XCS224N Assignment 2

Understanding and Implementing Word2Vec

François Chesnay

Logistics for assignment 2

Important to note for this week:

- The **deadline** for **A2** is **June 22, 11:59 PM PST**
  - **late deadline on June 27, -1 point per late day**
  - If you plan to submit A1 or A2 late you can use a one-time late penalty waiver per course: please fill out this **form**

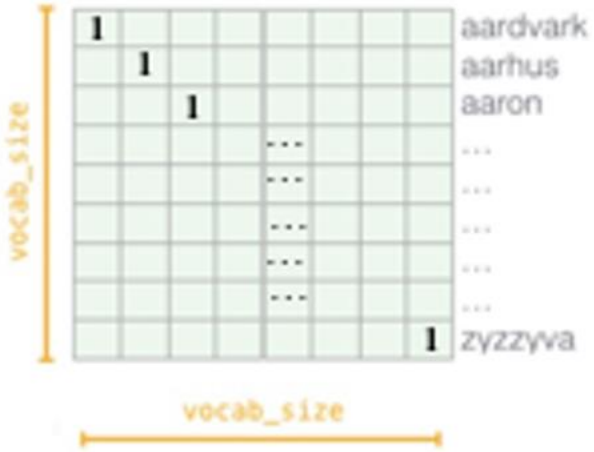
- If you are stuck, post on Slack and ask for help, and/or DM your course facilitator.

Agenda

- Review: main ideas of word2vec
- How to complete the programming part in assignment 2

Main ideas of word2vec: higher level of intelligence

One-hot vector matrix



n-grams matrix

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

vocab_size

vocab_size

Embeddings Matrix

Higher intelligence: capture **syntactic** and **semantic** similarities. Continuous Bag of Words Skip-gram

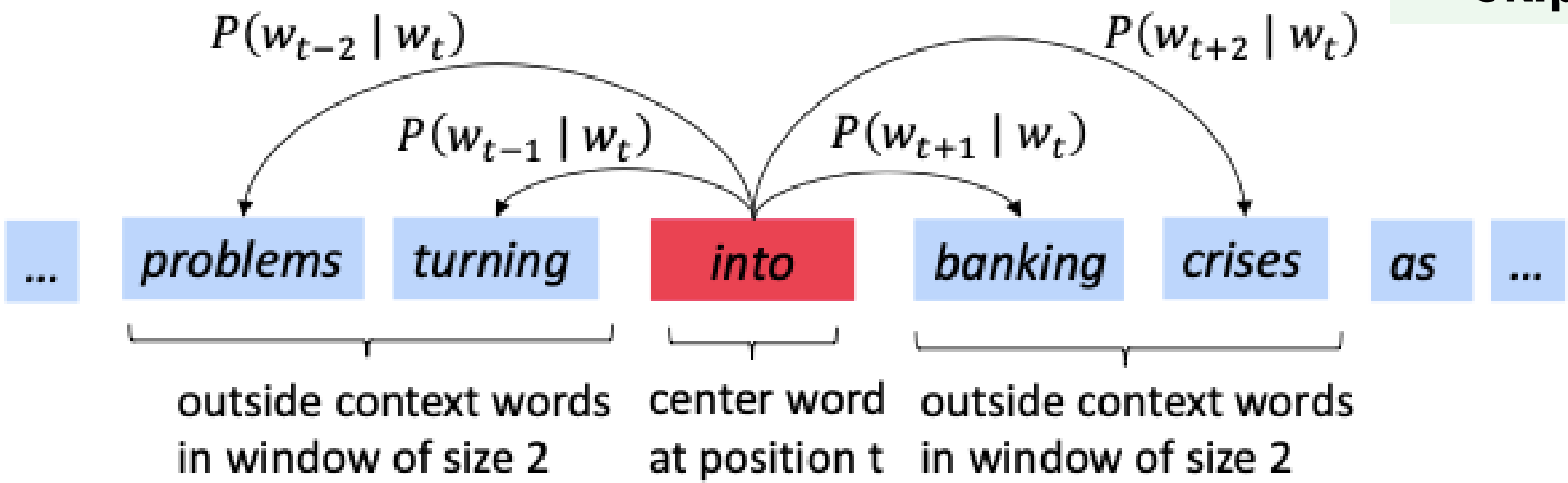
The number of parameters is divided by 1'100.

Assuming vocab_size = 662'109 and embedding_size = 300.

Main ideas of word2vec

- Go through each word of the whole corpus
- Predict surrounding words of each word

Skip-gram



- $$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Stochastic gradients with word vectors

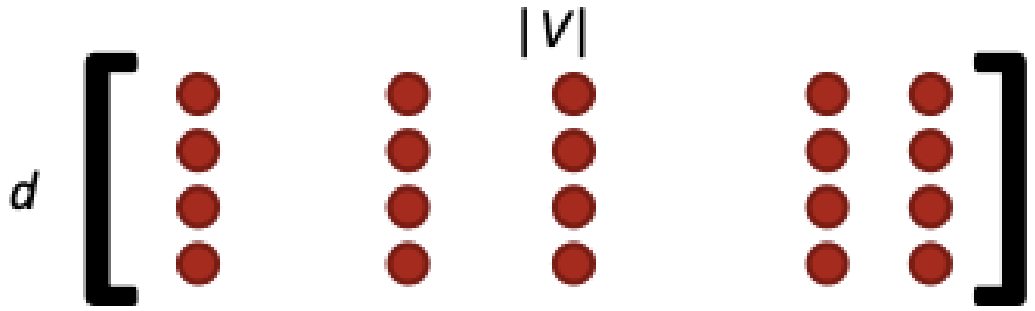
- Then take gradients at each such window for SGD
- But in each window, we only have at most 2m + 1 words, so $\nabla_\theta J_t(\theta)$ is very sparse!

$$\nabla_\theta J_t(\theta) = \begin{bmatrix} 0 \\ \vdots \\ \nabla_{v_{like}} \\ \vdots \\ 0 \\ \nabla_{u_I} \\ \vdots \\ \nabla_{u_{learning}} \\ \vdots \end{bmatrix} \in \mathbb{R}^{2dV}$$

Stochastic gradients with word vectors

- We may as well only update the word vectors that actually appear.

