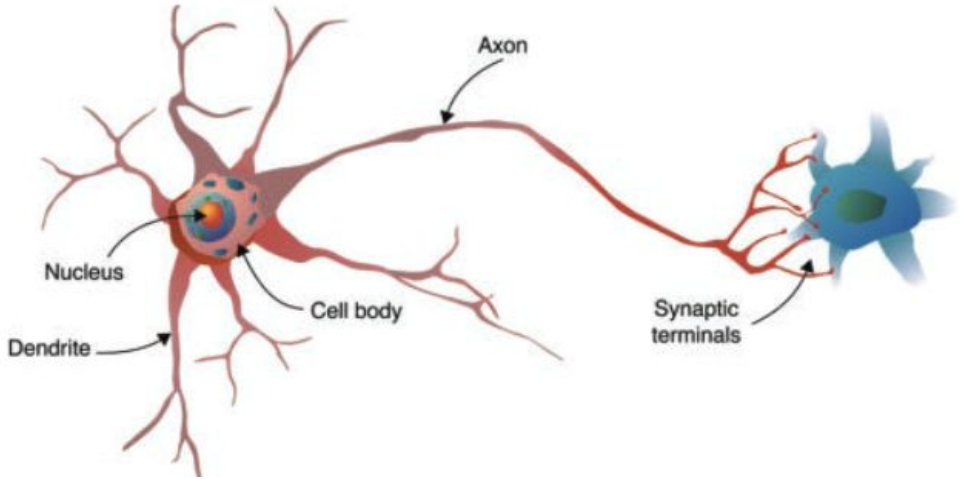
Machine learning 1a

Ekman Chapter 1

Perceptron vs Neuron



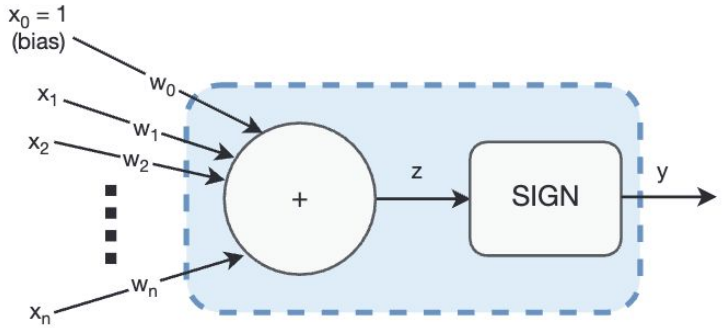
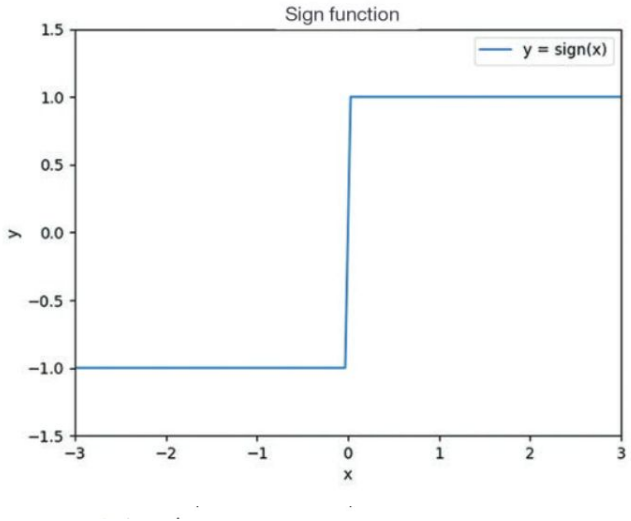


