Week 2 Vectorization

In the deep learning era vectorization is a key skill.

$z = w^T x + b$ $\quad w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \quad x = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$ $\quad w \in \mathbb{R}^{n_x}$ $\quad x \in \mathbb{R}^{n_x}$

non-vectorized

```
z = 0
for i in range(n_x):
    z += w[i] * x[i]
z += b
```

this is going to be really slow

Vectorized

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

($w^T x$)

in contrast, we just transpose x directly this is going to be much faster

```
import numpy as np

a = np.array([1, 2, 3, 4])

import time

a = np.random.rand(1,000,000)
b = np.random.rand(1,000,000)

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(1,000,000):
    c += a[i]*b[i]
toc = time.time()

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

Vectorized version ~ 1.5ms
Non-vectorized ~ 500ms

Non-vectorized version runs about 300 times faster 300x

Both CPUs and GPUs have parallelization instructions. "SIMD" instructions Single Instruction Multiple Data. If you use numpy functions you can take much better advantage of parallelism.

True of both but GPUs are remarkably good at SIMD calculations. CPUs are not too bad though.

More vectorization Examples

The key rule of thumb: avoid explicit for-loops whenever possible.

It's not always possible to avoid for-loops but you want to look for every way to.

Want: compute vector u as the product of matrix A times some vector v.

$$u = Av$$
$$u_i = \sum_j A_{ij} A_j$$

```
u = np.zeros(n, 1)
for i...
  for j...
    u[i] += A[i][j] * v[j]
```

non-vectorized (slow)

```
u = np.dot(A, v)
```

vectorized (way faster)
eliminates 2 for loops

Want: apply the exponential operation on every element of a matrix/vector

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \qquad u = \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

non-vectorized

```
u = np.zeros((n,1))
for i in range(n):
    u[i] = math.exp(v[i])
```

vectorized

