EDAN96
Applied Machine Learning
Lecture 2: An Introduction to Linear Algebra with NumPy

Pierre Nugues

Pierre.Nugues@cs.lth.se
http://cs.lth.se/pierre_nugues/

November 2nd, 2022

Content

Elementary linear algebra and its implementation with NumPy

- Vector space
- Vectors and operations on vectors
- Dot product and norm
- Linear and affine applications
- Matrices and matrix products
- Relation with neural nets
- Datasets

We will use:

- NumPy, `https://numpy.org/`, and
- PyTorch, `http://pytorch.org/`.

A Word on Me

In August 2016, I had a work accident at LTH: Workers demolished the window of my office while I was working and without warning me.
Since then, I have a very debilitating tinnitus (ringing hears):

- I cannot use a microphone in a lecture hall
- I am very sensitive to loud noise

A word of caution for you: Be careful of renovation works, public works, loud music in parties, earphones.
Tinnitus is very annoying for some people and irreversible: You will never recover from it...

Vector Space (I)

A vector space is a set consisting of

1. Elements called vectors, represented as $\mathbb{R}^n$ tuples:
   - 2D vectors: $(2, 3)$
   - 3D vectors: $(1, 2, 3)$
   - n-dimentional vectors: $(1, 2, 3, 4, \ldots)$
2. With two operations:
   1. internal, $\mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$, the addition, denoted $+$
   2. external, $\mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$, the multiplication by a scalar (a real number), denoted $\cdot$

Concepts used in many applications, for instance mechanics

Vector Space (II)

Let $\mathbf{u}$ and $\mathbf{v}$ be two vectors:

$$\begin{aligned} \mathbf{u} &= (u_1, u_2, ..., u_n) \\ \mathbf{v} &= (v_1, v_2, ..., v_n) \end{aligned}$$

The operations:

1. Addition: $\mathbf{u} + \mathbf{v}$, a vector: $(u_1 + v_1, u_2 + v_2, ..., u_n + v_n)$
2. Multiplication by a scalar: $\lambda \cdot \mathbf{v}$, a vector. We usually drop the dot: $(\lambda u_1, \lambda u_2, ..., \lambda u_n)$.

Vectors in NumPy

```
import numpy as np

np.array([2, 3])
np.array([1, 2, 3])
```

With PyTorch

```
import torch

torch.tensor([2, 3])
torch.tensor([1, 2, 3])
```

NumPy Indices

We access vector coordinates, read or write, with NumPy indices and slices.
They are identical to Python.
Indices start at 0, following most programming languages, and contrary to the mathematical convention to start at 1.

```
vector = np.array([1, 2, 3, 4])
vector[1] # 2
vector[:1] # [1]
vector[1:3] # [2, 3]
```

Operations

- Addition

```
np.array([1, 2]) + np.array([3, 4])
# [4, 6]
np.array([1, 2, 3]) + np.array([4, 5, 6])
# [3, 7, 9]
```

- Multiplication

```
2 * np.array([1, 2])
# [2, 4]
3 * np.array([1, 2, 3])
[3, 6, 9]
```

Comparison with Lists

The same operators applied to Python list

```
[1, 2, 3] + [4, 5, 6]
# [1, 2, 3, 4, 5, 6]

3 * [1, 2, 3]
# [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Data Types in NumPy

```
np.array([1, 2, 3]).dtype
# int64

np.array([1, 2, 3], dtype='float64')
# [1.0, 2.0, 3.0]

np.array([0, 1, 2, 3], dtype='bool')
# [False,  True,  True,  True]
```

Dot Product

A Euclidian vector space consists of a vector space and a bilinear form:

$$\mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$$

$$f(\mathbf{u}, \mathbf{v}) = z$$

called the dot product and defined as

$$\begin{aligned} \mathbf{u} &= (u_1, u_2, \ldots, u_n) \\ \mathbf{v} &= (v_1, v_2, \ldots, v_n) \\ \mathbf{u} \cdot \mathbf{v} &= \sum_i u_i v_i \end{aligned}$$

Dot Product in NumPy

```
np.dot(np.array([1, 2, 3]), np.array([4, 5, 6]))
# 1 x 4 + 2 x 5 + 3 x 6
# 32
```

or