Figure 1-2 The perceptron

Funxctions



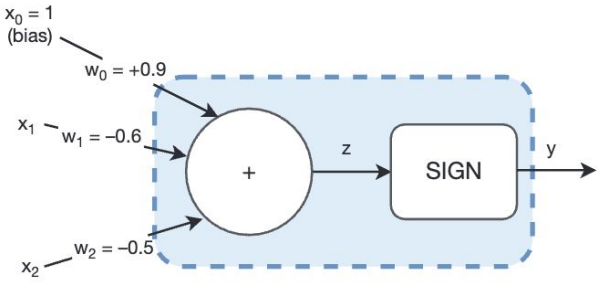
$y = f(z)$, where

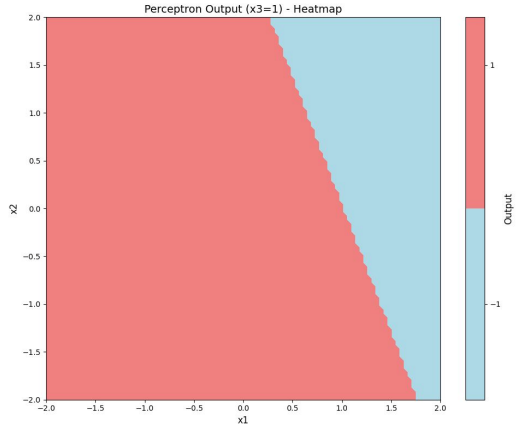
$$z = \sum_{i=0}^{n} w_i x_i$$

$$f(z) = \begin{cases} -1, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

$x_0 = 1$ (bias term)

2-state perceptron





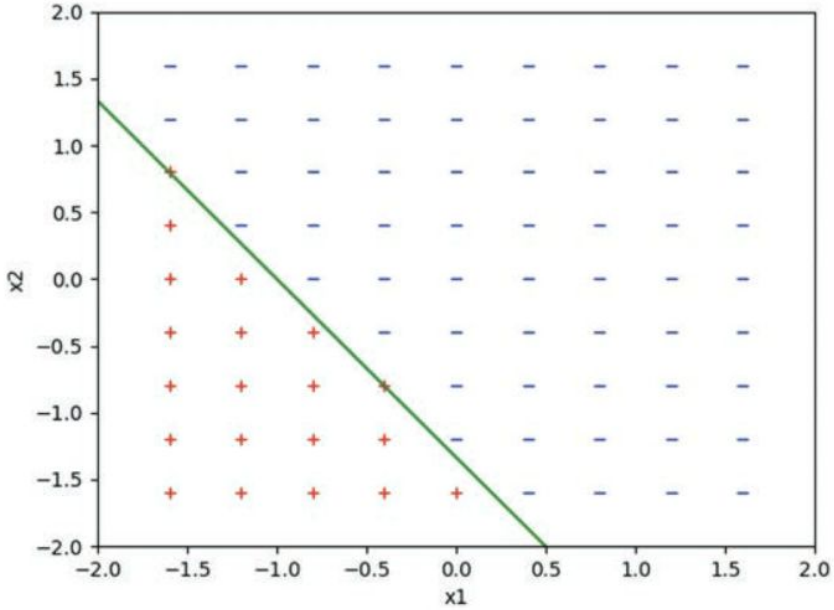


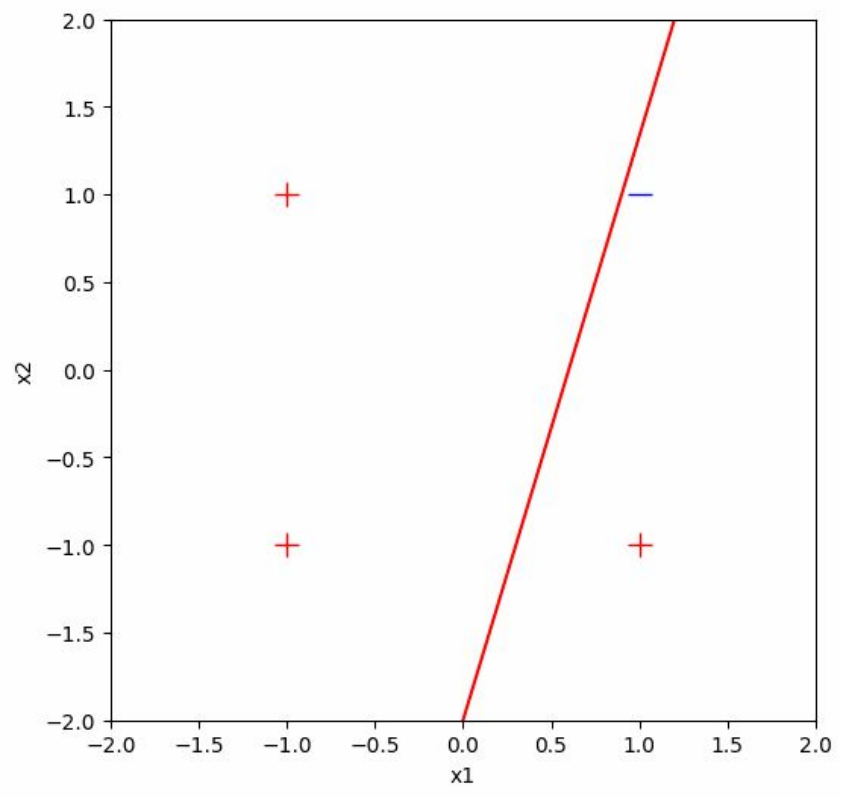
Figure 1-5 Output of a perceptron as a function of two inputs x_1 and x_2

