

# March 4-8: Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen<sup>1,2</sup>

<sup>1</sup>Department of Physics and Center for Computing in Science Education, University of Oslo, Norway

<sup>2</sup>Department of Physics and Astronomy and Facility for Rare Isotope Beams, Michigan State University, East Lansing, Michigan, USA

March 5, 2024

## Plans for the week March 4-8

1. RNNs and discussion of Long-Short-Term memory
2. Discussion of specific examples relevant for project 1, [see project from last year by Daniel and Keran](#)
3. Start discussion of Autoencoders (AEs) if we get time.

## Reading recommendations

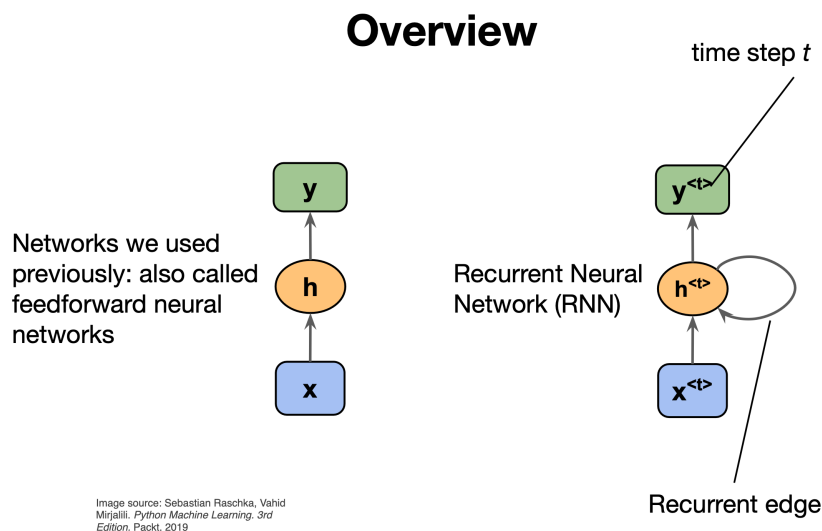
1. For RNNs see Goodfellow et al chapter 10.
2. For AEs, see chapter 14 of same book
3. Reading suggestions for implementation of RNNs in PyTorch: Rashcka et al's text, chapter 15
4. Reading suggestions for implementation of RNNs in TensorFlow: [Aurelien Geron's chapter 14](#).

## RNNs

Recurrent neural networks (RNNs) have in general no probabilistic component in a model. With a given fixed input and target from data, the RNNs learn the intermediate association between various layers. The inputs, outputs, and internal representation (hidden states) are all real-valued vectors.

In a traditional NN, it is assumed that every input is independent of each other. But with sequential data, the input at a given stage  $t$  depends on the input from the previous stage  $t - 1$

**Basic layout, Figures from Sebastian Raschka et al, Machine learning with Sickit-Learn and PyTorch**



## Solving differential equations with RNNs

To gain some intuition on how we can use RNNs for time series, let us tailor the representation of the solution of a differential equation as a time series.

Consider the famous differential equation (Newton's equation of motion for damped harmonic oscillations, scaled in terms of dimensionless time)

$$\frac{d^2x}{dt^2} + \eta \frac{dx}{dt} + x(t) = F(t),$$

where  $\eta$  is a constant used in scaling time into a dimensionless variable and  $F(t)$  is an external force acting on the system. The constant  $\eta$  is a so-called damping.

## Two first-order differential equations

In solving the above second-order equation, it is common to rewrite it in terms of two coupled first-order equations with the velocity

$$v(t) = \frac{dx}{dt},$$

and the acceleration

$$\frac{dv}{dt} = F(t) - \eta v(t) - x(t).$$

With the initial conditions  $v_0 = v(t_0)$  and  $x_0 = x(t_0)$  defined, we can integrate these equations and find their respective solutions.

## Velocity only

Let us focus on the velocity only. Discretizing and using the simplest possible approximation for the derivative, we have Euler's forward method for the updated velocity at a time step  $i + 1$  given by

$$v_{i+1} = v_i + \Delta t \frac{dv}{dt} \Big|_{v=v_i} = v_i + \Delta t (F_i - \eta v_i - x_i).$$

Defining a function

$$h_i(x_i, v_i, F_i) = v_i + \Delta t (F_i - \eta v_i - x_i),$$

we have

$$v_{i+1} = h_i(x_i, v_i, F_i).$$

## Linking with RNNs

The equation

$$v_{i+1} = h_i(x_i, v_i, F_i).$$

can be used to train a feed-forward neural network with inputs  $v_i$  and outputs  $v_{i+1}$  at a time  $t_i$ . But we can think of this also as a recurrent neural network with inputs  $v_i$ ,  $x_i$  and  $F_i$  at each time step  $t_i$ , and producing an output  $v_{i+1}$ .

Noting that

$$v_i = v_{i-1} + \Delta t (F_{i-1} - \eta v_{i-1} - x_{i-1}) = h_{i-1}.$$

we have

$$v_i = h_{i-1}(x_{i-1}, v_{i-1}, F_{i-1}),$$

and we can rewrite

$$v_{i+1} = h_i(x_i, h_{i-1}, F_i).$$

## Minor rewrite

We can thus set up a recurring series which depends on the inputs  $x_i$  and  $F_i$  and the previous values  $h_{i-1}$ . We assume now that the inputs at each step (or time  $t_i$ ) is given by  $x_i$  only and we denote the outputs for  $\tilde{y}_i$  instead of  $v_{i1}$ , we have then the compact equation for our outputs at each step  $t_i$

$$y_i = h_i(x_i, h_{i-1}).$$

We can think of this as an element in a recurrent network where our network (our model) produces an output  $y_i$  which is then compared with a target value through a given cost/loss function that we optimize. The target values at a given step  $t_i$  could be the results of a measurement or simply the analytical results of a differential equation.

## RNNs in more detail

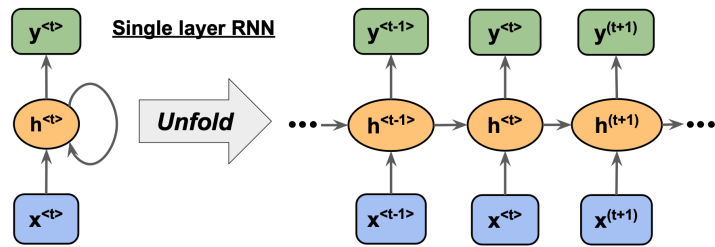
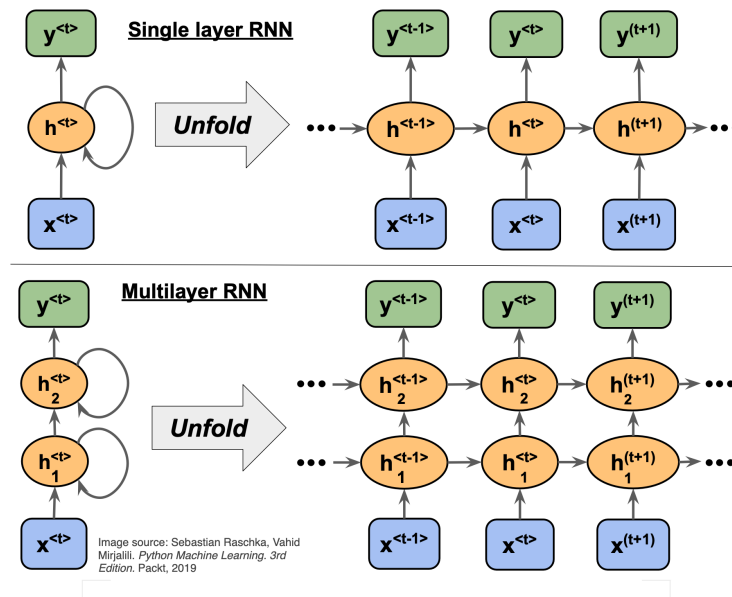


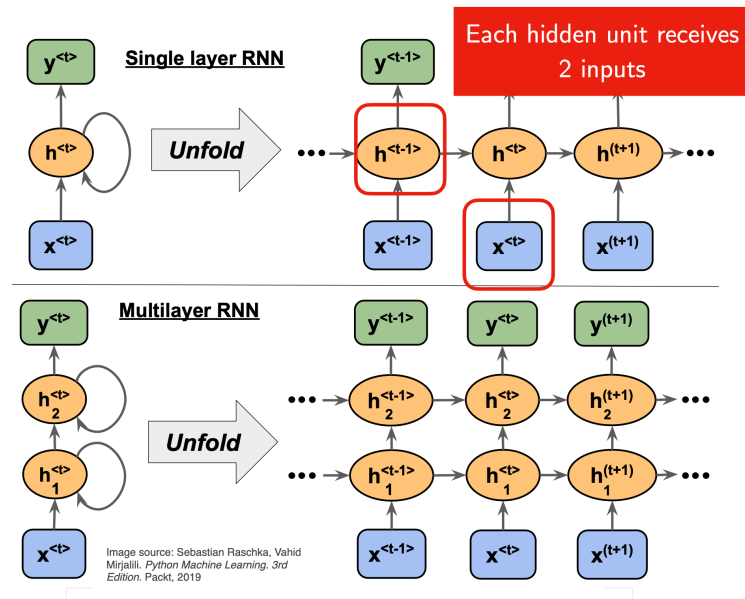
Image source: Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning*, 3rd Edition. Packt, 2019

