

딥러닝을 활용한 디지털 영상 처리

Digital Image Processing via Deep Learning

Lecture 6 – Python Revision

Python – Numpy functions

Some important Numpy functions we will be revising;

Array Creation:

- `np.array()`
- `np.zeros()`
- `np.ones()`
- `np.full()`

Array Manipulation:

- `np.insert()`

Random Number Generation

- `np.random.rand()`

Matrix Operation:

- `np.transpose()`
- `np.dot()`

Maths:

- `np.sum()`
- `np.maximum()`

Creating Arrays

```
>>> arr = np.array([2, 1, 5, 3, 7, 4, 6, 8])
```

Converts a list/lists to an array

```
>>> np.zeros(2)  
array([0., 0.])
```

Creates an array of size 2(the input) and initialise the values to 0 in floating point.

```
>>> np.ones(2)  
array([1., 1.])
```

Creates an array of size 2(the input) and initialise the values to 1 in floating point.

```
>>> np.full((2, 2), np.inf)  
array([[inf, inf],  
       [inf, inf]])  
>>> np.full((2, 2), 10)  
array([[10, 10],  
       [10, 10]])
```

Creates an array of given dimensions filled with the given value.

Inserting a value

```
>>> a = np.array([[1, 1], [2, 2], [3, 3]])
>>> a
array([[1, 1],
       [2, 2],
       [3, 3]])
>>> np.insert(a, 1, 5)
array([1, 5, 1, ..., 2, 3, 3])
```

An insert function inserts a given value to a given index of an array. Notice that the array has been flattened after the insertion.

numpy.**insert**(arr, obj, values, axis=None) [\[source\]](#)


Insert values along the given axis before the given indices.

Parameters: arr : *array_like*

Input array.

obj : *int, slice or sequence of ints*

Object that defines the index or indices before which *values* is inserted.

 **New in version 1.8.0.**

Support for multiple insertions when *obj* is a single scalar or a sequence with one element (similar to calling insert multiple times).

values : *array_like*

Values to insert into *arr*. If the type of *values* is different from that of *arr*, *values* is converted to the type of *arr*. *values* should be shaped so that **arr[...,obj,...] = values** is legal.

axis : *int, optional*

Axis along which to insert *values*. If *axis* is None then *arr* is flattened first.

Returns:

out : *ndarray*

A copy of *arr* with *values* inserted. Note that **insert** does not occur in-place: a new array is returned. If *axis* is None, *out* is a flattened array.

Random Number Generation

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618], #random
       [ 0.37601032,  0.25528411], #random
       [ 0.49313049,  0.94909878]]) #random
```

Generates an array of random numbers between 0 and 1 (sampled from a uniform distribution) with given dimensions.

`random.rand(d0, d1, ..., dn)`

Random values in a given shape.

Note

This is a convenience function for users porting code from Matlab, and wraps `random_sample`. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like `numpy.zeros` and `numpy.ones`.

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

Parameters: *d0*, *d1*, ..., *dn* : *int, optional*

The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

Returns: *out* : *ndarray, shape (d0, d1, ..., dn)*

Random values.

Matrix Operation

```
>>> np.dot(3, 4)  
12
```

Numpy.dot() function enables matrix multiplication