Learning

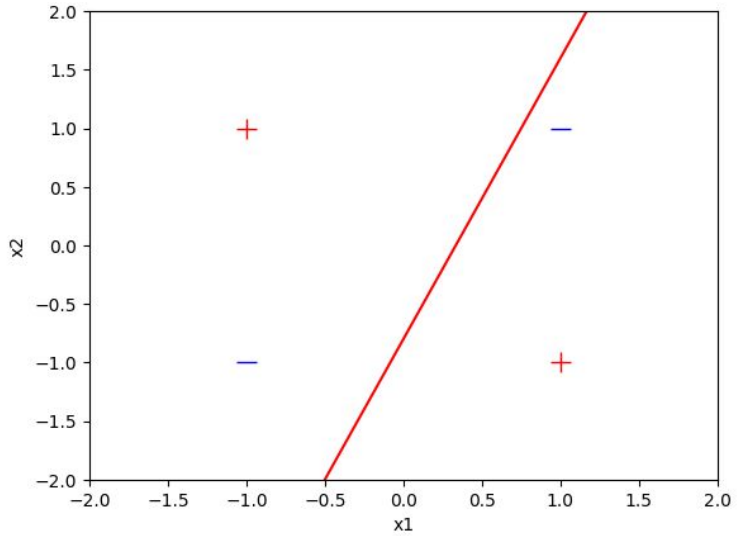
Code Snippet 1-4 Perceptron Training Loop

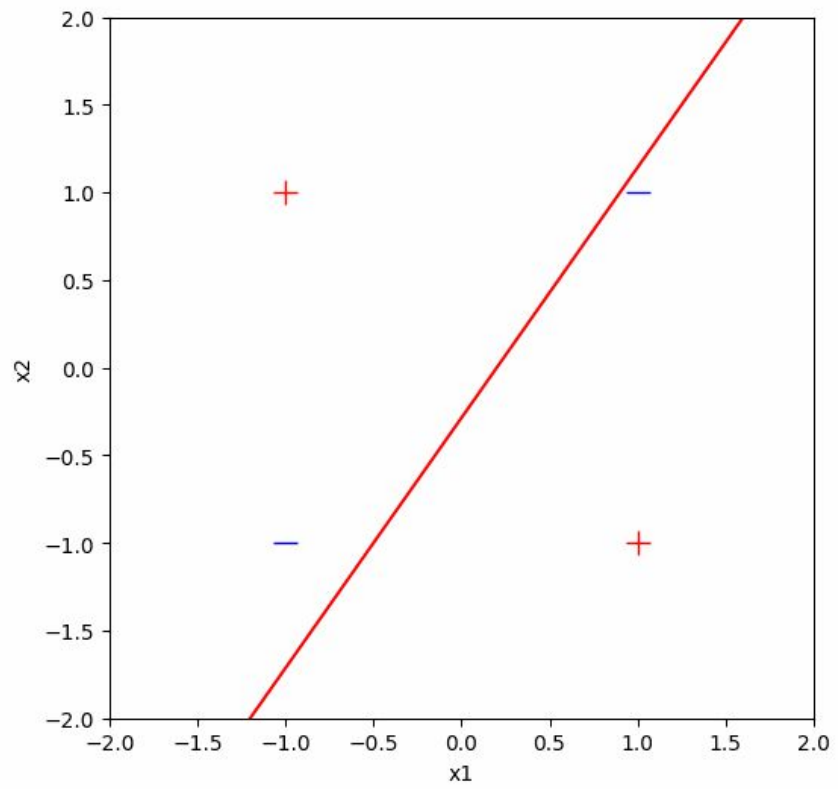
```
# Perceptron training loop.
all_correct = False
while not all_correct:
    all_correct = True
    random.shuffle(index_list) # Randomize order
    for i in index_list:
        x = x_train[i]
        y = y_train[i]
        p_out = compute_output(w, x) # Perceptron function

        if y != p_out: # Update weights when wrong
            for j in range(0, len(w)):
                w[j] += (y * LEARNING_RATE * x[j])
            all_correct = False
            show_learning(w) # Show updated weights
```