## RNNs in more detail, part 2

### Overview

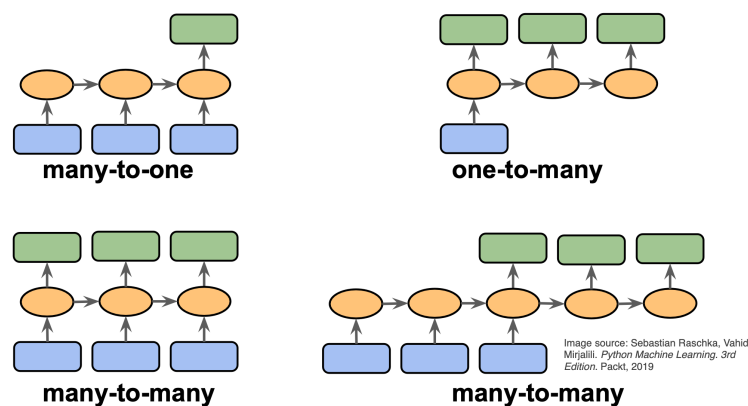


## RNNs in more detail, part 3



## RNNs in more detail, part 4

### Different Types of Sequence Modeling Tasks



## RNNs in more detail, part 5

### Weight matrices in a single-hidden layer RNN

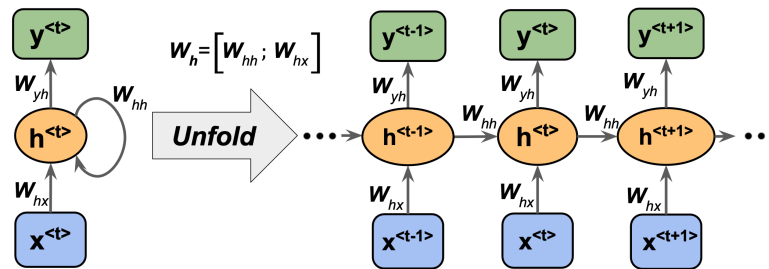


Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning, 3rd Edition, Packt, 2019

## RNNs in more detail, part 6

### Weight matrices in a single-hidden layer RNN

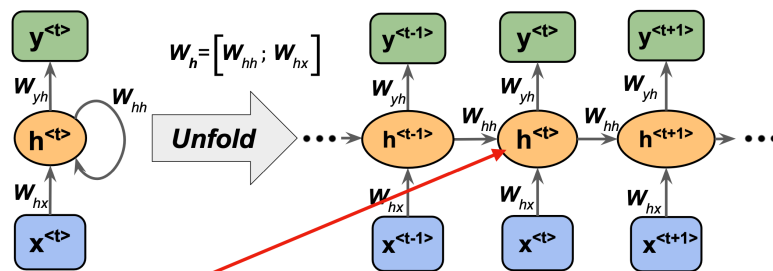


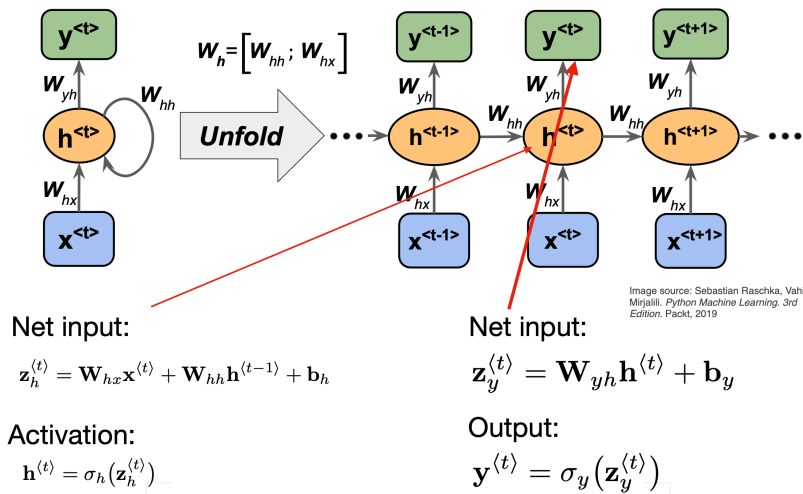
Image source: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning, 3rd Edition, Packt, 2019

Net input: 
$$\mathbf{z}_h^{(t)} = \mathbf{W}_{hx} \mathbf{x}^{(t)} + \mathbf{W}_{hh} \mathbf{h}^{(t-1)} + \mathbf{b}_h$$

Activation: 
$$\mathbf{h}^{(t)} = \sigma_h(\mathbf{z}_h^{(t)})$$

## RNNs in more detail, part 7

### Weight matrices in a single-hidden layer RNN



### Backpropagation through time

We can think of the recurrent net as a layered, feed-forward net with shared weights and then train the feed-forward net with weight constraints.

We can also think of this training algorithm in the time domain:

1. The forward pass builds up a stack of the activities of all the units at each time step.
2. The backward pass peels activities off the stack to compute the error derivatives at each time step.
3. After the backward pass we add together the derivatives at all the different times for each weight.

### The backward pass is linear

1. There is a big difference between the forward and backward passes.
2. In the forward pass we use squashing functions (like the logistic) to prevent the activity vectors from exploding.
3. The backward pass, is completely linear. If you double the error derivatives at the final layer, all the error derivatives will double.



The forward pass determines the slope of the linear function used for backpropagating through each neuron

## The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
  1. If the weights are small, the gradients shrink exponentially.
  2. If the weights are big the gradients grow exponentially.
- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.
- In an RNN trained on long sequences (e.g. 100 time steps) the gradients can easily explode or vanish.
  1. We can avoid this by initializing the weights very carefully.
- Even with good initial weights, its very hard to detect that the current target output depends on an input from many time-steps ago.

RNNs have difficulty dealing with long-range dependencies.

## Mathematical setup

The expression for the simplest Recurrent network resembles that of a regular feed-forward neural network, but now with the concept of temporal dependencies

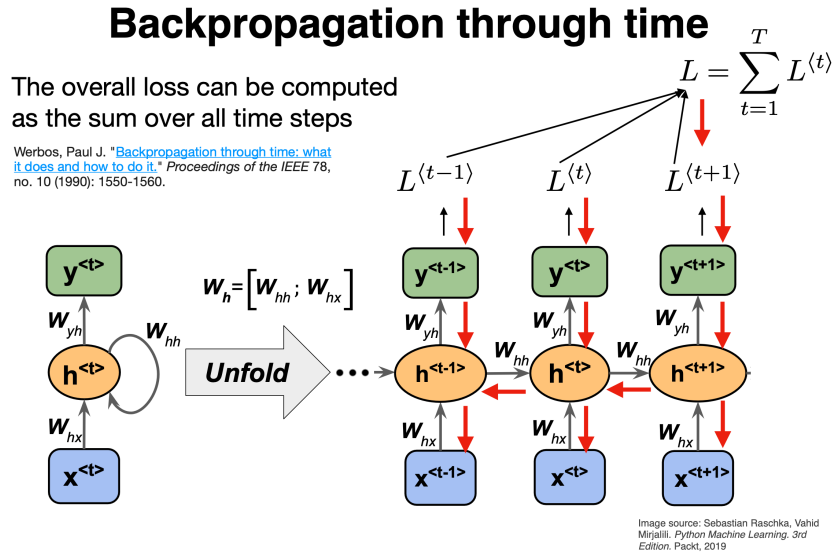
$$\mathbf{a}^{(t)} = U * \mathbf{x}^{(t)} + W * \mathbf{h}^{(t-1)} + \mathbf{b},$$

$$\mathbf{h}^{(t)} = \sigma_h(\mathbf{a}^{(t)}),$$

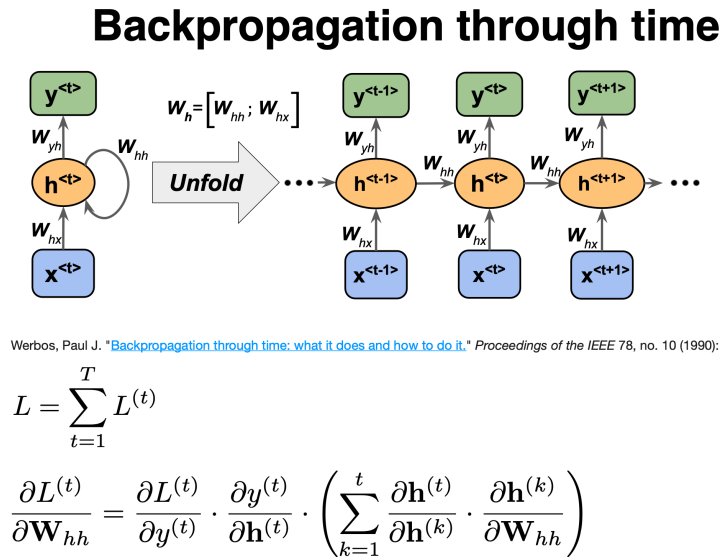
$$\mathbf{y}^{(t)} = V * \mathbf{h}^{(t)} + \mathbf{c},$$

$$\hat{\mathbf{y}}^{(t)} = \sigma_y(\mathbf{y}^{(t)}).$$

## Back propagation in time through figures, part 1

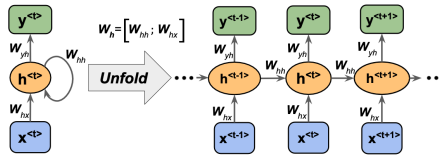


## Back propagation in time, part 2



## Back propagation in time, part 3

### Backpropagation through time



Werbos, Paul J. "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* 78, no. 10 (1990): 1550-1560.