```
np.array([1, 2, 3]) @ np.array([4, 5, 6])
# 32
```

Norm

The dot product of a vector by itself defines a metric in the Euclidian vector space.
Its square root is called the norm:

$$||\mathbf{u}|| = \sqrt{\mathbf{u} \cdot \mathbf{u}}$$

It corresponds to the magnitude of the vector (its length)

Norm in NumPy

```
np.linalg.norm(np.array([1, 2, 3]))
# 3.74

torch.norm(torch.tensor([1.0, 2.0, 3.0]))
# 3.74
```

Cosine

We often use a generalized cosine to compute similarity of two vectors:

$$\cos(\widehat{\mathbf{u}, \mathbf{v}}) = \frac{\mathbf{u} \cdot \mathbf{v}}{||\mathbf{u}||.||\mathbf{v}||}.$$

Linear and Affine Maps

Simplest forms of maps:

- Linear map (function)

$$\begin{array}{rcl} \mathbb{R} & \rightarrow & \mathbb{R} \\ x & \rightarrow & f(x) = ax \end{array}$$

- Affine map (function)

$$\begin{array}{rcl} \mathbb{R} & \rightarrow & \mathbb{R} \\ x & \rightarrow & f(x) = ax + b \end{array}$$

Linear maps are defined by the properties:

$$\begin{array}{rcl} f(x+y) & = & f(x)+f(y) \\ f(\lambda x) & = & \lambda f(x) \end{array}$$

Linear Functions and Vectors

Extending linear functions to vectors: A linear combination of the coordinates

$$
\begin{array}{lcll}
\mathbb{R}^2 & \rightarrow & \mathbb{R}^2 & \\
\mathbf{x} & \rightarrow & f(\mathbf{x}) & = \mathbf{y} \\
\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} & \rightarrow & \begin{bmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \end{bmatrix} & = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \\
\mathbf{x} & \rightarrow & A\mathbf{x} & = \mathbf{y}
\end{array}
$$

With a matrix notation:

$$
A\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \mathbf{y}
$$

This is the dot product of $\mathbf{x}$ with each row of the matrix

Matrice Example

Let us give values to this system:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= y_1 \\ a_{21}x_1 + a_{22}x_2 &= y_2 \end{aligned}$$

For instance:

$$\begin{aligned} 1 \times x_1 + 2 \times x_2 &= y_1 \\ 3 \times x_1 + 4 \times x_2 &= y_2 \end{aligned}$$

We have:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 17 \\ 39 \end{bmatrix}$$

NumPy Matrices

We create the NumPy matrix as a list of lists in Python:

```
A = np.array([[1, 2],
              [3, 4]])
A @ np.array([5, 6])
# array([17, 39])
```

We access the matrix elements with indices and slices:

```
A[0, 1]
# 2
A[1, 1] = 3
```

Matrices and Rotations

From algebra courses, we know that we can use a matrix to compute a rotation of angle θ.
For a two-dimensional vector, the rotation matrix is:

$$\mathbf{R}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

Rotation of an angle of $\pi/4$ of vector $(1,1)$:

$$\begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ \sqrt{2} \end{bmatrix}$$

Rotation Matrices with NumPy

We create a NumPy rotation matrix with an angle of $\pi/4$:

```
theta_45 = np.pi / 4
rot_mat_45 = np.array([[np.cos(theta_45), -np.sin(theta_45)],
                       [np.sin(theta_45), np.cos(theta_45]])
    # array([[ 0.707, -0.707],
             [ 0.707,  0.707]])
```

and we rotate vector (1, 1) by this angle with the code:

```
rot_mat_45 @ np.array([1, 1])
  # array([1.110e-16, 1.414e+00])
```

NumPy makes a roundoff error.

Function Composition

Defined as

$$(f \circ g)(x) = f(g(x))$$

The composition of linear maps corresponds to their matrix product

$$M_{(f \circ g)} = M_f M_g.$$

We compute $M_{(f \circ g)}$ by computing the product of M_f with each column of M_g.
In NumPy and PyTorch, the operator of matrix products is `@`.