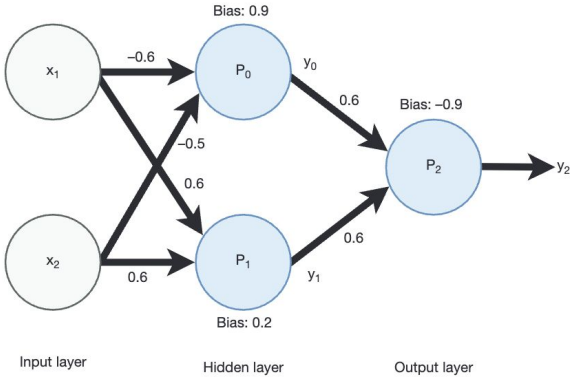


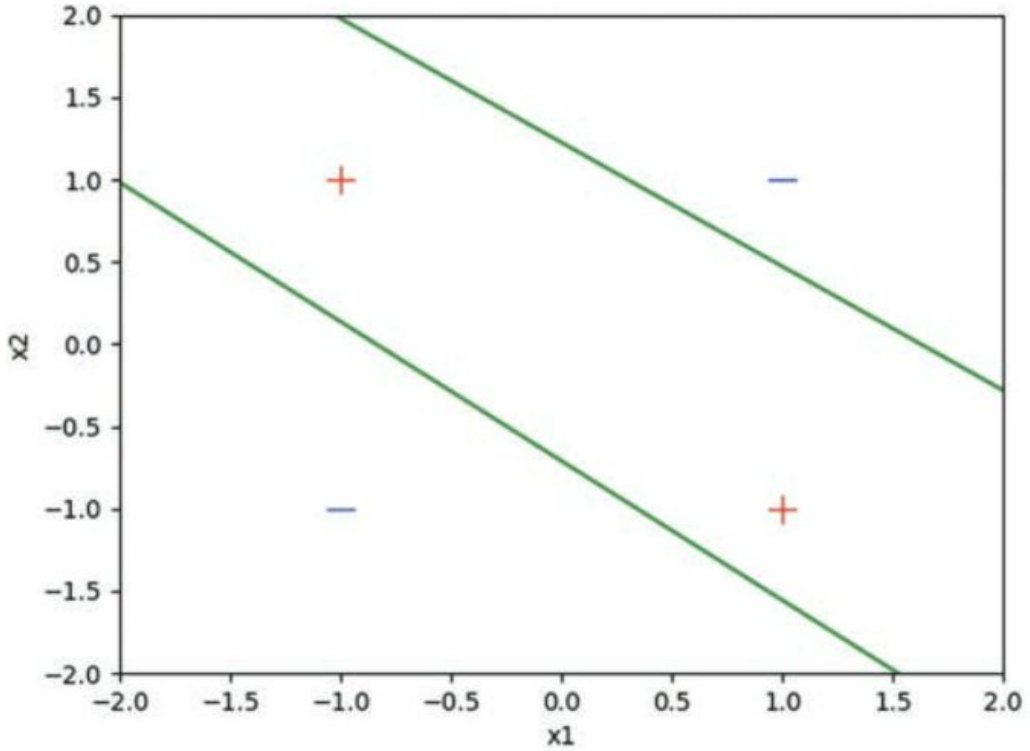
XOR problem


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
x2
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x1