```
>>> a = [[1, 0], [0, 1]]  
>>> b = [[4, 1], [2, 2]]  
>>> np.dot(a, b)  
array([[4, 1],  
       [2, 2]])
```

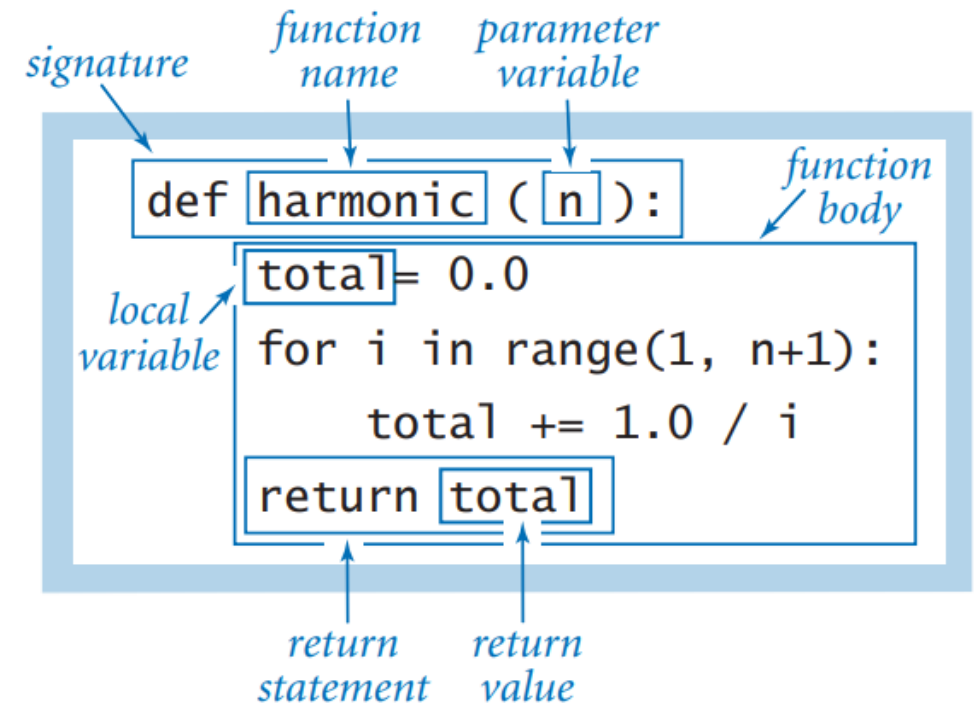
$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 4 & 1 \\ 2 & 2 \end{pmatrix} = \begin{pmatrix} 4 & 1 \\ 2 & 2 \end{pmatrix}$$

Functions

Function is defined by 'def' statement.
Followed by the name of the function, and input required.
This line is called "signature".

The function body contains lines of code for the function's purpose.

Return statement is optional. The function may return a specific value or just process the function body.



Anatomy of a function definition

From Previous Lecture

Let Input vector, $\mathbf{x} = \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix}$ ← bias

Then,

$$\mathbf{h} = a(\mathbf{w}^{(1)}\mathbf{x})$$

Where

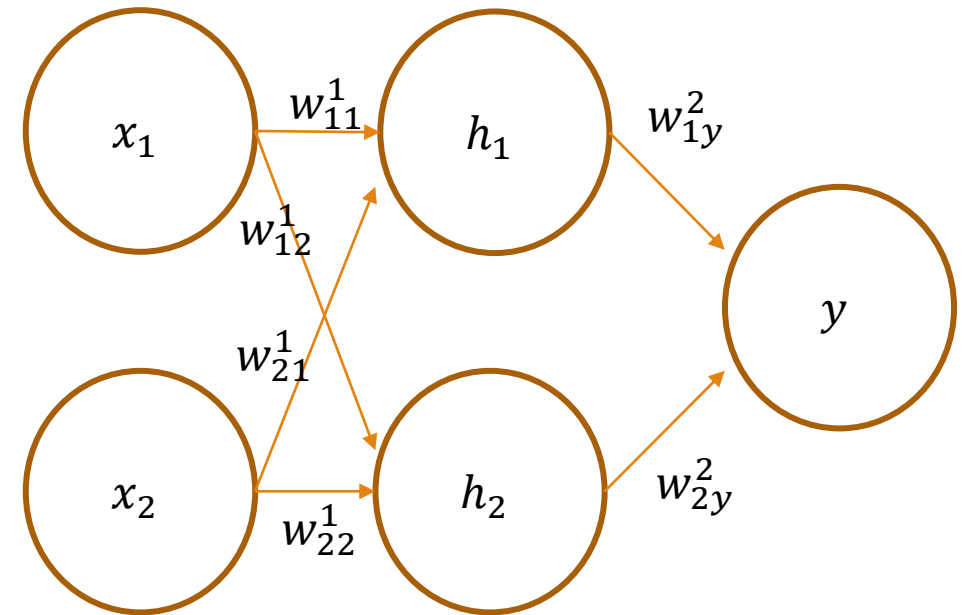
$$\mathbf{w}^{(1)} = \begin{pmatrix} w_{b1}^1 & w_{11}^1 & w_{21}^1 \\ w_{b2}^1 & w_{12}^1 & w_{22}^1 \end{pmatrix}$$

Then,

$$y = a(\mathbf{w}^{(2)}\mathbf{h})$$

Where

$$\mathbf{w}^{(2)} = \begin{pmatrix} w_{bh}^2 & w_{1h}^2 & w_{2h}^2 \end{pmatrix}$$




We denote the activation function as $a()$


Initializing Parameters

Let's start with writing a function that initializes network parameters, w .

