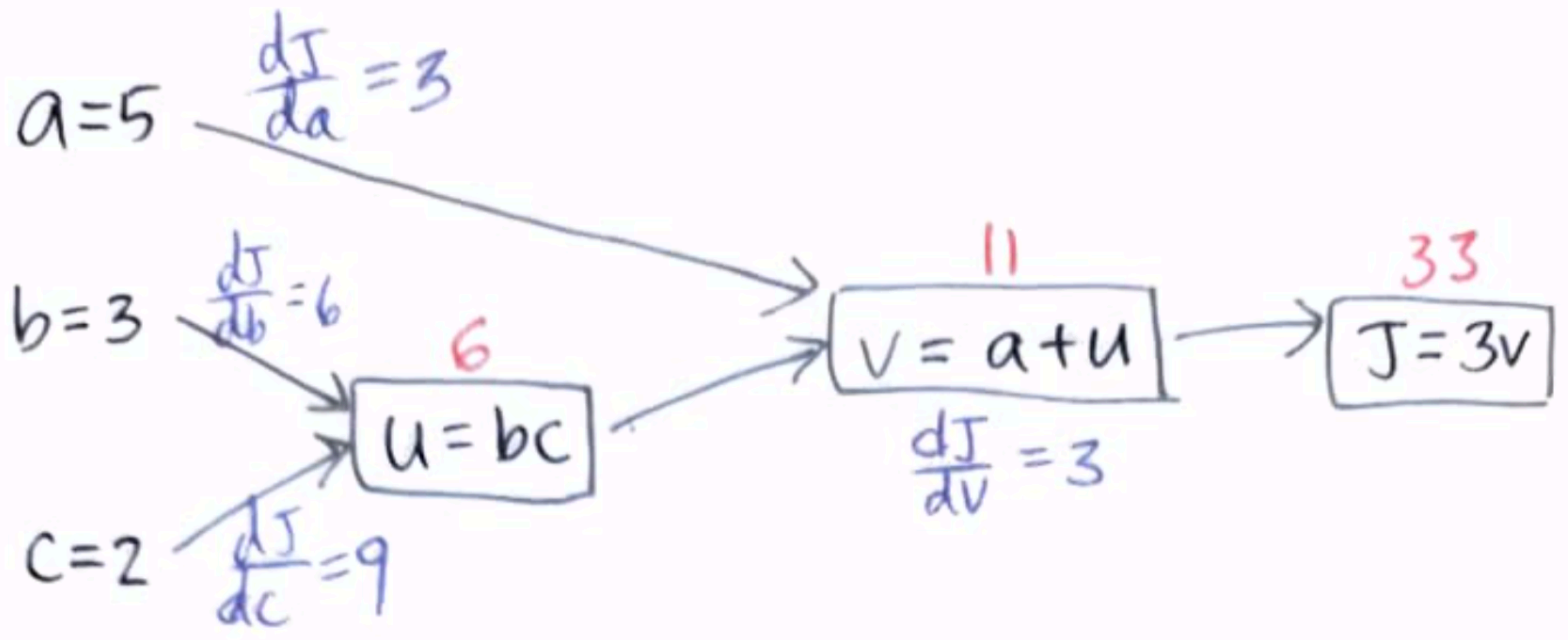
Neural Networks and Deep Learning

박달님 04.30.21

Derivatives with a Computational Graph



$$\frac{dJ}{dv} = 3$$

$$\frac{dJ}{da} = 3 = \frac{dJ}{dv} \cdot \frac{dv}{da}$$

$$3 \times 1$$

$$\frac{dJ}{du} = \frac{dJ}{dv} \cdot \frac{dv}{du} = 3 \times 1 = 3$$

$$\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db} = 3 \times 2 = 6$$

$$\frac{dJ}{dc} = \frac{dJ}{du} \cdot \frac{du}{dc} = 3 \times 3 = 9$$

Logistic Regression Gradient Descent

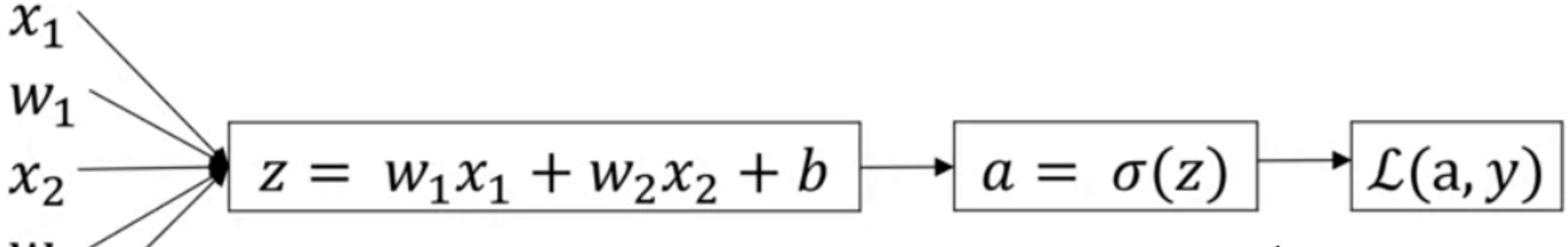
$$z = w^T x + b$$

$$\hat{y} = a = \sigma(z)$$

$$\mathcal{L}(a, y) = -(y \log(a) + (1 - y) \log(1 - a))$$

A = output of logistic regression, y = ground truth label

x1, x2와 같은 2개의 특성
z를 산출하기 위해서 w1, w2, b를 입력
로지스틱 회귀 분석 - loss 값을 줄이기 위해 w와 b의 파라미터를 변형
Loss를 계산하기 위해서 derivative을 구함



$$dz = \frac{dL}{da} \times \frac{da}{dz} = \frac{dL}{dz} = \frac{dL(a,y)}{dz} = a - y$$

$$\frac{dL(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$\frac{dL}{dw1} = x1dz \qquad \frac{dL}{dw2} = x2dz \qquad db = dz$$