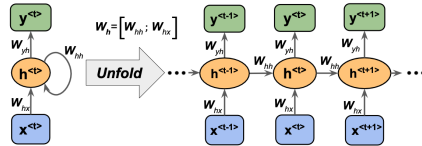
$$\frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial \mathbf{h}^{(t)}} \cdot \left( \sum_{k=1}^t \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \cdot \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right)$$

computed as a multiplication of adjacent time steps:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^t \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

## Back propagation in time, part 4

### Backpropagation through time



Werbos, Paul J. "Backpropagation through time: what it does and how to do it." *Proceedings of the IEEE* 78, no. 10 (1990): 1550-1560.

$$L = \sum_{t=1}^T L^{(t)} \quad \frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial y^{(t)}} \cdot \frac{\partial y^{(t)}}{\partial \mathbf{h}^{(t)}} \cdot \left( \sum_{k=1}^t \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \cdot \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right)$$

computed as a multiplication of adjacent time steps:

This is very problematic:  
Vanishing/Exploding gradient problem!

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^t \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

## Back propagation in time in equations

To derive the expression of the gradients of  $\mathcal{L}$  for the RNN, we need to start recursively from the nodes closer to the output layer in the temporal unrolling scheme - such as  $\mathbf{y}$  and  $\mathbf{h}$  at final time  $t = \tau$ ,

$$\begin{aligned}(\nabla_{\mathbf{y}^{(t)}} \mathcal{L})_i &= \frac{\partial \mathcal{L}}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial y_i^{(t)}}, \\ \nabla_{\mathbf{h}^{(\tau)}} \mathcal{L} &= \mathbf{V}^\top \nabla_{\mathbf{y}^{(\tau)}} \mathcal{L}.\end{aligned}$$

## Chain rule again

For the following hidden nodes, we have to iterate through time, so by the chain rule,

$$\nabla_{\mathbf{h}^{(t)}} \mathcal{L} = \left( \frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \right)^\top \nabla_{\mathbf{h}^{(t+1)}} \mathcal{L} + \left( \frac{\partial \mathbf{y}^{(t)}}{\partial \mathbf{h}^{(t)}} \right)^\top \nabla_{\mathbf{y}^{(t)}} \mathcal{L}.$$

## Gradients of loss functions

Similarly, the gradients of  $\mathcal{L}$  with respect to the weights and biases follow,

$$\begin{aligned}\nabla_{\mathbf{c}} \mathcal{L} &= \sum_t \left( \frac{\partial \mathbf{y}^{(t)}}{\partial \mathbf{c}} \right)^\top \nabla_{\mathbf{y}^{(t)}} \mathcal{L} \\ \nabla_{\mathbf{b}} \mathcal{L} &= \sum_t \left( \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{b}} \right)^\top \nabla_{\mathbf{h}^{(t)}} \mathcal{L} \\ \nabla_{\mathbf{v}} \mathcal{L} &= \sum_t \sum_i \left( \frac{\partial \mathcal{L}}{\partial y_i^{(t)}} \right) \nabla_{\mathbf{v}^{(t)}} y_i^{(t)} \\ \nabla_{\mathbf{w}} \mathcal{L} &= \sum_t \sum_i \left( \frac{\partial \mathcal{L}}{\partial h_i^{(t)}} \right) \nabla_{\mathbf{w}^{(t)}} h_i^{(t)} \\ \nabla_{\mathbf{u}} \mathcal{L} &= \sum_t \sum_i \left( \frac{\partial \mathcal{L}}{\partial h_i^{(t)}} \right) \nabla_{\mathbf{u}^{(t)}} h_i^{(t)}.\end{aligned}$$

## Summary of RNNs

Recurrent neural networks (RNNs) have in general no probabilistic component in a model. With a given fixed input and target from data, the RNNs learn the intermediate association between various layers. The inputs, outputs, and internal representation (hidden states) are all real-valued vectors.

In a traditional NN, it is assumed that every input is independent of each other. But with sequential data, the input at a given stage  $t$  depends on the input from the previous stage  $t - 1$

## Summary of a typical RNN

1. Weight matrices  $U$ ,  $W$  and  $V$  that connect the input layer at a stage  $t$  with the hidden layer  $h_t$ , the previous hidden layer  $h_{t-1}$  with  $h_t$  and the hidden layer  $h_t$  connecting with the output layer at the same stage and producing an output  $\tilde{y}_t$ , respectively.
2. The output from the hidden layer  $h_t$  is often modulated by a tanh function  $h_t = \sigma_h(x_t, h_{t-1}) = \tanh(Ux_t + Wh_{t-1} + b)$  with  $b$  a bias value
3. The output from the hidden layer produces  $\tilde{y}_t = \sigma_y(Vh_t + c)$  where  $c$  is a new bias parameter.
4. The output from the training at a given stage is in turn compared with the observation  $y_t$  through a chosen cost function.

The function  $g$  can any of the standard activation functions, that is a Sigmoid, a Softmax, a ReLU and other. The parameters are trained through the so-called back-propagation through time (BPTT) algorithm.

## Four effective ways to learn an RNN and preparing for next week

1. Long Short Term Memory Make the RNN out of little modules that are designed to remember values for a long time.
2. Hessian Free Optimization: Deal with the vanishing gradients problem by using a fancy optimizer that can detect directions with a tiny gradient but even smaller curvature.
3. Echo State Networks: Initialize the input a hidden and hidden-hidden and output-hidden connections very carefully so that the hidden state has a huge reservoir of weakly coupled oscillators which can be selectively driven by the input.
  - ESNs only need to learn the hidden-output connections.
4. Good initialization with momentum Initialize like in Echo State Networks, but then learn all of the connections using momentum

## Gating mechanism: Long Short Term Memory (LSTM)

Besides a simple recurrent neural network layer, as discussed above, there are two other commonly used types of recurrent neural network layers: Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU). For a short introduction to these layers see <https://medium.com/mindboard/lstm-vs-gru-experimental-comparison-955820c21e8b> and <https://medium.com/mindboard/lstm-vs-gru-experimental-comparison-955820c21e8b>.

LSTM uses a memory cell for modeling long-range dependencies and avoid vanishing gradient problems. Capable of modeling longer term dependencies by having memory cells and gates that controls the information flow along with the memory cells.

1. Introduced by Hochreiter and Schmidhuber (1997) who solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps).
2. They designed a memory cell using logistic and linear units with multiplicative interactions.
3. Information gets into the cell whenever its “write” gate is on.
4. The information stays in the cell so long as its **keep** gate is on.
5. Information can be read from the cell by turning on its **read** gate.

## Implementing a memory cell in a neural network

To preserve information for a long time in the activities of an RNN, we use a circuit that implements an analog memory cell.

1. A linear unit that has a self-link with a weight of 1 will maintain its state.
2. Information is stored in the cell by activating its write gate.
3. Information is retrieved by activating the read gate.
4. We can backpropagate through this circuit because logistics are have nice derivatives.

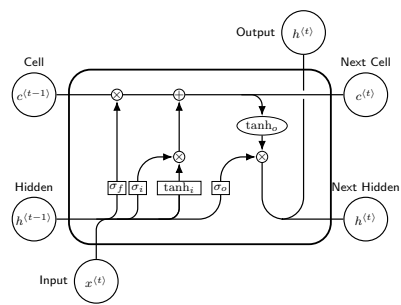
## LSTM details

The LSTM is a unit cell that is made of three gates:

1. the input gate,
2. the forget gate,
3. and the output gate.

It also introduces a cell state  $c$ , which can be thought of as the long-term memory, and a hidden state  $h$  which can be thought of as the short-term memory.

## Basic layout



## More LSTM details

The first stage is called the forget gate, where we combine the input at (say, time  $t$ ), and the hidden cell state input at  $t - 1$ , passing it through the Sigmoid activation function and then performing an element-wise multiplication, denoted by  $\otimes$ .