```
M_fg = M_f @ M_g
```

Rotations

For instance, the matrix of a sequence of rotations is the matrix product of the individual rotations

$$\mathbf{R}_{\theta_1}\mathbf{R}_{\theta_2} = \mathbf{R}_{\theta_1+\theta_2},$$

For a rotation of $\pi/6$ followed by a rotation of $\pi/4$, it is:

$$\mathbf{R}_{\pi/4}\mathbf{R}_{\pi/6} = \mathbf{R}_{5\pi/12}.$$

NumPy

```
theta_30 = np.pi/6
rot_mat_30 = np.array([[np.cos(theta_30), -np.sin(theta_30)],
          [np.sin(theta_30), np.cos(theta_30)]])

rot_mat_30 @ rot_mat_45

array([[ 0.25881905, -0.96592583],
       [ 0.96592583,  0.25881905]])
```

Inverse Function

Inverse function g of f defined as:

$$\forall x, (f \circ g)(x) = (g \circ f)(x) = x.$$

Denoted:

$$g = f^{-1}.$$

For matrices:

$$MM^{-1} = M^{-1}M = I,$$

where I is the identity matrix with a diagonal of ones.

NumPy Inverse

NumPy

```
np.linalg.inv(rot_mat_30)
```

```
array([[ 0.8660254,  0.5      ],
       [-0.5      ,  0.8660254]])
```

```
np.linalg.inv(rot_mat_30) @ rot_mat_30
```

```
array([[1.00000000e+00, 0.00000000e+00],
       [5.55111512e-17, 1.00000000e+00]])
```

PyTorch

```
torch.inverse(torch.from_numpy(rot_mat_30))
```

Transpose

The transpose of a matrix $A = [a_{i,j}]$ is defined as $A^\intercal = [a_{j,i}]$.
This is the flipped matrix with regard to the diagonal.
An important property:

$$(AB)^\intercal = B^\intercal A^\intercal.$$

In NumPy:

```
A = np.array([[1, 2],
              [3, 4]])
A.T
```

```
array([[1, 3],
       [2, 4]])
```

Size of a Matrix

The size of a matrix is the number of rows and number of columns. The are called the dimensions of the matrix:

- NumPy

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])
A.shape
# (2, 3)
```

- PyTorch

```
A = torch.tensor([[1, 2, 3],
                  [4, 5, 6]])
A.size()
# torch.Size([2, 3])
```

For a Vector

The dimension of the vector space (sometimes called the dimension of the vector)

```
x = np.array([1, 2, 3])
x.shape
# (3,)

x = torch.tensor([1, 2, 3])
x.size()
# torch.Size([3])
```

Transposing a Vector

- This is something that does not exist (except in Matlab)
- Why? See the definition: A vector coordinate has one index: $(x_1, x_2, ..., x_n)$.
- In NumPy, trying to transpose a vector is a frequent source of error:

```
x = np.array([1, 2, 3])
x.T
# array([1, 2, 3])
```

- Even worse in PyTorch:

```
x = torch.tensor([1, 2, 3])
x.T
# tensor([1, 2, 3])
UserWarning: The use of ‘x.T‘ on tensors of dimension
other than 2 to reverse their shape is deprecated and
it will throw an error in a future release.
```

Column and Row Vectors

We can convert a vector $(x_1, x_2, ..., x_n)$ that has no direction, neither vertical or horizontal, to a **row vector** or a **column vector**.
Such vectors are matrices. Their size is $(1 \times n)$ or $(n \times 1)$
They are also called a **row matrix** or a **column matrix**.