```
def init_parameters_fixed(x,y,p):  
    return np.full((x,y), p)  
  
def init_parameters(x, y):  
    return np.random.rand(x, y)
```



Creates an array of a given dimension (x,y) filled with a given value p.



Creates an array of a given dimension (x,y) filled with random numbers between 0 and 1.

Initializing Network (all parameters)

```
def init_linear_network(network_info):  
    # network_info must be a list containing [input dim, num of nodes in hidden layers, output dim], eg: [3,4,4,4,1]  
    param = []  
    for i in range(len(network_info) - 1):  
        param.append(init_parameters(network_info[i+1], network_info[i]+1))  
    param = np.array(param)  
    return param
```

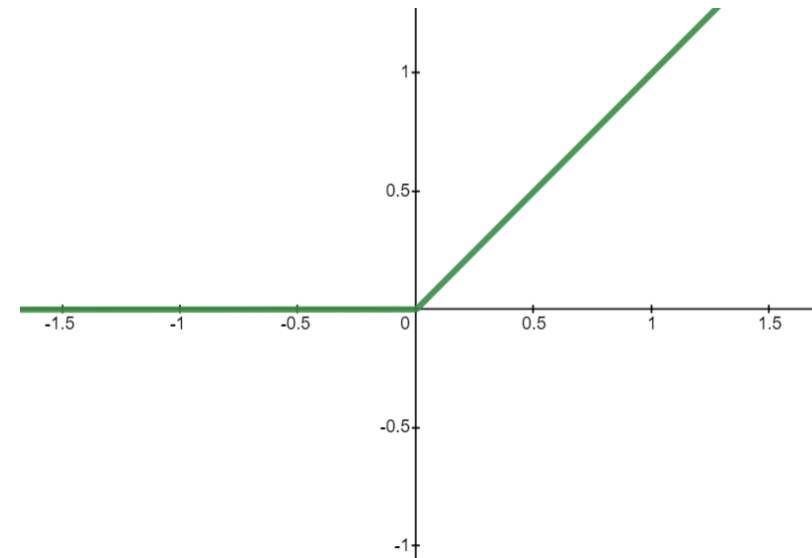
- The function takes a list as an input that contains information about layer structures. (2,2,1) means 2 input values, one hidden layer with 2 nodes and a single value output.
- Sets an empty list called param → this list will contain information of all parameters in the network
- Loops for the number of weight matrixes in the list. For (2,2,1) there will be two sets of weight matrixes.
- Appends the param list with the weight matrix between the layers denoted by i and i+1
- Returns param as an array format.

Activation function - ReLU

ReLU(Rectified Linear Unit):

$$a(x) = \max(0, x)$$

```
def ReLU(x):  
    return np.maximum(0, x)
```



Forward Propagation

```
def forward(x, param):
    num_param_layers = len(param)
    h_history = []
    x = np.insert(x, 0, [1])
    x = np.transpose(x)
    h = ReLU(np.dot(param[0], x))
    h = np.insert(h, 0, [1])
    h_history.append(h)
    if num_param_layers > 2:
        for j in range(num_param_layers-3):
            h = ReLU(np.dot(param[j], h))
            h = np.insert(h, 0, [1])
            h_history.append(h)
    y = ReLU(np.dot(param[-1], h))
    return np.array(y)
```

Task: fill in the blanks!

Let Input vector, $x = \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix}$ ← bias

$h = a(w^{(1)}x)$

Bias?

$a(w^{(n)}h_n)$

Mean Square Loss

```
def L2Loss(output, label):  
    total_loss = 0  
    for i in range(len(output)):  
        y = output[i]  
        t = label[i]  
        total_loss += 0.5 * (np.sum((y-t)**2))  
    mean_loss = total_loss / len(output)  
    return mean_loss
```

Computes a loss from batch data

Next Lecture

Object Oriented Programming

Object Oriented
Programming
with Python