It follows

$$\mathbf{f}^{(t)} = \sigma(W_f \mathbf{x}^{(t)} + U_f \mathbf{h}^{(t-1)} + \mathbf{b}_f)$$

where  $W$  and  $U$  are the weights respectively.

## The forget gate

This is called the forget gate since the Sigmoid activation function's outputs are very close to 0 if the argument for the function is very negative, and 1 if the argument is very positive. Hence we can control the amount of information we want to take from the long-term memory.

## Input gate

The next stage is the input gate, which consists of both a Sigmoid function ( $\sigma_i$ ), which decide what percentage of the input will be stored in the long-term memory, and the  $\tanh_i$  function, which decide what is the full memory that can be stored in the long term memory. When these results are calculated and multiplied together, it is added to the cell state or stored in the long-term memory, denoted as  $\oplus$ .

We have

$$\mathbf{i}^{(t)} = \sigma_g(W_i \mathbf{x}^{(t)} + U_i \mathbf{h}^{(t-1)} + \mathbf{b}_i),$$

and

$$\tilde{\mathbf{c}}^{(t)} = \tanh(W_c \mathbf{x}^{(t)} + U_c \mathbf{h}^{(t-1)} + \mathbf{b}_c),$$

again the  $W$  and  $U$  are the weights.

## Forget and input

The forget gate and the input gate together also update the cell state with the following equation,

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \otimes \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \otimes \tilde{\mathbf{c}}^{(t)},$$

where  $f^{(t)}$  and  $i^{(t)}$  are the outputs of the forget gate and the input gate, respectively.

## Output gate

The final stage of the LSTM is the output gate, and its purpose is to update the short-term memory. To achieve this, we take the newly generated long-term memory and process it through a hyperbolic tangent ( $\tanh$ ) function creating a potential new short-term memory. We then multiply this potential memory by



the output of the Sigmoid function ( $\sigma_o$ ). This multiplication generates the final output as well as the input for the next hidden cell ( $h^{(t)}$ ) within the LSTM cell.

We have

$$\begin{aligned}\mathbf{o}^{(t)} &= \sigma_g(W_o \mathbf{x}^{(t)} + U_o \mathbf{h}^{(t-1)} + \mathbf{b}_o), \\ \mathbf{h}^{(t)} &= \mathbf{o}^{(t)} \otimes \sigma_h(\mathbf{c}^{(t)}).\end{aligned}$$

where  $\mathbf{W}_o, \mathbf{U}_o$  are the weights of the output gate and  $\mathbf{b}_o$  is the bias of the output gate.

## Differential equations

The examples here are taken from the project of Keran Chen and Daniel Haas Beccatini Lima from 2023.

The dynamics of a stable spiral evolve in such a way that the system's trajectory converges to a fixed point while spiraling inward. These oscillations around the fixed point are gradually dampened until the system reaches a steady state at a fixed point. Suppose we have a two-dimensional system of coupled differential equations of the form

$$\begin{aligned}\frac{dx}{dt} &= ax + by, \\ \frac{dy}{dt} &= cx + dy.\end{aligned}$$

The choice of  $a, b, c, d \in \mathbb{R}$  completely determines the behavior of the solution, and for some of these values, albeit not all, the system is said to be a stable spiral. This condition is satisfied when the eigenvalues of the matrix formed by the coefficients are complex conjugates with a negative real part.

## Lorenz attractor

A Lorenz attractor presents some added complexity. It exhibits what is called a chaotic behavior and its behavior is extremely sensitive to initial conditions.

The expression for the Lorenz attractor evolution consists of a set of three coupled nonlinear differential equations given by

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(\rho - z) - y, \\ \frac{dz}{dt} &= xy - \beta z.\end{aligned}$$

For this problem,  $x, y, z$  are the variables that determine the state of the system in the space while  $\sigma, \rho$  and  $\beta$  are, similarly to the constants  $a, b, c, d$  of the stable spiral, parameters that influence largely how the system evolves.

## Generating data

Both of the above-mentioned systems are governed by differential equations, and as such, they can be solved numerically through some integration scheme such as forward-Euler or fourth-order Runge-Kutta.

We use the common choice of parameters  $\sigma = 10$ ,  $\rho = 28$ ,  $\beta = 8/3$ . This choice generates complex and aesthetic trajectories that have been extensively investigated and benchmarked in the literature of numerical simulations.

For the stable spiral, we employ  $a = 0.2$ ,  $b = -1.0$ ,  $c = 1.0$ ,  $d = 0.2$ . This gives a good number of oscillations before reaching a steady state.

## Training and testing

Training and testing procedures in recurrent neural networks follow what is usual for regular FNNs, but some special consideration needs to be taken into account due to the sequential character of the data. **Training and testing batches must not be randomly shuffled** for it would clearly decorrelate the time-series points and leak future information into present or past points of the model.

## Computationally expensive

The training algorithm can become computationally costly, especially if the losses are evaluated for all previous time steps. While other architectures such as that of LSTMs can be used to mitigate that, it is also possible to introduce another hyperparameter responsible for controlling how much of the network will be unfolded in the training process, adjusting how much the network will remember from previous points in time. Similarly, the number of steps the network predicts in the future per iteration greatly influences the assessment of the loss function.

## Choice of training data

The training and testing batches were separated into whole trajectories. This means that instead of training and testing on different fractions of the same trajectory, all trajectories that were tested had completely new initial conditions. In this sense, from a total of 10 initial conditions (independent trajectories), 9 were used for training and 1 for testing. Each trajectory consisted of 800 points in each space coordinate.

## Cost/Loss function

The problem we have is a time-series forecasting problem, so, we are free to choose the loss function amongst the big collection of regression losses. Using the mean-squared error of the predicted versus factual trajectories of the dynamic systems is a natural choice.

It is a convex function, so given sufficient time and appropriate learning rates, it is guaranteed to converge to global minima irrespective of the weights random initializations.

$$\mathcal{L}_{MSE} = \frac{1}{N} \sum_i^N (y(\mathbf{x}_i) - \hat{y}(\mathbf{x}_i, \theta))^2 \quad (1)$$

where  $\theta$  represents the set of all parameters of the network, and  $\mathbf{x}_i$  are the input values

### Modifying the cost/loss function, adding more info

A cost/loss function that is based on the observational and predicted data, is normally referred to as a purely data-driven approach.

While this is a well-established way of assessing regressions, it does not make use of other intuitions we might have over the problem we are trying to solve. At the same time, it is a well-established fact that neural network models are data-greedy - they need large amounts of data to be able to generalize predictions outside the training set. One way to try to mitigate this is by using physics-informed neural networks (PINNs) when possible.

### Changing the function to optimize

Trying to improve the performance of our model beyond training sets, PINNs then add physics-informed penalties to the loss function. In essence, this means that we add a worse evaluation score to predictions that do not respect physical laws we think our real data should obey. This procedure often has the advantage of trimming the parameter space without adding bias to the model if the constraints imposed are correct, but the choice of the physical laws can be a delicate one.

### Adding more information to the loss function

A general way of expressing this added penalty to the loss function is shown here

$$\mathcal{L} = w_{MSE} \mathcal{L}_{MSE} + w_{PI} \mathcal{L}_{PI}.$$

Here, the weights  $w_{MSE}$  and  $w_{PI}$  explicitly mediate how much influence the specific parts of the total loss function should contribute. See the above project link for more details.

### Autoencoders: Overarching view

Autoencoders are artificial neural networks capable of learning efficient representations of the input data (these representations are called codings) without any supervision (i.e., the training set is unlabeled). These codings typically have a

much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction.

More importantly, autoencoders act as powerful feature detectors, and they can be used for unsupervised pretraining of deep neural networks.

Lastly, they are capable of randomly generating new data that looks very similar to the training data; this is called a generative model. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces. Surprisingly, autoencoders work by simply learning to copy their inputs to their outputs. This may sound like a trivial task, but we will see that constraining the network in various ways can make it rather difficult. For example, you can limit the size of the internal representation, or you can add noise to the inputs and train the network to recover the original inputs. These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data. In short, the codings are byproducts of the autoencoder's attempt to learn the identity function under some constraints.

## First introduction of AEs

Autoencoders were first introduced by Rumelhart, Hinton, and Williams in 1986 with the goal of learning to reconstruct the input observations with the lowest error possible.

Why would one want to learn to reconstruct the input observations? If you have problems imagining what that means, think of having a dataset made of images. An autoencoder would be an algorithm that can give as output an image that is as similar as possible to the input one. You may be confused, as there is no apparent reason of doing so. To better understand why autoencoders are useful we need a more informative (although not yet unambiguous) definition.

An autoencoder is a type of algorithm with the primary purpose of learning an "informative" representation of the data that can be used for different applications (see [Bank, D., Koenigstein, N., and Giryes, R., Autoencoders](#)) by learning to reconstruct a set of input observations well enough.