- Row vector (matrix):

```
x = np.array([1, 2, 3])
x.reshape((1,3))
# array([[1, 2, 3]])
# or
x.reshape((1,-1))
```

- Column vector (matrix):

```
x = np.array([1, 2, 3])
x.reshape((3,-1))
```

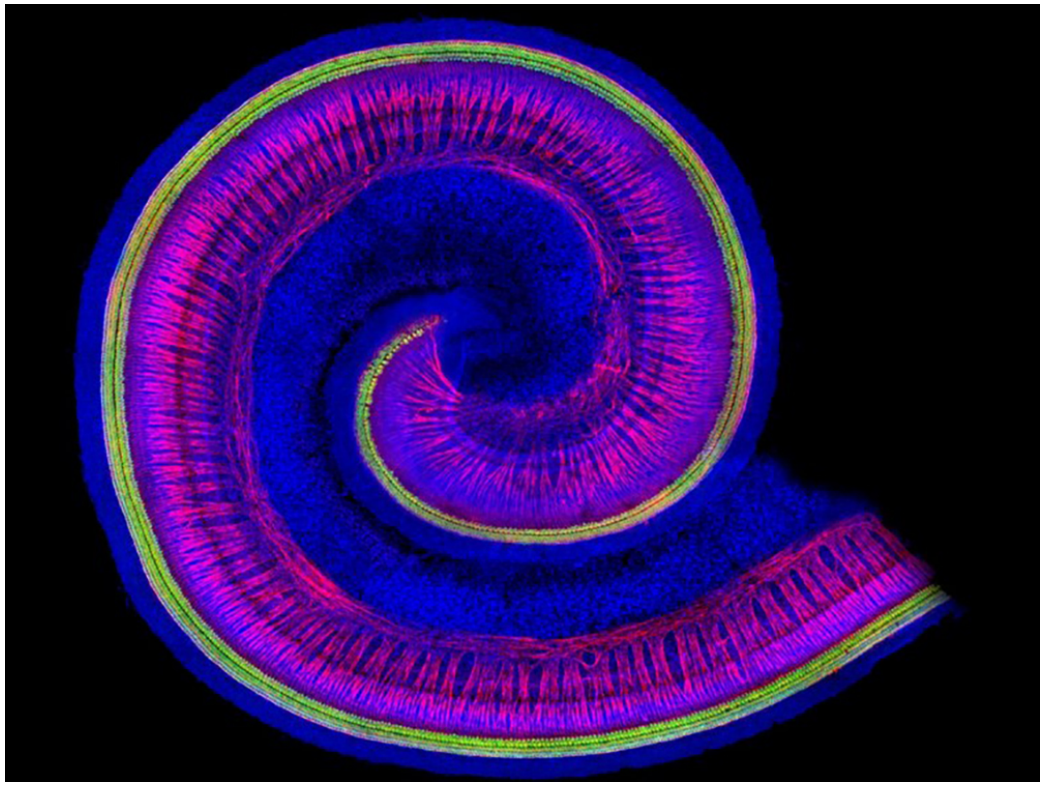
The transpose of a row vector is a column vector and vice-versa.

A Few Utilities

- `np.eye(n)` and `torch.eye(n)`
- `np.zeros((m,n))` and `torch.zeros((m,n))`
- `np.ones((m,n))` and `torch.ones((m,n))`
- `np.random.rand(m,n)` and `torch.rand(m, n)`