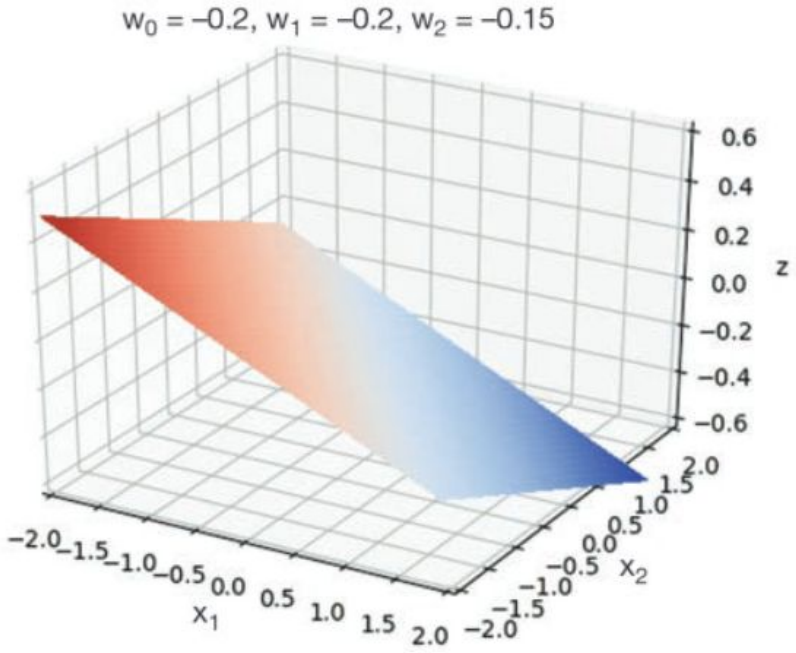

2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
x2
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x1

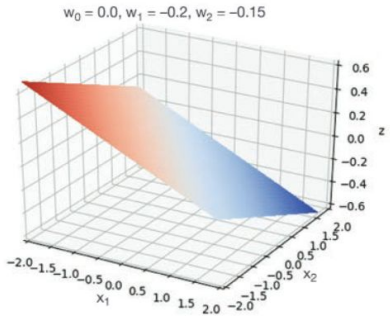
Combining multiple perceptrons

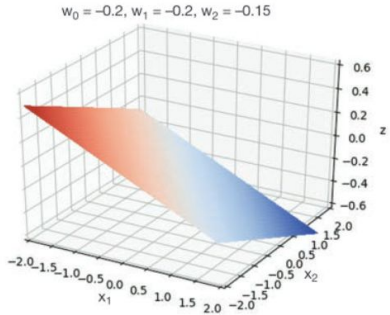


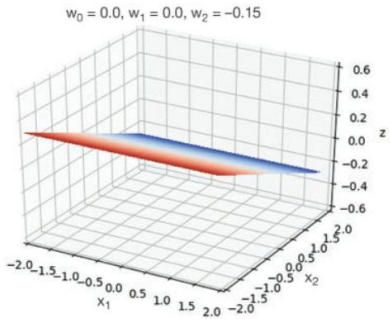


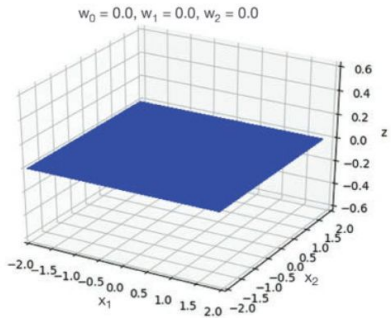
3D representations of double input perceptrons











Understanding the bias term

$$f(z)=\begin{cases} -1, & z<\theta \\ 1, & z\geq\theta \end{cases}$$

Looking more closely at the condition that needs to be fulfilled for the output to take on the value 1, we have

$$z \geq \theta$$

This can be rewritten as

$$z - \theta \geq 0$$

The bias term is simply the intercept term b in the equation for a straight line

$$y = mx + b$$

Perceptrons using linear algebra - basis

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} \qquad \boldsymbol{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix}$$

Transformation

$$\boldsymbol{x} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix}, \qquad \boldsymbol{x}^T = \begin{pmatrix} x_0 & x_1 & \dots & x_n \end{pmatrix}$$

Addition

$$\boldsymbol{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} \quad \boldsymbol{b} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{pmatrix} \quad \boldsymbol{a} + \boldsymbol{b} = \begin{pmatrix} a_0 + b_0 \\ a_1 + b_1 \\ \vdots \\ a_n + b_n \end{pmatrix}$$

Dot product

$$\boldsymbol{w} \cdot \boldsymbol{x} = w_0 x_0 + w_1 x_1 + \ldots + w_n x_n = \sum_{i=0}^{n} w_i x_i$$

$$y = sign(\boldsymbol{w} \cdot \boldsymbol{x})$$

```
import numpy as np
def compute_output_vector(w, x):
    z = np.dot(w, x)
    return np.sign(z)
```