## Autoencoder structure

Autoencoders are neural networks where the outputs are its own inputs. They are split (see whiteboard drawing) into an **encoder part** which maps the input  $\mathbf{x}$  via a function  $f(\mathbf{x}, \mathbf{W})$  (this is the encoder part) to a **so-called code part** (or intermediate part) with the result  $\mathbf{h}$

$$\mathbf{h} = f(\mathbf{x}, \mathbf{W}),$$

where  $\mathbf{W}$  are the weights to be determined. The **decoder** parts maps, via its own parameters (weights given by the matrix  $\mathbf{V}$  and its own biases) to the final output

$$\tilde{\mathbf{x}} = g(\mathbf{h}, \mathbf{V}).$$

The goal is to minimize the construction error.

## Dimensionality reduction and links with Principal component analysis

The hope is that the training of the autoencoder can unravel some useful properties of the function  $f$ . They are often trained with only single-layer neural networks (although deep networks can improve the training) and are essentially given by feed forward neural networks.

If the function  $f$  and  $g$  are given by a linear dependence on the weight matrices  $\mathbf{W}$  and  $\mathbf{V}$ , we can show that for a regression case, by minimizing the mean squared error between  $\mathbf{x}$  and  $\tilde{\mathbf{x}}$ , the autoencoder learns the same subspace as the standard principal component analysis (PCA).

In order to see this, we define then

$$\mathbf{h} = f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x},$$

and

$$\tilde{\mathbf{x}} = g(\mathbf{h}, \mathbf{V}) = \mathbf{V}\mathbf{h} = \mathbf{V}\mathbf{W}\mathbf{x}.$$

## AE mean-squared error

With the above linear dependence we can in turn define our optimization problem in terms of the optimization of the mean-squared error, that is we wish to optimize

$$\min_{\mathbf{W}, \mathbf{V} \in \mathbb{R}} \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \tilde{x}_i)^2 = \frac{1}{n} \|\mathbf{x} - \mathbf{V}\mathbf{W}\mathbf{x}\|_2^2,$$

where we have used the definition of a norm-2 vector, that is

$$\|\mathbf{x}\|_2 = \sqrt{\sum_i x_i^2}.$$

This is equivalent to our functions learning the the same subspace as the PCA method. This means that we can interpret AEs as a dimensionality reduction method. To see this, we need to remind ourselves about the PCA method.

## Basic ideas of the Principal Component Analysis (PCA)

The principal component analysis deals with the problem of fitting a low-dimensional affine subspace  $S$  of dimension  $d$  much smaller than the total dimension  $D$  of the problem at hand (our data set). Mathematically it can be formulated as a statistical problem or a geometric problem. In our discussion of the theorem for the classical PCA, we will stay with a statistical approach. Historically, the PCA was first formulated in a statistical setting in order to estimate the principal component of a multivariate random variable.

We have a data set defined by a design/feature matrix  $\mathbf{X}$  (see below for its definition)

- Each data point is determined by  $p$  extrinsic (measurement) variables
- We may want to ask the following question: Are there fewer intrinsic variables (say  $d \ll p$ ) that still approximately describe the data?
- If so, these intrinsic variables may tell us something important and finding these intrinsic variables is what dimension reduction methods do.

A good read is for example [Vidal, Ma and Sastry](#).

## Introducing the Covariance and Correlation functions

Before we discuss the PCA theorem, we need to remind ourselves about the definition of the covariance and the correlation function. These are quantities

Suppose we have defined two vectors  $\hat{x}$  and  $\hat{y}$  with  $n$  elements each. The covariance matrix  $\mathbf{C}$  is defined as

$$\mathbf{C}[\mathbf{x}, \mathbf{y}] = \begin{bmatrix} \text{cov}[\mathbf{x}, \mathbf{x}] & \text{cov}[\mathbf{x}, \mathbf{y}] \\ \text{cov}[\mathbf{y}, \mathbf{x}] & \text{cov}[\mathbf{y}, \mathbf{y}] \end{bmatrix},$$

where for example

$$\text{cov}[\mathbf{x}, \mathbf{y}] = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})(y_i - \bar{y}).$$

With this definition and recalling that the variance is defined as

$$\text{var}[\mathbf{x}] = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})^2,$$

we can rewrite the covariance matrix as

$$\mathbf{C}[\mathbf{x}, \mathbf{y}] = \begin{bmatrix} \text{var}[\mathbf{x}] & \text{cov}[\mathbf{x}, \mathbf{y}] \\ \text{cov}[\mathbf{x}, \mathbf{y}] & \text{var}[\mathbf{y}] \end{bmatrix}.$$

## More on the covariance

The covariance takes values between zero and infinity and may thus lead to problems with loss of numerical precision for particularly large values. It is common to scale the covariance matrix by introducing instead the correlation matrix defined via the so-called correlation function

$$\text{corr}[\mathbf{x}, \mathbf{y}] = \frac{\text{cov}[\mathbf{x}, \mathbf{y}]}{\sqrt{\text{var}[\mathbf{x}]\text{var}[\mathbf{y}]}.$$

The correlation function is then given by values  $\text{corr}[\mathbf{x}, \mathbf{y}] \in [-1, 1]$ . This avoids eventual problems with too large values. We can then define the correlation matrix for the two vectors  $\mathbf{x}$  and  $\mathbf{y}$  as

$$\mathbf{K}[\mathbf{x}, \mathbf{y}] = \begin{bmatrix} 1 & \text{corr}[\mathbf{x}, \mathbf{y}] \\ \text{corr}[\mathbf{y}, \mathbf{x}] & 1 \end{bmatrix},$$

In the above example this is the function we constructed using **pandas**.

## Reminding ourselves about Linear Regression

In our derivation of the various regression algorithms like **Ordinary Least Squares** or **Ridge regression** we defined the design/feature matrix  $\mathbf{X}$  as

$$\mathbf{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & \dots & \dots & x_{0,p-1} \\ x_{1,0} & x_{1,1} & x_{1,2} & \dots & \dots & x_{1,p-1} \\ x_{2,0} & x_{2,1} & x_{2,2} & \dots & \dots & x_{2,p-1} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ x_{n-2,0} & x_{n-2,1} & x_{n-2,2} & \dots & \dots & x_{n-2,p-1} \\ x_{n-1,0} & x_{n-1,1} & x_{n-1,2} & \dots & \dots & x_{n-1,p-1} \end{bmatrix},$$

with  $\mathbf{X} \in \mathbb{R}^{n \times p}$ , with the predictors/features  $p$  referring to the column numbers and the entries  $n$  being the row elements. We can rewrite the design/feature matrix in terms of its column vectors as

$$\mathbf{X} = [\mathbf{x}_0 \quad \mathbf{x}_1 \quad \mathbf{x}_2 \quad \dots \quad \dots \quad \mathbf{x}_{p-1}],$$

with a given vector

$$\mathbf{x}_i^T = [x_{0,i} \quad x_{1,i} \quad x_{2,i} \quad \dots \quad \dots x_{n-1,i}].$$

## Simple Example

With these definitions, we can now rewrite our  $2 \times 2$  correlation/covariance matrix in terms of a more general design/feature matrix  $\mathbf{X} \in \mathbb{R}^{n \times p}$ . This leads to a  $p \times p$  covariance matrix for the vectors  $\mathbf{x}_i$  with  $i = 0, 1, \dots, p-1$

$$\mathbf{C}[\mathbf{x}] = \begin{bmatrix} \text{var}[\mathbf{x}_0] & \text{cov}[\mathbf{x}_0, \mathbf{x}_1] & \text{cov}[\mathbf{x}_0, \mathbf{x}_2] & \dots & \dots & \text{cov}[\mathbf{x}_0, \mathbf{x}_{p-1}] \\ \text{cov}[\mathbf{x}_1, \mathbf{x}_0] & \text{var}[\mathbf{x}_1] & \text{cov}[\mathbf{x}_1, \mathbf{x}_2] & \dots & \dots & \text{cov}[\mathbf{x}_1, \mathbf{x}_{p-1}] \\ \text{cov}[\mathbf{x}_2, \mathbf{x}_0] & \text{cov}[\mathbf{x}_2, \mathbf{x}_1] & \text{var}[\mathbf{x}_2] & \dots & \dots & \text{cov}[\mathbf{x}_2, \mathbf{x}_{p-1}] \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \text{cov}[\mathbf{x}_{p-1}, \mathbf{x}_0] & \text{cov}[\mathbf{x}_{p-1}, \mathbf{x}_1] & \text{cov}[\mathbf{x}_{p-1}, \mathbf{x}_2] & \dots & \dots & \text{var}[\mathbf{x}_{p-1}] \end{bmatrix},$$