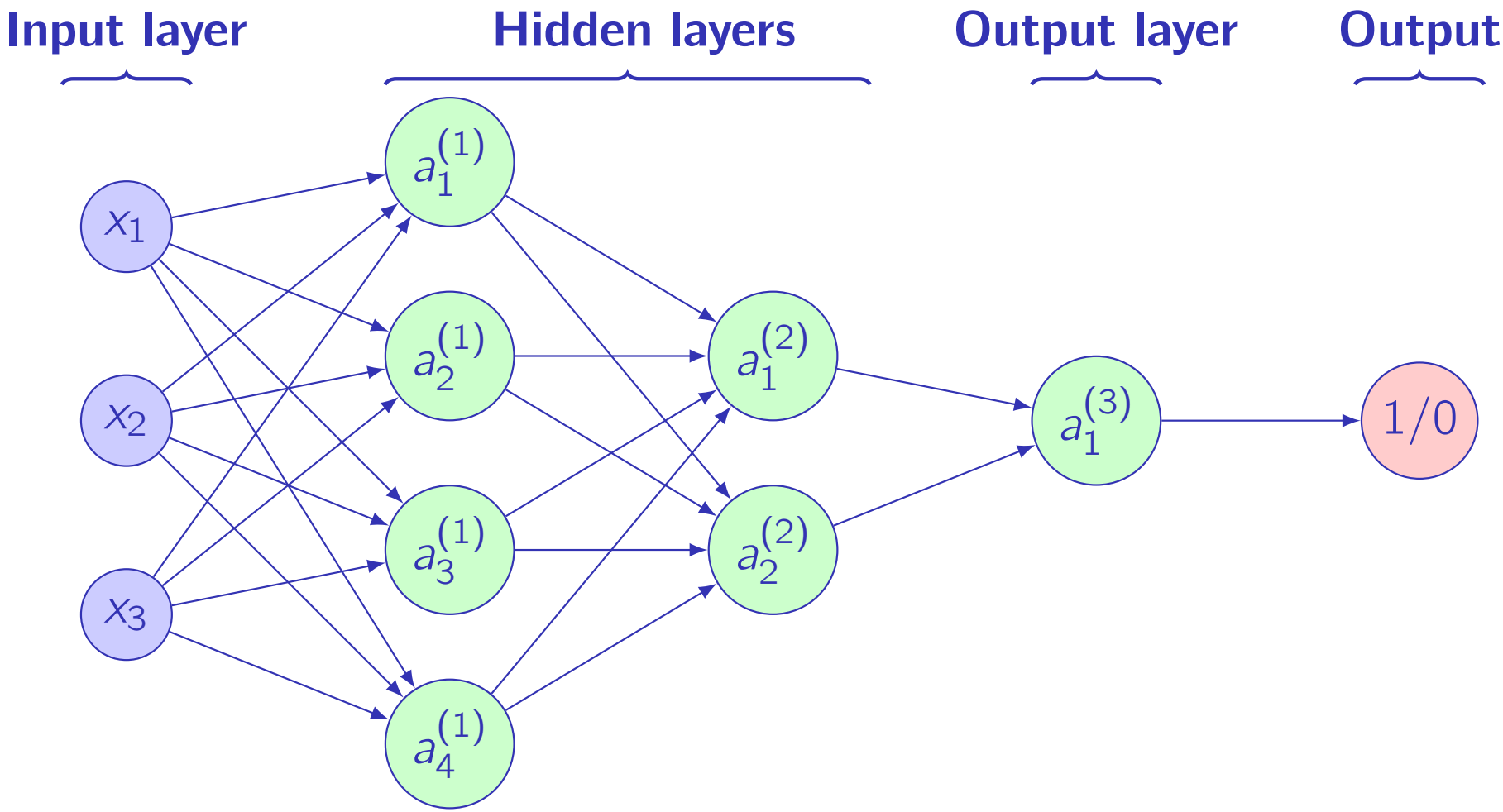
The reshaping and indexing capacities of NumPy and PyTorch are very versatile, but beyond this introduction

Neural Networks



A photomicrograph showing the classic view of the snail-shaped cochlea with hair cells stained green and neurons showing reddish-purple. [Decibel Therapeutics (https://www.decibeltx.com)]. Source: https://www.genengnews.com/insights/targeting-the-inner-ear/

Neural Networks



Applying Transformations

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

See notebook

Matrices and Keras: One Layer

```
layer1_k = keras.Sequential([
    layers.Dense(4, input_dim=3, use_bias=False)
    ])

layer1_k.weights
# array([[-0.62047374,  0.77481735,  0.01930374, -0.05884182],
#        [ 0.6178018 , -0.06696081,  0.2296443 , -0.2170434 ],
#        [ 0.13795424,  0.22487962,  0.47652638, -0.75400156]])

x_np = np.array([[1, 2, 3]])

layer1_k(x_np)
# array([[ 1.0289925,  1.3155346,  1.9081714, -2.7549334]]
x_np @ layer1_k.weights[0]
# array([[ 1.0289925,  1.3155346,  1.9081714, -2.7549334]]
```

Matrices and PyTorch: One Layer

```
layer1_t = torch.nn.Linear(3, 4, bias=False)
layer1_t.weight
# tensor([[ 0.2472, -0.4360,  0.0955],
#         [-0.4775, -0.2369,  0.0147],
#         [ 0.2489,  0.3770,  0.2392],
#         [-0.1870, -0.0463, -0.2020]])

x_t = torch.tensor([1.0, 2.0, 3.0])

layer1_t(x_t)
# tensor([-0.3382, -0.9072,  1.7203, -0.8855])
layer1_t.weight @ x_t
# tensor([-0.3382, -0.9072,  1.7203, -0.8855])
x_t @ layer1_t.weight.T
# tensor([-0.3382, -0.9072,  1.7203, -0.8855])
```