$$w1 = w1 - \alpha dw1$$

$$w2 = w2 - \alpha dw2$$

$$b = b - \alpha db$$

Gradient Descent on m Training Examples

비용함수 정의

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y)$$

$$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$$

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial w_i}\mathcal{L}(a^{(i)}, y^{(i)})$$

$J=0;\ dw=0;\ dw_2=0;\ db=0$

① For loop

For $i=1$ to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

② For loop

$$\left.\begin{aligned} dw_1 &\mathrel{+}= \cancel{x}_1^{(i)}\, dz^{(i)} \\ dw_2 &\mathrel{+}= x_2^{(i)}\, dz^{(i)} \end{aligned}\right]\ n=2$$

$db \mathrel{+}= dz^{(i)}$

$J /= m$

$dw_1 /= m;\ dw_2 /= m;\ db /= m$

$$dw_1 = \frac{\partial J}{\partial w_1}$$

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

Vectorization

- 코딩에서 foor loop들을 제거하는 기술
- 딥러닝 - 큰 데이터세트 트레이닝. 코딩을 빨리 진행하는 것이 매우 중요.
- 로지스틱 회귀분석 z = w^Tx + b

Non-vectorized Implementation:

```
Z = 0
for I in range(n-x):
    Z += w[i] * x[i]
Z += b
```

매우 느리다.

Vectorized Implementation:

```
Z = np.dot(w,x) + b
```

더 빠르다.

