boldsymbol{w}}{dt} &= 2\, \boldsymbol{w}^\top \dot{\boldsymbol{w}} \\ &= 2\, \boldsymbol{w}^\top \left(\boldsymbol{I} - \boldsymbol{w}\boldsymbol{w}^\top\right) \boldsymbol{C}\, \boldsymbol{w} \\ &= 2\, \boldsymbol{w}^\top \boldsymbol{C}\, \boldsymbol{w} - 2\, \boldsymbol{w}^\top \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{C}\, \boldsymbol{w} \end{aligned}$$

Since we assumed that $\boldsymbol{w}$ is a unit vector for which $\boldsymbol{w}^\top \boldsymbol{w} = 1$, this expression further simplifies to

$$\frac{dL}{dt} = 2\, \boldsymbol{w}^\top \boldsymbol{C}\, \boldsymbol{w} - 2\, \boldsymbol{w}^\top \boldsymbol{C}\, \boldsymbol{w} = 0$$

In other words, if $\boldsymbol{w}$ is a unit vector that evolves under the Oja flow, its direction may change over time but its length will not. □

ACKNOWLEDGMENTS

This material was produced within the Competence Center for Machine Learning Rhine-Ruhr (**ML2R**) which is funded by the Federal Ministry of Education and Research of Germany (grant no. 01|S18038C). The authors gratefully acknowledge this support.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (3rd ed.). SIAM.

[2] C. Bauckhage. 2015. NumPy / SciPy Recipes for Data Science: Eigenvalues / Eigenvectors of Covariance Matrices. researchgate.net. https://dx.doi.org/10.13140/RG.2.1.2307.5046.

[3] C. Bauckhage, F. Beaumont, and S. Müller. 2021. *ML2R Coding Nuggets: Principal Components via Oja's Rule.* Technical Report. MLAI, University of Bonn.

[4] C. Bauckhage, A. Drachen, and R. Sifa. 2015. Clustering Game Behavior Data. *IEEE Trans. on Computational Intelligence and AI in Games* 7, 3 (2015).

[5] C. Bauckhage and K. Kersting. 2013. Data Mining and Pattern Recognition in Agriculture. *KI – Künstliche Intelligenz* 27, 4 (2013).

[6] D.W. Berry. 2014. High-order Quantum Algorithm for Solving Linear Differential Equations. *J. of Physics A: Mathematical and Theoretical* 47, 10 (2014).

[2] Here are but a few examples from our own work which illustrate the wide range of practical applications: [4, 5, 13, 19].

[3] The *Matrix Cookbook* [16] is a great resource for looking up multi-variate calculus identities.

[7] O. Bournez and A. Pouly. 2021. A Survey on Analog Models of Computation. In *Handbook of Computability and Complexity in Analysis*, V. Brattka and P. Hertling (Eds.). Springer. to appear.

[8] R.W. Brockett. 1992. Analog and Digital Computing. In *Future Tendencies in Computer Science, Control and Applied Mathematics*, A. Bensoussan and J.P. Verjus (Eds.). LNCS, Vol. 653. Springer.

[9] A. Celik, M. Stanacevic, and G. Cauwenberghs. 2005. Gradient Flow Independent Component Analysis in Micropower VLSI. In *Proc. NIPS*.

[10] L. Franken, B. Georgiev, S. Muecke, M. Wolter, N. Piatkowski, and C. Bauckhage. 2020. Gradient-free Quantum Optimization on NISQ Devices. *arXiv:2012.13453 [quant-ph]* (2020).

[11] D. Fu, S. Shah, T. Song, and J. Reif. 2018. DNA-Based Analog Computing. In *Synthetic Biology*, J. Braman (Ed.). Humana Press.

[12] B.T. Kiani, G. De Palma, D. Englund, W. Kaminsky, M. Marvian, and S. Lloyd. 2020. Quantum Advantage for Differential Equation Analysis. *arXiv:2010.15776 [quant-ph]* (2020).

[13] J. Kunegis, D. Fay, and C. Bauckhage. 2010. Network Growth and the Spectral Evolution Model. In *Proc. CIKM*. ACM.

[14] S. Lloyd, G. De Palma, C. Gokler, B. Kiani, Z.-W. Liu, M. Marvian, F. Tennie, and T. Palmer. 2020. Quantum Algorithm for Nonlinear Differential Equations. *arXiv:2011.06571 [quant-ph]* (2020).

[15] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).

[16] K. B. Petersen and M. S. Pedersen. 2012. *The Matrix Cookbook*. Technical University of Denmark.

[17] H.T. Siegelmann and S. Fishman. 1998. Analog Computation with Dynamical Systems. *Physica D* 120, 1–2 (1998).

[18] D. Solli and B. Jalali. 2015. Analog Optical Computing. *Nature Photonics* 9 (2015).

[19] L. von Rueden, S. Mayer, K. Beckh, B. Georgiev, S. Giesselbach, R. Heese, B. Kirsch, J. Pfrommer, A. Pick, R. Ramamurthy, M. Walczak, J. Garcke, C. Bauckhage, and J. Schuecker. 2019. Informed Machine Learning – A Taxonomy and Survey of Integrating Knowledge into Learning Systems. *arXiv:1903.12394 [stat.ML]* (2019).

[20] W.-Y. Yan, U. Helmke, and B. Moore. 1994. Global Analysis of Oja's Flow for Neural Networks. *IEEE Trans. on Neural Networks* 5, 5 (1994).