Matrices and Keras: More Layers

```
nn_k = keras.Sequential([
    layers.Dense(4, input_dim=3, use_bias=False),
    layers.Dense(2, use_bias=False),
    layers.Dense(1, use_bias=False)
    ])
nn_k.weights
# See notebook output

nn_k(x_np)
# array([[2.7943149]])
x_np @ nn_k.weights[0] @ nn_k.weights[1] @ nn_k.weights[2]
# array([[2.7943149]])
```

Matrices and PyTorch: More Layers

```
layer1_t = torch.nn.Linear(3, 4, bias=False)
layer2_t = torch.nn.Linear(4, 2, bias=False)
layer3_t = torch.nn.Linear(2, 1, bias=False)
(layer1_t.weight, layer2_t.weight, layer3_t.weight)
# See notebook output

layer3_t(layer2_t(layer1_t(x_t)))
# tensor([0.3210])
x_t @ layer1_t.weight.T @ layer2_t.weight.T @ \\
                                        layer3_t.weight.T
# tensor([0.3210])
```

Neural Networks

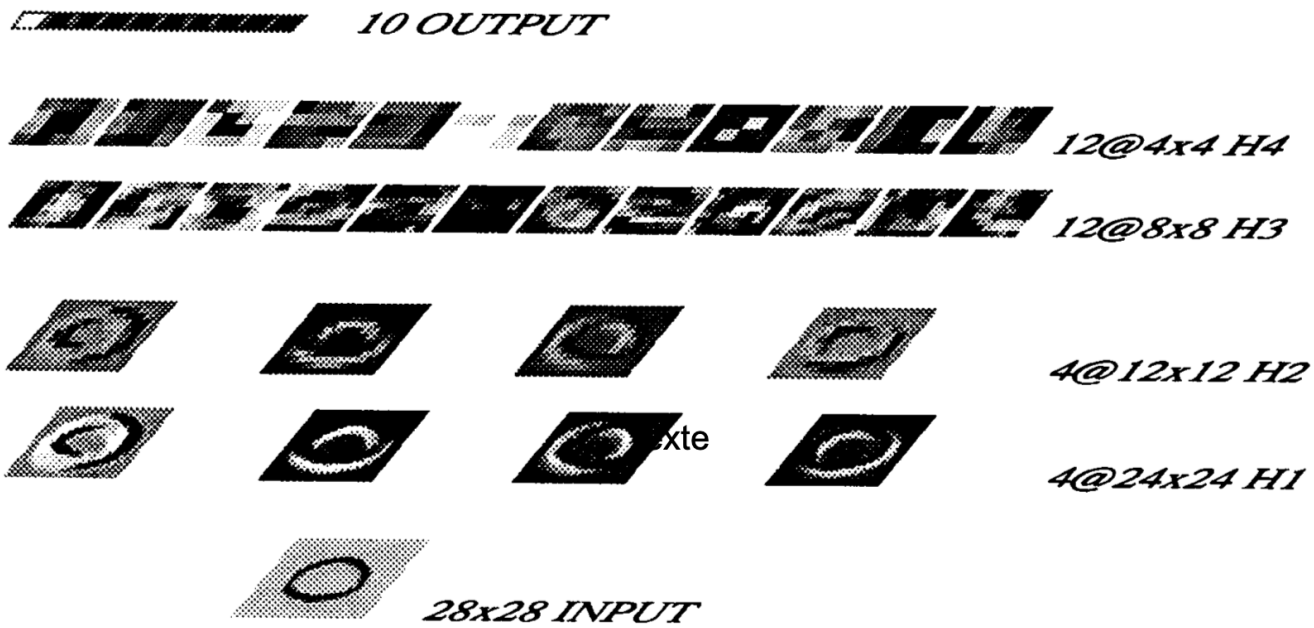


Figure 4: Network Architecture with 5 layers of fully-adaptive connections.