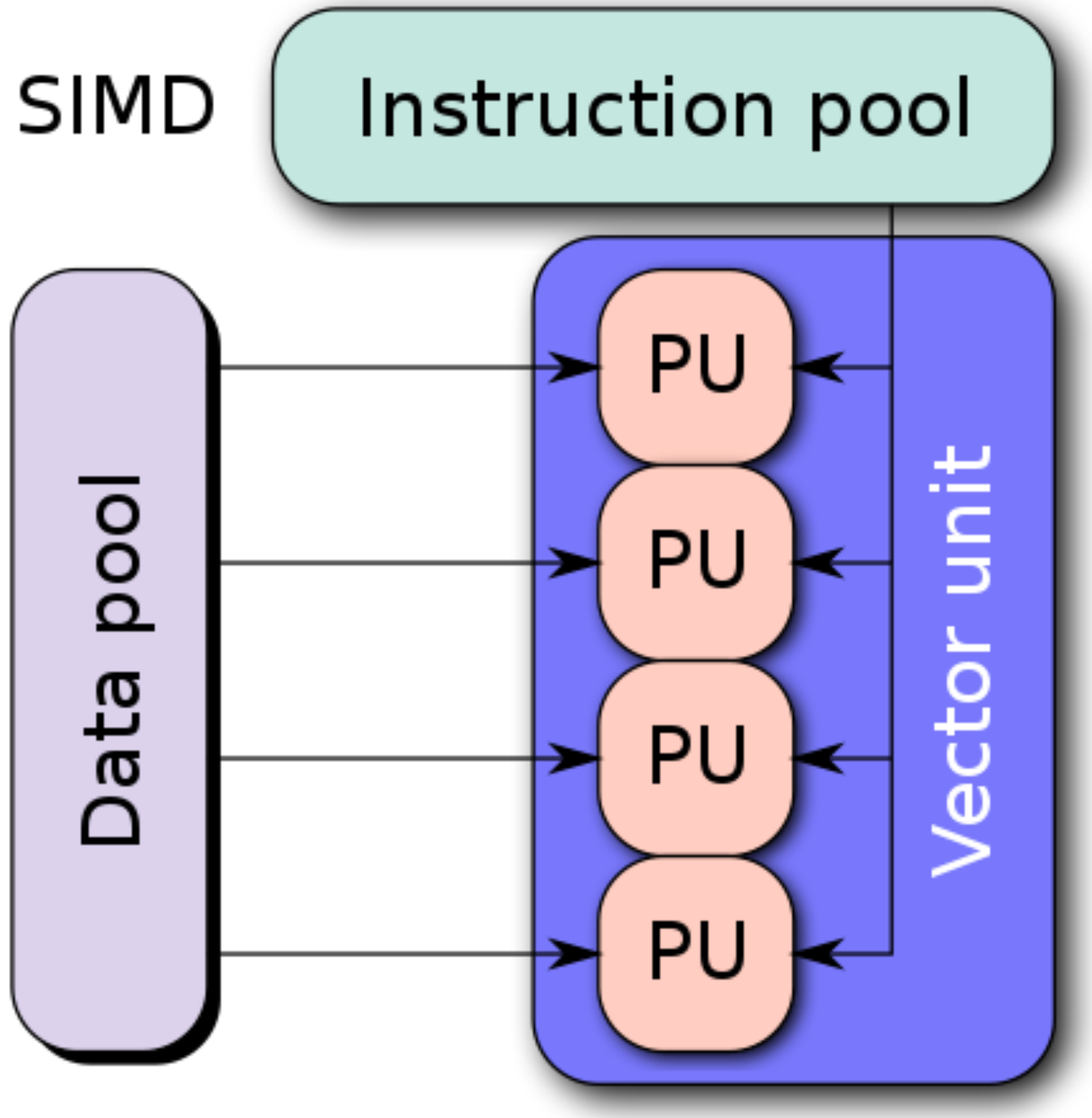
Vectorization이 필요한 이유

```
a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()
c = np.dot(a,b)
toc = time.time()

print(c)
print("Vectorized version:" + str(1000*(toc-tic)) +"ms")

c = 0
tic = time.time()
for i in range(1000000):
    c += a[i]*b[i]
toc = time.time()

print(c)
print("For loop:" + str(1000*(toc-tic)) + "ms")
```

```
250286.989866
Vectorized version:1.5027523040771484ms
250286.989866
For loop:474.29513931274414ms
```

- 백만 다이메션을 만든다음 실행하면
- Vectorized version은 1.5 ms, non-vectorized version은 474ms가 소요됨
- 거의 300배 더 빠르다
- 훨씬 빠르게 결과값을 알아낼 수 있다