2D-matrix

$$A = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0n} \\ a_{10} & a_{11} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \qquad A^T = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

Matrix multiplication

$$\boldsymbol{y} = A\boldsymbol{x} = \begin{pmatrix} a_{00} & a_{01} & \dots & a_{0n} \\ a_{10} & a_{11} & \dots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & \dots & a_{mn} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} a_{00}x_0 + a_{01}x_1 + \dots + a_{0n}x_n \\ a_{10}x_0 + a_{11}x_1 + \dots + a_{1n}x_n \\ \vdots \\ a_{m0}x_0 + a_{m1}x_1 + \dots + a_{mn}x_n \end{pmatrix}$$

$$y_i = \sum_{j=0}^{n} a_{ij}x_j$$

$$\boldsymbol{y} = A\boldsymbol{x} = \begin{pmatrix} \boldsymbol{a}_0^T \\ \boldsymbol{a}_1^T \\ \vdots \\ \boldsymbol{a}_m^T \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \boldsymbol{a}_0^T \cdot \boldsymbol{x} \\ \boldsymbol{a}_1^T \cdot \boldsymbol{x} \\ \vdots \\ \boldsymbol{a}_m^T \cdot \boldsymbol{x} \end{pmatrix}$$

Matrix multiplications in

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} \qquad \boldsymbol{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix}$$

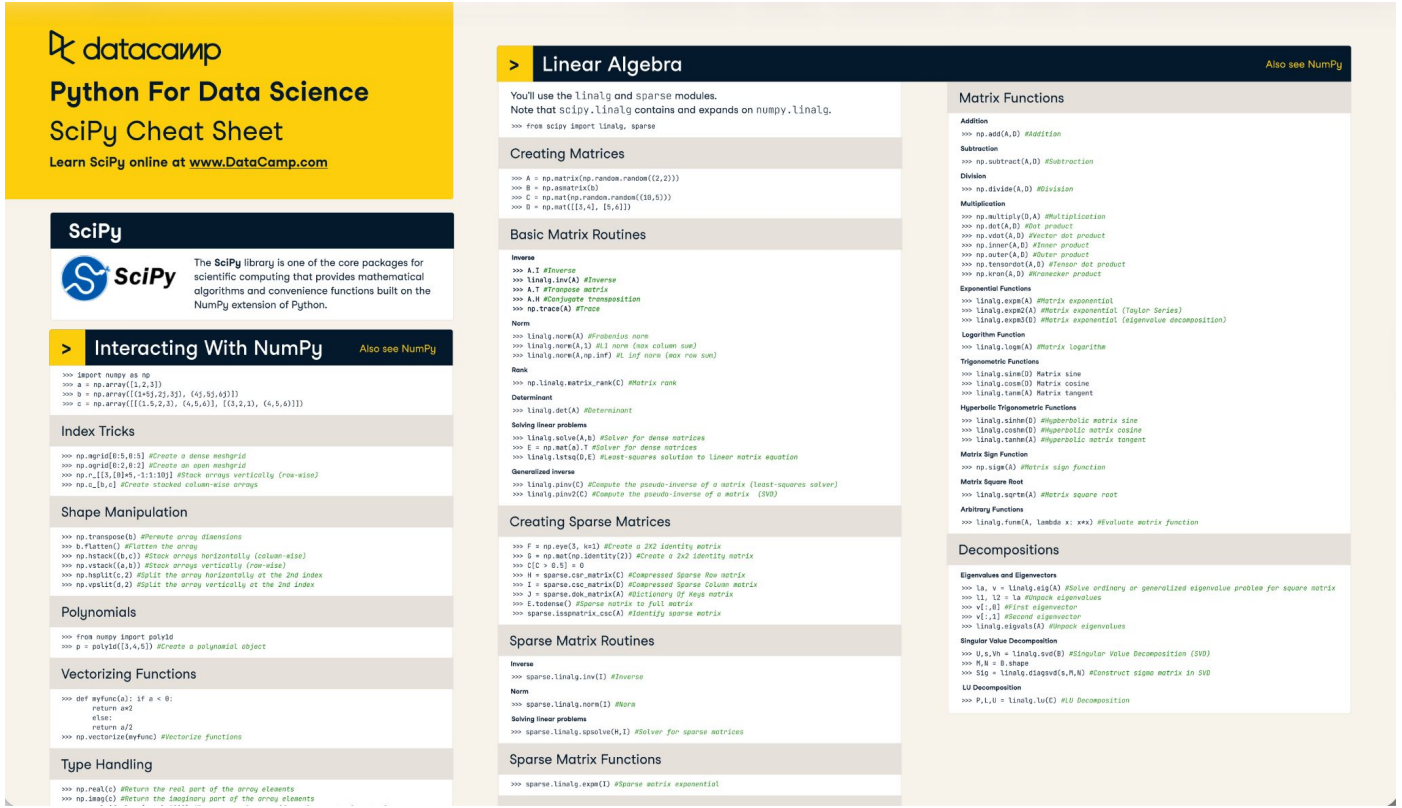
$$\boldsymbol{z} = W\boldsymbol{x} \qquad W = \begin{pmatrix} \boldsymbol{w}_0^T \\ \boldsymbol{w}_1^T \\ \vdots \\ \boldsymbol{w}_m^T \end{pmatrix}$$