Lecun et al. 1990, Handwritten ZIP code recognition with multilayer networks. Proceedings of the 10th ICPR.

Datsets: Types of Iris

Iris virginica

Iris setosa

Iris versicolor

Courtesy Wikipedia

Datasets Format

In statistics, observations are traditionally arranged by rows

Fisher's Iris Dataset (1936)

180 MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS

Table I

Iris setosa				*Iris versicolor*				*Iris virginica*			
Sepal length	Sepal width	Petal length	Petal width	Sepal length	Sepal width	Petal length	Petal width	Sepal length	Sepal width	Petal length	Petal width
5·1	3·5	1·4	0·2	7·0	3·2	4·7	1·4	6·3	3·3	6·0	2·5
4·9	3·0	1·4	0·2	6·4	3·2	4·5	1·5	5·8	2·7	5·1	1·9
4·7	3·2	1·3	0·2	6·9	3·1	4·9	1·5	7·1	3·0	5·9	2·1
4·6	3·1	1·5	0·2	5·5	2·3	4·0	1·3	6·3	2·9	5·6	1·8
5·0	3·6	1·4	0·2	6·5	2·8	4·6	1·5	6·5	3·0	5·8	2·2
5·4	3·9	1·7	0·4	5·7	2·8	4·5	1·3	7·6	3·0	6·6	2·1
4·6	3·4	1·4	0·3	6·3	3·3	4·7	1·6	4·9	2·5	4·5	1·7
5·0	3·4	1·5	0·2	4·9	2·4	3·3	1·0	7·3	2·9	6·3	1·8
4·4	2·9	1·4	0·2	6·6	2·9	4·6	1·3	6·7	2·5	5·8	1·8
4·9	3·1	1·5	0·1	5·2	2·7	3·9	1·4	7·2	3·6	6·1	2·5
5·4	3·7	1·5	0·2	5·0	2·0	3·5	1·0	6·5	3·2	5·1	2·0
4·8	3·4	1·6	0·2	5·9	3·0	4·2	1·5	6·4	2·7	5·3	1·9
4·8	3·0	1·4	0·1	6·0	2·2	4·0	1·0	6·8	3·0	5·5	2·1
4·3	3·0	1·1	0·1	6·1	2·9	4·7	1·4	5·7	2·5	5·0	2·0
5·8	4·0	1·2	0·2	5·6	2·9	3·6	1·3	5·8	2·8	5·1	2·4
5·7	4·4	1·5	0·4	6·7	3·1	4·4	1·4	6·4	3·2	5·3	2·3
5·4	3·9	1·3	0·4	5·6	3·0	4·5	1·5	6·5	3·0	5·5	1·8
5·1	3·5	1·4	0·3	5·8	2·7	4·1	1·0	7·7	3·8	6·7	2·2
5·7	3·8	1·7	0·3	6·2	2·2	4·5	1·5	7·7	2·6	6·9	2·3
5·1	3·8	1·5	0·3	5·6	2·5	3·9	1·1	6·0	2·2	5·0	1·5
5·4	3·4	1·7	0·2	5·9	3·2	4·8	1·8	6·9	3·2	5·7	2·3
5·1	3·7	1·5	0·4	6·1	2·8	4·0	1·3	5·6	2·8	4·9	2·0

Matrices and Datasets

Representing the $\mathbf{x}$ input as a column vector and $\mathbf{W}$, denoting the weight matrix, we just have to transpose the matrix product:

$$\begin{aligned}(\mathbf{W}\mathbf{x})^\intercal &= \mathbf{x}^\intercal \mathbf{W}^\intercal, \\ &= \hat{y}.\end{aligned}$$

$\mathbf{x}^\intercal$ is now a row and we can apply the product to the whole $\mathbf{X}$ dataset yielding a column vector of predicted outputs:

$$\mathbf{X}\mathbf{W}^\intercal = \hat{\mathbf{y}}.$$

More on Datasets

See the notebook: https://github.com/pnugues/ilppp/blob/master/programs/appB/python/tourofpython.ipynb

Tensors

Pandas