```
import numpy as np
u = np.exp(v)
```

numpy has many vectorized functions

```
np.log(v)
np.abs(v)
np.maximum(v,0)   # element-wise max (compared to 0 here)
v**2              # element-wise square of each element of v
1/v               # element-wise inverse
```

In our logistic regression example we had 2 for-loops

$J = 0$, ~~$dw1 = 0$, $dw2 = 0$~~, $db = 0$

→ for $i = 1$ to m:

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

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

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

$dz^{(i)} = a^{(i)}(1 - a^{(i)})$

→ ~~for $j = 1$ to n_x:~~

~~$dw_j \mathrel{+}= x_j^{(i)} dz^{(i)}$~~ ⇒ $dw \mathrel{+}= x^{(i)} dz^{(i)}$

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

$J = J/m$, ~~$dw_j = dw_j/m$~~ ..., $db = db/m$

Let's look at the second for loop. Make dw a vector $dw = np.zeros((n_x, 1))$

$dw = dw/m$

Removed the for loop over the features n_x

Still have for loop over all the training examples but you can also get rid of this with vectorization.

Vectorizing Logistic Regression

Implement a single elevation of gradient descent with respect to an entire training set without using a single explicit for loop.

Forward prop:

1st example: $z^{(1)} = w^T x^{(1)} + b$, $a^{(1)} = \sigma(z^{(1)})$

2nd example: $z^{(2)} = w^T x^{(2)} + b$, $a^{(2)} = \sigma(z^{(2)})$ ... m

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \qquad \mathbb{R}^{n_x \times m}$$

recall that we stacked our inputs in a (n_x, m) matrix

construct a $(1, m)$ matrix, a row vector of zs

$$[z^{(1)} \; z^{(2)} \cdots z^{(m)}] = \underbrace{w^T X + [b \; b \ldots b]}_{(1 \times m) \text{ row vector}}$$

↑ Can be expressed as

row vector

$$w^T \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} = w^T X = [w^T x^{(1)} \; w^T x^{(2)} \cdots w^T x^{(m)}] + [b \; b \ldots b]$$

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

↑ $1 \times m$ vector

first element is exactly definition of $z^{(1)}$, second element $z^{(2)}$, and so on.

Just as X is what you obtained by stacking your training examples horizontally We're going to define capital $Z = [z^{(1)} \; z^{(2)} \cdots z^{(m)}]$

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

↖ b is a $(1,1)\mathbb{R}$ real number, when you do this python automatically takes the real number b and expands it out to a $1 \times m$ row vector. Called broadcasting in python.

now we want: $[a^{(1)} \; a^{(2)} \cdots a^{(m)}] = A = \sigma(Z)$

So instead of looping over M training examples we can just use these two formulas for capital Z and capital A to compute all of the lowercase z's and a's all at the same time. Very efficient way to compute all of the activations, a's.

Vectorizing Logistic Regression's Gradient Computation

we want ↪ $dz^{(1)} = a^{(1)} - y^{(1)}$ $\quad dz^{(2)} = a^{(2)} - y^{(2)}$ .... and so on for all training examples.

$$dZ = [dz^{(1)}\ dz^{(2)} \dots dz^{(m)}]$$

$(1, m)$ dims matrix

$$A = [a^{(1)} \dots a^{(m)}] \qquad Y = [y^{(1)} \dots y^{(m)}]$$

$$dZ = A - Y = [a^{(1)} - y^{(1)} \quad a^{(2)} - y^{(2)} \dots a^{(m)} - y^{(m)}]$$

previously we had gotten rid of one for loop ($dw_1, dw_2 \dots$) and replaced it with just dw. Now we still have dws and dbs for each training example

$dw = 0$ ← initialize → $db = 0$

$dw += x^{(1)} dz^{(1)}$ $\quad db += dz^{(1)}$

$dw += x^{(2)} dz^{(2)}$ $\quad db += dz^{(2)}$

$dw : x^{(m)} dz^{(m)}$ $\quad \vdots$

$db += dz^{(m)}$

then divide by m

$dw /= m$ $\quad db /= m$

now we want to vectorize this too.

We are really just doing

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$$

$$dw = \frac{1}{m}(X\, dZ^T)$$

$$db = \frac{1}{m}(\text{np.sum}(dZ))$$ ← lines you use to compute dw + db

$$= \frac{1}{m} \begin{bmatrix} | & & | \\ x^{(1)} & \dots & x^{(m)} \\ | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$

↑ (transposed $1 \times m$ row vector)

$$= \frac{1}{m}\left[x^{(1)} dz^{(1)} + \dots + x^{(m)} dz^{(m)} \right]$$

$n \times 1$

putting this all together:

$$Z = w^T X + b$$
$$= \text{np.dot}(w.T, X) + b$$
$$A = \sigma(Z)$$
$$dZ = A - Y$$
$$dw = \frac{1}{m} X dZ^T$$
$$db = \frac{1}{m} \text{np.sum}(dZ)$$

this is both forward prop and back prop: computes the predictions and the derivatives for all training examples without using a for loop

then you would perform the gradient descent update

$w := w - \alpha dw$

$b := b - \alpha db$

which would be one step of gradient descent for logistic regression

You still need and outer for loop for multiple steps of gradient descent.

Broadcasting in Python

example — Calories from Carbs, Proteins, Fats in 100g of different foods

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

59 cal

Calculate % of calories from carbs, proteins, fats

Calories from carbs in Apples $56/59 = 94.9\%$

So what you really want to do is sum all four columns of the matrix to get total nums of cals Then divide throughout the matrix so as to get the % of cals from C, P, F, for each of the 4 foods

The Question is

Can you do this without an explicit for loop?

Yes:

set the 3,4 matrix = A

Sum the columns In second line

redundant call for clarity