CPU & GPU

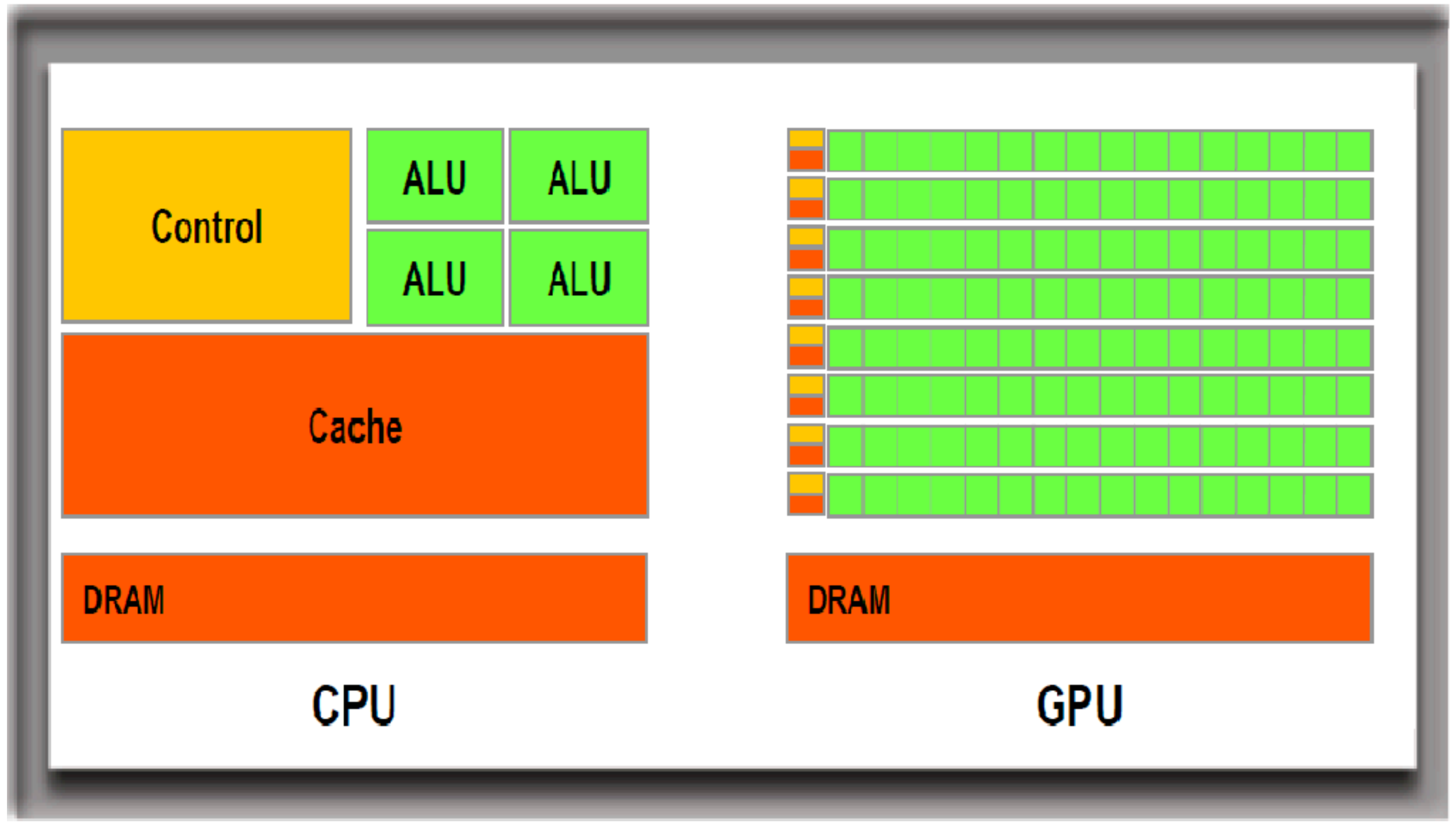


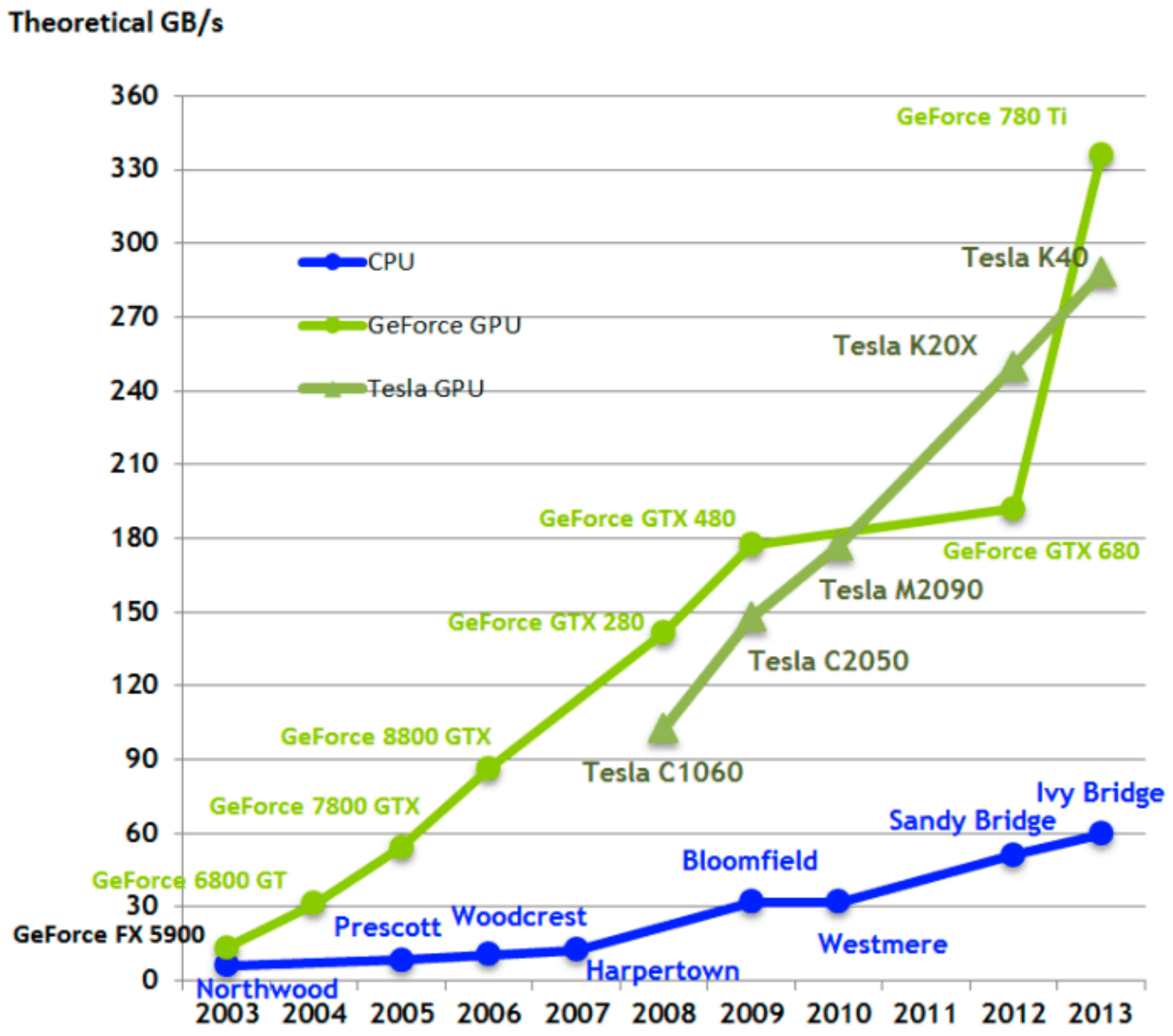
- ‘Scalable Deep Learning’ 도입이 GPU(Graphic Processing Unit)에서 이루어진다
- 하지만 방금 Jupitor notebook의 데모는 모두 CPU에서 이루어졌다
- CPU와 GPU 모두 parallelization instruction이 있다
  - SIMD instructions (single instruction multiple data)
    - 여러 데이터 포인트에서 동시에 동일한 작업을 수행하는 여러 처리 요소가 있는 컴퓨터
    - 데이터 수준 병렬 처리를 이용하지만 동시성은 사용하지 않는다. 동시 (병렬) 계산이 있지만 주어진 순간에는 단일 프로세스 (명령)만 있다
    - 이런 built in function을 이용하면 np.function 이나 다른 for loop의 도입이 필요 없는 기능을 파이썬 Pi가 parallelism을 활용할 수 있게 계산을 빨리 처리하도록 해준다
  - GPU가 특별히 SIMD calculations에 뛰어나기 때문에 그렇다
  - 하지만 CPU가 GPU보다 못하더라도 CPU도 나쁘지 않다

CPU & GPU 아키텍처 주요 차이점

- CPU가 광범위한 작업을 신속하게 처리하도록 설계되었지만 실행할 수 있는 작업의 동시성이 제한됨. 더 복잡한 단일 계산을 순차적으로 처리하는데 가장 적합함.

- GPU는 고해상도 이미지와 비디오를 동시에 빠르게 렌더링 하도록 설계. 여러 가지이지만 더 간단한 계산을 병렬로 처리하는데 적합함.





Vectorizing Logistic Regression

m training examples

$z^{(1)} = w^T x^{(1)} + b$ $z^{(2)} = w^T x^{(2)} + b$ $z^{(3)} = w^T x^{(3)} + b$

$a^{(1)} = \sigma(z^{(1)})$ $a^{(2)} = \sigma(z^{(2)})$ $a^{(3)} = \sigma(z^{(3)})$ ... m times

training inputs $\rightarrow X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$

in python (n_x, m)

$\mathbb{R}^{n_x \times m}$ matrix

1xm matrix row vector

$Z = [z^{(1)}\ z^{(2)} \ldots z^{(m)}] = w^T X + [b\ b \cdots b] = [w^T x^{(1)} + b \quad w^T x^{(2)} + b \quad \cdots \quad w^T x^{(m)} + b]$

1xm matrix row vector

$\downarrow z^{(1)}$ $\downarrow z^{(2)}$ $\downarrow z^{(m)}$

$w^T \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$

row vector

Take this matrix stack it horizontally, get Z

1xm matrix

$Z = np.dot(w.T, X) + b$

vector

python automatically expands b to 1xm vector

"Broadcasting"