Q. Why use vectorization?

→ Makes your code shorter and much more efficient

→ Enables you to take advantage of modern numeric linear algebraic libraries

→ Allows the program to take advantage of the computer's GPU hardware.

example

Parameters and features

$\vec{w} = [w_1, w_2, w_3]$

b is a number

$\vec{x} = [x_1, x_2, x_3]$

vectors → multiple parameters

To denote this using python we take advantage of python's NumPy library.

NumPy

```
w = np.array([1.0, 2.5, -3.3])
b = 4
x = np.array([10, 20, 30])
```

w[0] w[1] w[2] x[0] x[2] x[1]

syntax

Mathematically we write the algorithm like this in python

$f_{w,b}(x) = w[0] * x[0] + w[1] * x[1] + w[2] * x[2] + b$

→ This code is very inefficient for practical purposes

Using for loop :-

$$f_{w,b} = \left(\sum_{j=1}^{n} w_j x_j\right) + b \quad \Rightarrow \quad \sum_{j=1}^{n} \rightarrow j = 1 \ldots n \;,\; 1,2,3$$

```
f = 0
for j in range (0, n):
    f += wj * xj
f += b
```

this is more efficient than the previous code, but not as efficient as vectorization

Using vectorization :-

$$f_{w,b}(x) = \vec{w} \cdot \vec{x} + b$$

mathematically

```
library numpy
f = np.dot(w, x) + b
    function in numpy library
```

→ faster and shorter

The ability of numpy library to use parallel hardware (GPUs for accelerating machine learning) makes it more efficient than for loop.

Usage of parallel hardware

without vectorization

```
for j in range(16):
    f = f + w[j] * x[j]
```

t_0 ← first time frame

$f + w[0] * x[0]$

t_1

$f + w[1] * x[1]$

...

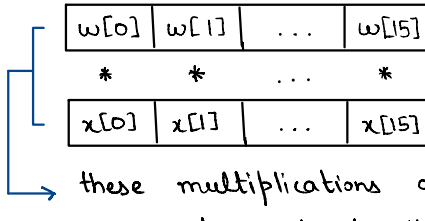
t_{15}

$f + w[15] * x[15]$

with vectorization

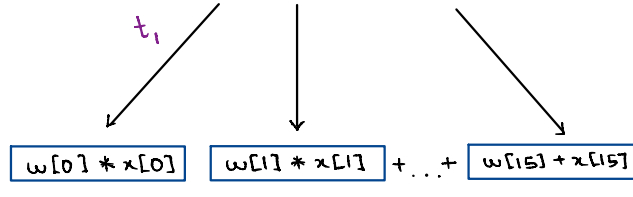
```
np.dot(w, x)
```

t_0



these multiplications are being done in parallel using computer's hardware

t_1



As we can see without vectorization, the code runs n-1 times, but with vectorization the code runs only twice and thus is much more efficient.

This efficiency is more noticeable when we have large datasets.

Gradient Descent using vectorization

$\vec{w} = (w_1 \quad w_2 \quad \ldots \quad w_{16})$ ← parameters

$\vec{d} = (d_1 \quad d_2 \quad \ldots \quad d_{16})$ ← derivatives

syntax

```
w = np.array([0.5, 1.3, ... 3.4])
d = np.array([0.3, 0.2, ... 0.4])
```

→ We have to compute $w_j = w_j - 0.1 d_j$ for $j = 1..16$ (α = learning rate)

without vectorization

$w_1 = w_1 - 0.1 d_1$

$w_2 = w_2 - 0.1 d_2$

$w_3 = w_3 - 0.1 d_3$

.
.
.

$w_{16} = w_{16} - 0.1 d_{16}$

```
for j in range(16):
  w[j] = w[j] - 0.1 * d[j]
```

with vectorization

$\vec{w} = \vec{w} - 0.1 \vec{d}$

parallel processing hardware works something like this

```
w = w - 0.1 * d
```