```
cal = A.sum(axis=0)
percentage = (A / cal.reshape(1,4)) * 100
```

sum(axis=0) means you want to sum vertically ↓ (0)

axis=1 ← sums horizontally → (1)

reshape is a constant time call so don't hesitate to use it

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100$$

Broadcasting auto expands this into a 4 x 1 vector

$$\Rightarrow \begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100\begin{bmatrix}1\\1\\1\\1\end{bmatrix} = \begin{bmatrix}101\\102\\103\\104\end{bmatrix}$$

We use this type of broadcasting with parameter b.
Broadcasting works with both column and row vectors.

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \underbrace{[100\ 200\ 300]}_{1\times 3} \rightarrow \begin{bmatrix}100\ 200\ 300\\100\ 200\ 300\end{bmatrix} = \begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\end{bmatrix}$$

2×3
(m,n)

$(1,n) \xrightarrow[\text{copies the matrix } m \text{ times}]{\text{python}} (m,n)$

$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ + $\begin{bmatrix} 100 \\ 200 \end{bmatrix}$ $\Rightarrow \begin{bmatrix} 100 & 100 & 100 \\ 200 & 200 & 200 \end{bmatrix} = \begin{bmatrix} 101 & 102 & 103 \\ 204 & 205 & 206 \end{bmatrix}$

(m, n) $(m, 1)$

↓ copies this horizontally 3 times

(m, n)

General principle

(m, n) matrix — if you add, subtract, divide or multiple $+ - * /$ then $(1, n) \rightarrow (m, n)$

performs element-wise

conversely if you have $(m, 1) \rightarrow (m, n)$

- if you have a column vector, and perform these operations against a real num:

$(m, 1)$ $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ + $\mathbb{R}$ 100 $= \begin{bmatrix} 101 \\ 102 \\ 103 \end{bmatrix}$

also works for row vectors

$[1, 2, 3]$ + 100 $= [101 \ 102 \ 103]$

$(1, m)$

Read numpy docs for all broadcasting cases.
Broadcasting in python is somewhat like bsxfun in Matlab / Octave.

A note on Python / Numpy Vectors

a = np.random.randn(5) will give you a rank 1 python array that is neither a row or a column vector.

a.shape = (5,) ← These rank 1 arrays are counter-intuitive and often lead to very hard to find bugs. It is recommended that you <u>don't use them</u> and rather:

a = np.random.randn(5, 1) ← This will give you a 5×1 column vector

a.T transpose will behave as expected

if you're not sure of the dimensions of your vectors you can throw in asserts like

assert(a.shape == (5, 1)) ← They're inexpensive and serve as documentation

Explanation of Logistic Regression Cost Function

the prediction $\hat{y} = \sigma(w^T x + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$

we said that we want to interpret $\hat{y}$ as the probability that $y=1$ given x

$\hat{y} = P(y=1|x)$

so we want our alg to output the chance that $y=1$ for a given set of input features x

If $y=1$: $P(y|x) = \hat{y}$

If $y=0$: $P(y|x) = 1-\hat{y}$ ← chance that y is equal to zero

we want to summarize these two equations

This equation does what we want → $P(y|x) = \hat{y}^{y}(1-\hat{y})^{(1-y)}$

If $y=1$ then the first term $= \hat{y}$, the second term $= 1 = (1-\hat{y})^{0=(1-1)}$

$\therefore p(y|x) = \hat{y}$ ← which is exactly what we want

If $y=0$ then the first term $= 1 = \hat{y}^0$, the second term $(1-\hat{y})^{(1-0)=1}$

$\therefore p(y|x) = 1-\hat{y}$ ← which again is what we want

The above equation is a correct definition for $P(y|x)$

Now, because the log function is a strictly monotonically increasing function, maximizing $\log P(y|x)$ should give you a similar result as optimizing $P(y|x)$ and if you compute $\log P(y|x) =$

$$= \log \hat{y}^{y}(1-\hat{y})^{(1-y)}$$

simplifies to

$$= y \log \hat{y} + (1-y)\log(1-\hat{y})$$

$$= -\mathcal{L}(\hat{y}, y)$$

this is negative because we want to minimize the loss function minimizing the loss corresponds to maximizing the probability

Cost on m examples

identically independently distributed

$$P(\text{labels in training set}) = \prod_{i=1}^{m} P(y^{(i)}|x^{(i)})$$

you want to carry out maximum likelihood estimation. Maximizing the above is the same as maximizing the log of both sides

$$\log P(\cdots) = \sum_{i=1}^{m} \log \underbrace{p(y^{(i)}|x^{(i)})}_{-\mathcal{L}(\hat{y}^{(i)}, y^{(i)})}$$

a principle of stats called max likihood estimation means to choose parameters that maximize

$$= -\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

This justifies the cost we had for logistic regression
By minimizing the cost function J we're really carrying out maximum likelihood estimation with the logistic regression model under the assumption that our training examples were identically independently distributed.

$$\underset{\text{minimize}}{\text{Cost:}} \quad J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

This explains why we use the cost function that we do for logistic regression.