## The Correlation Matrix

and the correlation matrix

$$\mathbf{K}[\mathbf{x}] = \begin{bmatrix} 1 & \text{corr}[\mathbf{x}_0, \mathbf{x}_1] & \text{corr}[\mathbf{x}_0, \mathbf{x}_2] & \dots & \dots & \text{corr}[\mathbf{x}_0, \mathbf{x}_{p-1}] \\ \text{corr}[\mathbf{x}_1, \mathbf{x}_0] & 1 & \text{corr}[\mathbf{x}_1, \mathbf{x}_2] & \dots & \dots & \text{corr}[\mathbf{x}_1, \mathbf{x}_{p-1}] \\ \text{corr}[\mathbf{x}_2, \mathbf{x}_0] & \text{corr}[\mathbf{x}_2, \mathbf{x}_1] & 1 & \dots & \dots & \text{corr}[\mathbf{x}_2, \mathbf{x}_{p-1}] \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \text{corr}[\mathbf{x}_{p-1}, \mathbf{x}_0] & \text{corr}[\mathbf{x}_{p-1}, \mathbf{x}_1] & \text{corr}[\mathbf{x}_{p-1}, \mathbf{x}_2] & \dots & \dots & 1 \end{bmatrix},$$

## Numpy Functionality

The Numpy function **np.cov** calculates the covariance elements using the factor  $1/(n-1)$  instead of  $1/n$  since it assumes we do not have the exact mean values. The following simple function uses the **np.vstack** function which takes each vector of dimension  $1 \times n$  and produces a  $2 \times n$  matrix **W**

$$\mathbf{W}^T = \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \dots & \dots \\ x_{n-2} & y_{n-2} \\ x_{n-1} & y_{n-1} \end{bmatrix},$$

which in turn is converted into the  $2 \times 2$  covariance matrix **C** via the Numpy function **np.cov()**. We note that we can also calculate the mean value of each set of samples **x** etc using the Numpy function **np.mean(x)**. We can also extract the eigenvalues of the covariance matrix through the **np.linalg.eig()** function.

```
# Importing various packages
import numpy as np
n = 100
x = np.random.normal(size=n)
print(np.mean(x))
y = 4+3*x+np.random.normal(size=n)
print(np.mean(y))
W = np.vstack((x, y))
C = np.cov(W)
print(C)
```

## Correlation Matrix again

The previous example can be converted into the correlation matrix by simply scaling the matrix elements with the variances. We should also subtract the mean values for each column. This leads to the following code which sets up the correlations matrix for the previous example in a more brute force way. Here we scale the mean values for each column of the design matrix, calculate the relevant mean values and variances and then finally set up the  $2 \times 2$  correlation matrix (since we have only two vectors).

```
import numpy as np
n = 100
# define two vectors
x = np.random.random(size=n)
y = 4+3*x+np.random.normal(size=n)
#scaling the x and y vectors
x = x - np.mean(x)
y = y - np.mean(y)
variance_x = np.sum(x*x)/n
variance_y = np.sum(y*y)/n
print(variance_x)
```



```

print(variance_y)
cov_xy = np.sum(x@y)/n
cov_xx = np.sum(x@x)/n
cov_yy = np.sum(y@y)/n
C = np.zeros((2,2))
C[0,0] = cov_xx/variance_x
C[1,1] = cov_yy/variance_y
C[0,1] = cov_xy/np.sqrt(variance_y*variance_x)
C[1,0] = C[0,1]
print(C)

```

We see that the matrix elements along the diagonal are one as they should be and that the matrix is symmetric. Furthermore, diagonalizing this matrix we easily see that it is a positive definite matrix.

The above procedure with **numpy** can be made more compact if we use **pandas**.

## Using Pandas

We show here how we can set up the correlation matrix using **pandas**, as done in this simple code

```

import numpy as np
import pandas as pd
n = 10
x = np.random.normal(size=n)
x = x - np.mean(x)
y = 4+3*x+np.random.normal(size=n)
y = y - np.mean(y)
X = (np.vstack((x, y))).T
print(X)
Xpd = pd.DataFrame(X)
print(Xpd)
correlation_matrix = Xpd.corr()
print(correlation_matrix)

```

## And then the Franke Function

We expand this model to the Franke function discussed above.

```

# Common imports
import numpy as np
import pandas as pd

def FrankeFunction(x,y):
    term1 = 0.75*np.exp(-(0.25*(9*x-2)**2) - 0.25*((9*y-2)**2))
    term2 = 0.75*np.exp(-((9*x+1)**2)/49.0 - 0.1*(9*y+1))
    term3 = 0.5*np.exp(-(9*x-7)**2/4.0 - 0.25*((9*y-3)**2))
    term4 = -0.2*np.exp(-(9*x-4)**2 - (9*y-7)**2)
    return term1 + term2 + term3 + term4

def create_X(x, y, n):

```

```

if len(x.shape) > 1:
    x = np.ravel(x)
    y = np.ravel(y)

N = len(x)
l = int((n+1)*(n+2)/2)          # Number of elements in beta
X = np.ones((N,l))

for i in range(1,n+1):
    q = int((i)*(i+1)/2)
    for k in range(i+1):
        X[:,q+k] = (x**(i-k))*(y**k)

return X

# Making meshgrid of datapoints and compute Franke's function
n = 4
N = 100
x = np.sort(np.random.uniform(0, 1, N))
y = np.sort(np.random.uniform(0, 1, N))
z = FrankeFunction(x, y)
X = create_X(x, y, n=n)

Xpd = pd.DataFrame(X)
# subtract the mean values and set up the covariance matrix
Xpd = Xpd - Xpd.mean()
covariance_matrix = Xpd.cov()
print(covariance_matrix)

```

We note here that the covariance is zero for the first rows and columns since all matrix elements in the design matrix were set to one (we are fitting the function in terms of a polynomial of degree  $n$ ). We would however not include the intercept and we can simply drop these elements and construct a correlation matrix without them by centering our matrix elements by subtracting the mean of each column.

## Lnks with the Design Matrix

We can rewrite the covariance matrix in a more compact form in terms of the design/feature matrix  $\mathbf{X}$  as

$$\mathbf{C}[\mathbf{x}] = \frac{1}{n} \mathbf{X}^T \mathbf{X} = \mathbb{E}[\mathbf{X}^T \mathbf{X}].$$

To see this let us simply look at a design matrix  $\mathbf{X} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{bmatrix} = [\mathbf{x}_0 \quad \mathbf{x}_1].$$

## Computing the Expectation Values

If we then compute the expectation value

$$\mathbb{E}[\mathbf{X}^T \mathbf{X}] = \frac{1}{n} \mathbf{X}^T \mathbf{X} = \begin{bmatrix} x_{00}^2 + x_{01}^2 & x_{00}x_{10} + x_{01}x_{11} \\ x_{10}x_{00} + x_{11}x_{01} & x_{10}^2 + x_{11}^2 \end{bmatrix},$$

which is just

$$\mathbf{C}[\mathbf{x}_0, \mathbf{x}_1] = \mathbf{C}[\mathbf{x}] = \begin{bmatrix} \text{var}[\mathbf{x}_0] & \text{cov}[\mathbf{x}_0, \mathbf{x}_1] \\ \text{cov}[\mathbf{x}_1, \mathbf{x}_0] & \text{var}[\mathbf{x}_1] \end{bmatrix},$$

where we wrote

$$\mathbf{C}[\mathbf{x}_0, \mathbf{x}_1] = \mathbf{C}[\mathbf{x}]$$

to indicate that this is the covariance of the vectors  $\mathbf{x}$  of the design/feature matrix  $\mathbf{X}$ .

It is easy to generalize this to a matrix  $\mathbf{X} \in \mathbb{R}^{n \times p}$ .

## Towards the PCA theorem

We have that the covariance matrix (the correlation matrix involves a simple rescaling) is given as

$$\mathbf{C}[\mathbf{x}] = \frac{1}{n} \mathbf{X}^T \mathbf{X} = \mathbb{E}[\mathbf{X}^T \mathbf{X}].$$

Let us now assume that we can perform a series of orthogonal transformations where we employ some orthogonal matrices  $\mathbf{S}$ . These matrices are defined as  $\mathbf{S} \in \mathbb{R}^{p \times p}$  and obey the orthogonality requirements  $\mathbf{S}\mathbf{S}^T = \mathbf{S}^T\mathbf{S} = \mathbf{I}$ . The matrix can be written out in terms of the column vectors  $\mathbf{s}_i$  as  $\mathbf{S} = [\mathbf{s}_0, \mathbf{s}_1, \dots, \mathbf{s}_{p-1}]$  and  $\mathbf{s}_i \in \mathbb{R}^p$ .

Assume also that there is a transformation  $\mathbf{S}^T \mathbf{C}[\mathbf{x}] \mathbf{S} = \mathbf{C}[\mathbf{y}]$  such that the new matrix  $\mathbf{C}[\mathbf{y}]$  is diagonal with elements  $[\lambda_0, \lambda_1, \lambda_2, \dots, \lambda_{p-1}]$ .