Dot products as matrix multiplication

$$\boldsymbol{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix}, \qquad \boldsymbol{b} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{pmatrix}$$

$$\boldsymbol{a} \cdot \boldsymbol{b} = \sum_{i=0}^{n} a_i b_i = \begin{pmatrix} a_0 & a_1 & \dots & a_n \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{pmatrix} = \boldsymbol{a}^T \boldsymbol{b}$$

Numpy commands for linear algebra



datacamp

Python For Data Science

SciPy Cheat Sheet

Learn SciPy online at www.DataCamp.com

SciPy

SciPy

The **SciPy** library is one of the core packages for scientific computing that provides mathematical algorithms and convenience functions built on the NumPy extension of Python.

> Interacting With NumPy

Also see NumPy

```
>>> import numpy as np
>>> a = np.array([1,2,3])
>>> b = np.array([(1+5j,2j,3j), (4j,5j,6j)])
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]])
```

Index Tricks

```
>>> np.mgrid[0:5,0:5] #Create a dense meshgrid
>>> np.ogrid[0:2,0:2] #Create an open meshgrid
>>> np.r_[3,[0]*5,-1:1:10j] #Stack arrays vertically (row-wise)
>>> np.c_[b,c] #Create stacked column-wise arrays
```

Shape Manipulation

```
>>> np.transpose(b) #Permute array dimensions
>>> b.flatten() #Flatten the array
>>> np.hstack((b,c)) #Stack arrays horizontally (column-wise)
>>> np.vstack((a,b)) #Stack arrays vertically (row-wise)
>>> np.hsplit(c,2) #Split the array horizontally at the 2nd index
>>> np.vpsiit(d,2) #Split the array vertically at the 2nd index
```

Polynomials

```
>>> from numpy import poly1d
>>> p = poly1d([3,4,5]) #Create a polynomial object
```

Vectorizing Functions

```
>>> def myfunc(a): if a < 0:
        return a*2
    else:
        return a/2
>>> np.vectorize(myfunc) #Vectorize functions
```

Type Handling

```
>>> np.real(c) #Return the real part of the array elements
>>> np.imag(c) #Return the imaginary part of the array elements
```

> Linear Algebra

Also see NumPy

You'll use the `linalg` and `sparse` modules.
Note that `scipy.linalg` contains and expands on `numpy.linalg`.

```
>>> from scipy import linalg, sparse
```

Creating Matrices

```
>>> A = np.matrix(np.random.random((2,2)))
>>> B = np.asmatrix(b)
>>> C = np.mat(np.random.random((10,5)))
>>> D = np.mat([[3,4], [5,6]])
```

Basic Matrix Routines

Inverse

```
>>> A.I #Inverse
>>> linalg.inv(A) #Inverse
>>> A.T #Transpose matrix
>>> A.H #Conjugate transposition
>>> np.trace(A) #Trace
```

Norm

```
>>> linalg.norm(A) #Frobenius norm
>>> linalg.norm(A,1) #L1 norm (max column sum)
>>> linalg.norm(A,np.inf) #L inf norm (max row sum)
```