That is we have

$$\mathbf{C}[\mathbf{y}] = \mathbb{E}[\mathbf{S}^T \mathbf{X}^T \mathbf{X} \mathbf{S}] = \mathbf{S}^T \mathbf{C}[\mathbf{x}] \mathbf{S},$$

since the matrix  $\mathbf{S}$  is not a data dependent matrix. Multiplying with  $\mathbf{S}$  from the left we have

$$\mathbf{S} \mathbf{C}[\mathbf{y}] = \mathbf{C}[\mathbf{x}] \mathbf{S},$$

and since  $\mathbf{C}[\mathbf{y}]$  is diagonal we have for a given eigenvalue  $i$  of the covariance matrix that

$$\mathbf{S}_i \lambda_i = \mathbf{C}[\mathbf{x}] \mathbf{S}_i.$$

## More on the PCA Theorem

In the derivation of the PCA theorem we will assume that the eigenvalues are ordered in descending order, that is  $\lambda_0 > \lambda_1 > \dots > \lambda_{p-1}$ .

The eigenvalues tell us then how much we need to stretch the corresponding eigenvectors. Dimensions with large eigenvalues have thus large variations (large variance) and define therefore useful dimensions. The data points are more spread out in the direction of these eigenvectors. Smaller eigenvalues mean on the other hand that the corresponding eigenvectors are shrunk accordingly and

the data points are tightly bunched together and there is not much variation in these specific directions. Hopefully then we could leave out dimensions where the eigenvalues are very small. If  $p$  is very large, we could then aim at reducing  $p$  to  $l \ll p$  and handle only  $l$  features/predictors.

## The Algorithm before the Theorem

Here's how we would proceed in setting up the algorithm for the PCA, see also discussion below here.

- Set up the datapoints for the design/feature matrix  $\mathbf{X}$  with  $\mathbf{X} \in \mathbb{R}^{n \times p}$ , with the predictors/features  $p$  referring to the column numbers and the entries  $n$  being the row elements.

$$\mathbf{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & \dots & \dots x_{0,p-1} \\ x_{1,0} & x_{1,1} & x_{1,2} & \dots & \dots x_{1,p-1} \\ x_{2,0} & x_{2,1} & x_{2,2} & \dots & \dots x_{2,p-1} \\ \dots & \dots & \dots & \dots & \dots \\ x_{n-2,0} & x_{n-2,1} & x_{n-2,2} & \dots & \dots x_{n-2,p-1} \\ x_{n-1,0} & x_{n-1,1} & x_{n-1,2} & \dots & \dots x_{n-1,p-1} \end{bmatrix},$$

- Center the data by subtracting the mean value for each column. This leads to a new matrix  $\mathbf{X} \rightarrow \bar{\mathbf{X}}$ .
- Compute then the covariance/correlation matrix  $\mathbb{E}[\bar{\mathbf{X}}^T \bar{\mathbf{X}}]$ .
- Find the eigenpairs of  $\mathbf{C}$  with eigenvalues  $[\lambda_0, \lambda_1, \dots, \lambda_{p-1}]$  and eigenvectors  $[\mathbf{s}_0, \mathbf{s}_1, \dots, \mathbf{s}_{p-1}]$ .
- Order the eigenvalue (and the eigenvectors accordingly) in order of decreasing eigenvalues.
- Keep only those  $l$  eigenvalues larger than a selected threshold value, discarding thus  $p - l$  features since we expect small variations in the data here.

## Writing our own PCA code

We will use a simple example first with two-dimensional data drawn from a multivariate normal distribution with the following mean and covariance matrix (we have fixed these quantities but will play around with them below):

$$\mu = (-1, 2) \quad \Sigma = \begin{bmatrix} 4 & 2 \\ 2 & 2 \end{bmatrix}$$

Note that the mean refers to each column of data. We will generate  $n = 10000$  points  $X = \{x_1, \dots, x_N\}$  from this distribution, and store them in the  $1000 \times 2$  matrix  $\mathbf{X}$ . This is our design matrix where we have forced the covariance and mean values to take specific values.

## Implementing it

The following Python code aids in setting up the data and writing out the design matrix. Note that the function **multivariate** returns also the covariance discussed above and that it is defined by dividing by  $n - 1$  instead of  $n$ .

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import display
n = 10000
mean = (-1, 2)
cov = [[4, 2], [2, 2]]
X = np.random.multivariate_normal(mean, cov, n)
```

Now we are going to implement the PCA algorithm. We will break it down into various substeps.

### First Step

The first step of PCA is to compute the sample mean of the data and use it to center the data. Recall that the sample mean is

$$\mu_n = \frac{1}{n} \sum_{i=1}^n x_i$$

and the mean-centered data  $\bar{X} = \{\bar{x}_1, \dots, \bar{x}_n\}$  takes the form

$$\bar{x}_i = x_i - \mu_n.$$

When you are done with these steps, print out  $\mu_n$  to verify it is close to  $\mu$  and plot your mean centered data to verify it is centered at the origin! The following code elements perform these operations using **pandas** or using our own functionality for doing so. The latter, using **numpy** is rather simple through the **mean()** function.

```
df = pd.DataFrame(X)
# Pandas does the centering for us
df = df - df.mean()
# we center it ourselves
X_centered = X - X.mean(axis=0)
```

### Scaling

Alternatively, we could use the functions we discussed earlier for scaling the data set. That is, we could have used the **StandardScaler** function in **Scikit-Learn**, a function which ensures that for each feature/predictor we study the mean value is zero and the variance is one (every column in the design/feature matrix). You would then not get the same results, since we divide by the variance. The diagonal covariance matrix elements will then be one, while the non-diagonal ones need to be divided by  $2\sqrt{2}$  for our specific case.

## Centered Data

Now we are going to use the mean centered data to compute the sample covariance of the data by using the following equation

$$\Sigma_n = \frac{1}{n-1} \sum_{i=1}^n \bar{x}_i^T \bar{x}_i = \frac{1}{n-1} \sum_{i=1}^n (x_i - \mu_n)^T (x_i - \mu_n)$$

where the data points  $x_i \in \mathbb{R}^p$  (here in this example  $p = 2$ ) are column vectors and  $x^T$  is the transpose of  $x$ . We can write our own code or simply use either the functionality of **numpy** or that of **pandas**, as follows

```
print(df.cov())
print(np.cov(X_centered.T))
```

Note that the way we define the covariance matrix here has a factor  $n - 1$  instead of  $n$ . This is included in the `cov()` function by **numpy** and **pandas**. Our own code here is not very elegant and asks for obvious improvements. It is tailored to this specific  $2 \times 2$  covariance matrix.

```
# extract the relevant columns from the centered design matrix of dim n x 2
x = X_centered[:,0]
y = X_centered[:,1]
Cov = np.zeros((2,2))
Cov[0,1] = np.sum(x.T@y)/(n-1.0)
Cov[0,0] = np.sum(x.T@x)/(n-1.0)
Cov[1,1] = np.sum(y.T@y)/(n-1.0)
Cov[1,0] = Cov[0,1]
print("Centered covariance using own code")
print(Cov)
plt.plot(x, y, 'x')
plt.axis('equal')
plt.show()
```

## Exploring

Depending on the number of points  $n$ , we will get results that are close to the covariance values defined above. The plot shows how the data are clustered around a line with slope close to one. Is this expected? Try to change the covariance and the mean values. For example, try to make the variance of the first element much larger than that of the second diagonal element. Try also to shrink the covariance (the non-diagonal elements) and see how the data points are distributed.

## Diagonalize the sample covariance matrix to obtain the principal components

Now we are ready to solve for the principal components! To do so we diagonalize the sample covariance matrix  $\Sigma$ . We can use the function `np.linalg.eig` to do so. It will return the eigenvalues and eigenvectors of  $\Sigma$ . Once we have these we can perform the following tasks:

- We compute the percentage of the total variance captured by the first principal component
- We plot the mean centered data and lines along the first and second principal components
- Then we project the mean centered data onto the first and second principal components, and plot the projected data.
- Finally, we approximate the data as

$$x_i \approx \tilde{x}_i = \mu_n + \langle x_i, v_0 \rangle v_0$$

where  $v_0$  is the first principal component.

## Collecting all Steps

Collecting all these steps we can write our own PCA function and compare this with the functionality included in **Scikit-Learn**.

The code here outlines some of the elements we could include in the analysis. Feel free to extend upon this in order to address the above questions.

```
# diagonalize and obtain eigenvalues, not necessarily sorted
EigValues, EigVectors = np.linalg.eig(Cov)
# sort eigenvectors and eigenvalues
#permute = EigValues.argsort()
#EigValues = EigValues[permute]
#EigVectors = EigVectors[:,permute]
print("Eigenvalues of Covariance matrix")
for i in range(2):
    print(EigValues[i])
FirstEigvector = EigVectors[:,0]
SecondEigvector = EigVectors[:,1]
print("First eigenvector")
print(FirstEigvector)
print("Second eigenvector")
print(SecondEigvector)
#thereafter we do a PCA with Scikit-learn
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2Dsl = pca.fit_transform(X)
print("Eigenvector of largest eigenvalue")
print(pca.components_.T[:, 0])
```

This code does not contain all the above elements, but it shows how we can use **Scikit-Learn** to extract the eigenvector which corresponds to the largest eigenvalue. Try to address the questions we pose before the above code. Try also to change the values of the covariance matrix by making one of the diagonal elements much larger than the other. What do you observe then?

## Classical PCA Theorem

We assume now that we have a design matrix  $\mathbf{X}$  which has been centered as discussed above. For the sake of simplicity we skip the overline symbol. The matrix is defined in terms of the various column vectors  $[\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{p-1}]$  each with dimension  $\mathbf{x} \in \mathbb{R}^n$ .

The PCA theorem states that minimizing the above reconstruction error corresponds to setting  $\mathbf{W} = \mathbf{S}$ , the orthogonal matrix which diagonalizes the empirical covariance(correlation) matrix. The optimal low-dimensional encoding of the data is then given by a set of vectors  $\mathbf{z}_i$  with at most  $l$  vectors, with  $l \ll p$ , defined by the orthogonal projection of the data onto the columns spanned by the eigenvectors of the covariance(correlations matrix).

## The PCA Theorem

To show the PCA theorem let us start with the assumption that there is one vector  $\mathbf{s}_0$  which corresponds to a solution which minimized the reconstruction error  $J$ . This is an orthogonal vector. It means that we now approximate the reconstruction error in terms of  $\mathbf{w}_0$  and  $\mathbf{z}_0$  as

We are almost there, we have obtained a relation between minimizing the reconstruction error and the variance and the covariance matrix. Minimizing the error is equivalent to maximizing the variance of the projected data.

We could trivially maximize the variance of the projection (and thereby minimize the error in the reconstruction function) by letting the norm-2 of  $\mathbf{w}_0$  go to infinity. However, this norm since we want the matrix  $\mathbf{W}$  to be an orthogonal matrix, is constrained by  $\|\mathbf{w}_0\|_2^2 = 1$ . Imposing this condition via a Lagrange multiplier we can then in turn maximize

$$J(\mathbf{w}_0) = \mathbf{w}_0^T \mathbf{C}[\mathbf{x}] \mathbf{w}_0 + \lambda_0(1 - \mathbf{w}_0^T \mathbf{w}_0).$$

Taking the derivative with respect to  $\mathbf{w}_0$  we obtain

$$\frac{\partial J(\mathbf{w}_0)}{\partial \mathbf{w}_0} = 2\mathbf{C}[\mathbf{x}] \mathbf{w}_0 - 2\lambda_0 \mathbf{w}_0 = 0,$$

meaning that

$$\mathbf{C}[\mathbf{x}] \mathbf{w}_0 = \lambda_0 \mathbf{w}_0.$$

**The direction that maximizes the variance (or minimizes the construction error) is an eigenvector of the covariance matrix!** If we left multiply with  $\mathbf{w}_0^T$  we have the variance of the projected data is

$$\mathbf{w}_0^T \mathbf{C}[\mathbf{x}] \mathbf{w}_0 = \lambda_0.$$

If we want to maximize the variance (minimize the construction error) we simply pick the eigenvector of the covariance matrix with the largest eigenvalue. This establishes the link between the minimization of the reconstruction function  $J$  in terms of an orthogonal matrix and the maximization of the variance and



thereby the covariance of our observations encoded in the design/feature matrix  $\mathbf{X}$ .

The proof for the other eigenvectors  $\mathbf{w}_1, \mathbf{w}_2, \dots$  can be established by applying the above arguments and using the fact that our basis of eigenvectors is orthogonal, see [Murphy chapter 12.2](#). The discussion in chapter 12.2 of Murphy's text has also a nice link with the Singular Value Decomposition theorem. For categorical data, see chapter 12.4 and discussion therein.

For more details, see for example [Vidal, Ma and Sastry, chapter 2](#).

## Geometric Interpretation and link with Singular Value Decomposition

For a detailed demonstration of the geometric interpretation, see [Vidal, Ma and Sastry, section 2.1.2](#).

Principal Component Analysis (PCA) is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it.

The following Python code uses NumPy's `svd()` function to obtain all the principal components of the training set, then extracts the first two principal components. First we center the data using either **pandas** or our own code

```
import numpy as np
import pandas as pd
from IPython.display import display
np.random.seed(100)
# setting up a 10 x 5 vanilla matrix
rows = 10
cols = 5
X = np.random.randn(rows, cols)
df = pd.DataFrame(X)
# Pandas does the centering for us
df = df - df.mean()
display(df)

# we center it ourselves
X_centered = X - X.mean(axis=0)
# Then check the difference between pandas and our own set up
print(X_centered - df)
# Now we do an SVD
U, s, V = np.linalg.svd(X_centered)
c1 = V.T[:, 0]
c2 = V.T[:, 1]
W2 = V.T[:, :2]
X2D = X_centered.dot(W2)
print(X2D)
```

PCA assumes that the dataset is centered around the origin. Scikit-Learn's PCA classes take care of centering the data for you. However, if you implement PCA yourself (as in the preceding example), or if you use other libraries, don't forget to center the data first.

Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to  $d$  dimensions by projecting it onto the

hyperplane defined by the first  $d$  principal components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible.

```
W2 = V.T[:, :2]
X2D = X_centered.dot(W2)
```

## PCA and scikit-learn

Scikit-Learn's PCA class implements PCA using SVD decomposition just like we did before. The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions (note that it automatically takes care of centering the data):

```
#thereafter we do a PCA with Scikit-learn
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
print(X2D)
```

After fitting the PCA transformer to the dataset, you can access the principal components using the `components` variable (note that it contains the PCs as horizontal vectors, so, for example, the first principal component is equal to

```
pca.components_.T[:, 0]
```

Another very useful piece of information is the explained variance ratio of each principal component, available via the `explained_variance_ratio` variable. It indicates the proportion of the dataset's variance that lies along the axis of each principal component.

## Back to the Cancer Data

We can now repeat the above but applied to real data, in this case our breast cancer data. Here we compute performance scores on the training data using logistic regression.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)

logreg = LogisticRegression()
logreg.fit(X_train, y_train)
print("Train set accuracy from Logistic Regression: {:.2f}".format(logreg.score(X_train, y_train)))
# We scale the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

```

scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Then perform again a log reg fit
logreg.fit(X_train_scaled, y_train)
print("Train set accuracy scaled data: {:.2f}".format(logreg.score(X_train_scaled,y_train)))
#thereafter we do a PCA with Scikit-learn
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2D_train = pca.fit_transform(X_train_scaled)
# and finally compute the log reg fit and the score on the training data
logreg.fit(X2D_train,y_train)
print("Train set accuracy scaled and PCA data: {:.2f}".format(logreg.score(X2D_train,y_train)))

```

We see that our training data after the PCA decomposition has a performance similar to the non-scaled data.

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is generally preferable to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%). Unless, of course, you are reducing dimensionality for data visualization — in that case you will generally want to reduce the dimensionality down to 2 or 3. The following code computes PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of the training set's variance:

```

pca = PCA()
pca.fit(X)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1

```

You could then set  $n\_components = d$  and run PCA again. However, there is a much better option: instead of specifying the number of principal components you want to preserve, you can set  $n\_components$  to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

```

pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X)

```

## Incremental PCA

One problem with the preceding implementation of PCA is that it requires the whole training set to fit in memory in order for the SVD algorithm to run. Fortunately, Incremental PCA (IPCA) algorithms have been developed: you can split the training set into mini-batches and feed an IPCA algorithm one minibatch at a time. This is useful for large training sets, and also to apply PCA online (i.e., on the fly, as new instances arrive).

**Randomized PCA.** Scikit-Learn offers yet another option to perform PCA, called Randomized PCA. This is a stochastic algorithm that quickly finds an

approximation of the first  $d$  principal components. Its computational complexity is  $O(m \times d^2) + O(d^3)$ , instead of  $O(m \times n^2) + O(n^3)$ , so it is dramatically faster than the previous algorithms when  $d$  is much smaller than  $n$ .

**Kernel PCA.** The kernel trick is a mathematical technique that implicitly maps instances into a very high-dimensional space (called the feature space), enabling nonlinear classification and regression with Support Vector Machines. Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the original space. It turns out that the same trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called Kernel PCA (kPCA). It is often good at preserving clusters of instances after projection, or sometimes even unrolling datasets that lie close to a twisted manifold. For example, the following code uses Scikit-Learn's KernelPCA class to perform kPCA with an

```
from sklearn.decomposition import KernelPCA
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

## Other techniques

There are many other dimensionality reduction techniques, several of which are available in Scikit-Learn.

Here are some of the most popular:

- **Multidimensional Scaling (MDS)** reduces dimensionality while trying to preserve the distances between the instances.
- **Isomap** creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the geodesic distances between the instances.
- **t-Distributed Stochastic Neighbor Embedding (t-SNE)** reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space (e.g., to visualize the MNIST images in 2D).
- **Linear Discriminant Analysis (LDA)** is actually a classification algorithm, but during training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data. The benefit is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm such as a Support Vector Machine (SVM) classifier discussed in the SVM lectures.