Rank

```
>>> np.linalg.matrix_rank(C) #Matrix rank
```

Determinant

```
>>> linalg.det(A) #Determinant
```

Solving linear problems

```
>>> linalg.solve(A,b) #Solver for dense matrices
>>> E = np.mat(a).T #Solver for dense matrices
>>> linalg.lstsq(D,E) #Least-squares solution to linear matrix equation
```

Generalized inverse

```
>>> linalg.pinv(C) #Compute the pseudo-inverse of a matrix (least-squares solver)
>>> linalg.pinv2(C) #Compute the pseudo-inverse of a matrix  (SVD)
```

Creating Sparse Matrices

```
>>> F = np.eye(3, k=1) #Create a 2X2 identity matrix
>>> G = np.mat(np.identity(2)) #Create a 2x2 identity matrix
>>> C[C > 0.5] = 0
>>> H = sparse.csr_matrix(C) #Compressed Sparse Row matrix
>>> I = sparse.csc_matrix(D) #Compressed Sparse Column matrix
>>> J = sparse.dok_matrix(A) #Dictionary Of Keys matrix
>>> E.todense() #Sparse matrix to full matrix
>>> sparse.isspmatrix_csc(A) #Identify sparse matrix
```

Sparse Matrix Routines

Inverse

```
>>> sparse.linalg.inv(I) #Inverse
```

Norm

```
>>> sparse.linalg.norm(I) #Norm
```

Solving linear problems

```
>>> sparse.linalg.spsolve(H,I) #Solver for sparse matrices
```

Sparse Matrix Functions

```
>>> sparse.linalg.expm(I) #Sparse matrix exponential
```

Matrix Functions

Addition

```
>>> np.add(A,D) #Addition
```

Subtraction

```
>>> np.subtract(A,D) #Subtraction
```

Division

```
>>> np.divide(A,D) #Division
```

Multiplication

```
>>> np.multiply(D,A) #Multiplication
>>> np.dot(A,D) #Dot product
>>> np.vdot(A,D) #Vector dot product
>>> np.inner(A,D) #Inner product
>>> np.outer(A,D) #Outer product
>>> np.tensordot(A,D) #Tensor dot product
>>> np.kron(A,D) #Kronecker product
```

Exponential Functions

```
>>> linalg.expm(A) #Matrix exponential
>>> linalg.expm2(A) #Matrix exponential (Taylor Series)
>>> linalg.expm3(D) #Matrix exponential (eigenvalue decomposition)
```

Logarithm Function

```
>>> linalg.logm(A) #Matrix logarithm
```

Trigonometric Functions

```
>>> linalg.sinm(D) Matrix sine
>>> linalg.cosm(D) Matrix cosine
>>> linalg.tanm(A) Matrix tangent
```

Hyperbolic Trigonometric Functions

```
>>> linalg.sinhm(D) #Hyperbolic matrix sine
>>> linalg.coshm(D) #Hyperbolic matrix cosine
>>> linalg.tanhm(A) #Hyperbolic matrix tangent
```

Matrix Sign Function

```
>>> np.sign(A) #Matrix sign function
```

Matrix Square Root

```
>>> linalg.sqrtm(A) #Matrix square root
```

Arbitrary Functions

```
>>> linalg.funm(A, lambda x: x*x) #Evaluate matrix function
```

Decompositions

Eigenvalues and Eigenvectors

```
>>> la, v = linalg.eig(A) #Solve ordinary or generalized eigenvalue problem for square matrix
>>> l1, l2 = la #Unpack eigenvalues
>>> v[:,0] #First eigenvector
>>> v[:,1] #Second eigenvector
>>> linalg.eigvals(A) #Unpack eigenvalues
```

Singular Value Decomposition

```
>>> U,s,Vh = linalg.svd(B) #Singular Value Decomposition (SVD)
>>> M,N = B.shape
>>> Sig = linalg.diagsvd(s,M,N) #Construct sigma matrix in SVD
```

LU Decomposition

```
>>> P,L,U = linalg.lu(C) #LU Decomposition
```

Chapter 1 summary

Perceptron can do logical operations (NAND, XOR)

Often Seen as binary classifications

Can be trained

Uses linear algebra to do themath

Mini-exercises (for students)

- Replace W and X with your own values and verify shapes.
- Make 3 neurons (W is 3×3) and 10 examples (X is 3×10). Compute Z = W@X.
- Implement matvec_manual(A, x) and matmul_manual(A, B) using nested loops, then compare
  NumPy with np.allclose.