- Solution: either you need sparse matrix update operations to only update certain columns of full embedding matrices U and V



- If you have millions of word vectors and do distributed computing, it is important to not have to send gigantic updates around

Approximations

- The normalization factor is too computationally expensive.

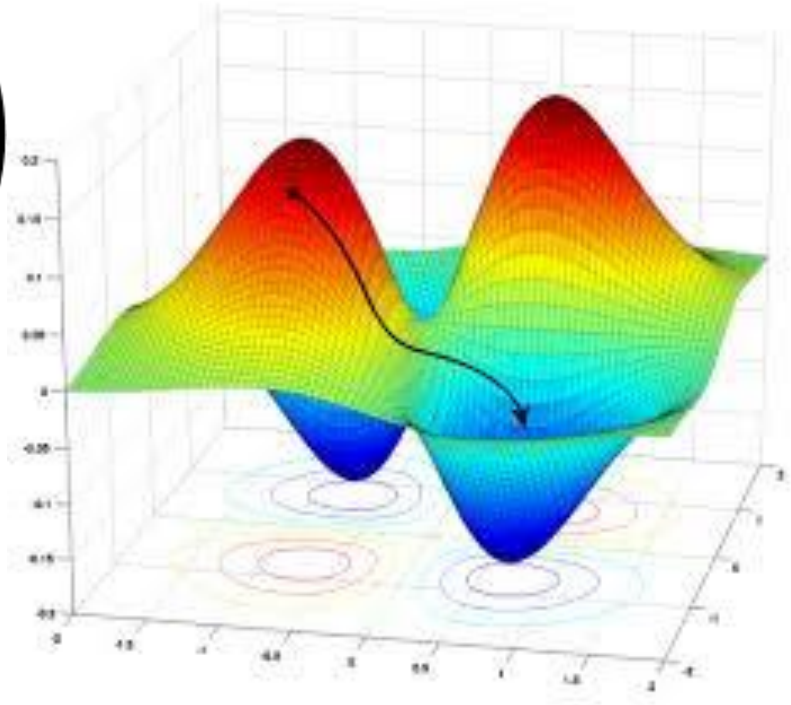
- $$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

- Hence, in this Assignment, you implement the **skip-gram model with negative sampling**

- **Main idea**: **train binary logistic regressions for a true pair (center word and word in its context window) versus a couple of noise pairs (the center word paired with a random word)**

Review: gradient descent – intuition in 2D world

- Have some function $f(\vec{x}) = f\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right)$
- Want $\min_{x_1, x_2} f\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right)$



- Outline
  - Start with some values of x_1, x_2
  - Keep changing x_1, x_2 to reduce the value of $f\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right)$ until a minimum is reached.

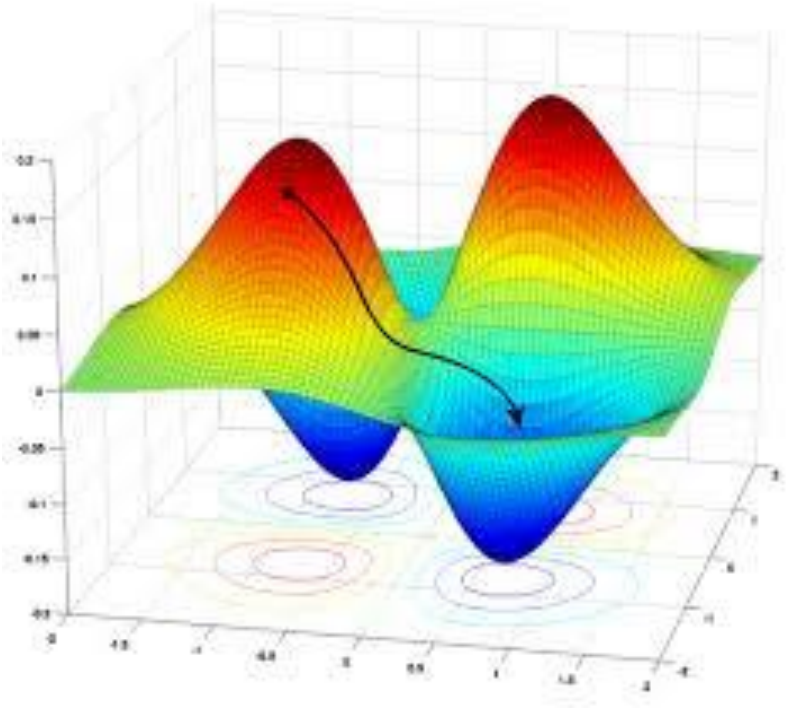
Review: gradient descent algorithm in 2D world

- Repeat until convergence {

$$x_1 := x_1 - \alpha \frac{\partial}{\partial_{x_1}} f\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right)$$

$$x_2 := x_2 - \alpha \frac{\partial}{\partial_{x_2}} f\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right)$$

  }
  where α is the learning rate (named **step** in in assignment 2)



- Intuition: find the direction leading to the lowest value
  - movement is direction * step (direction is the negative gradient).

Gradient descent implementation in 2D world

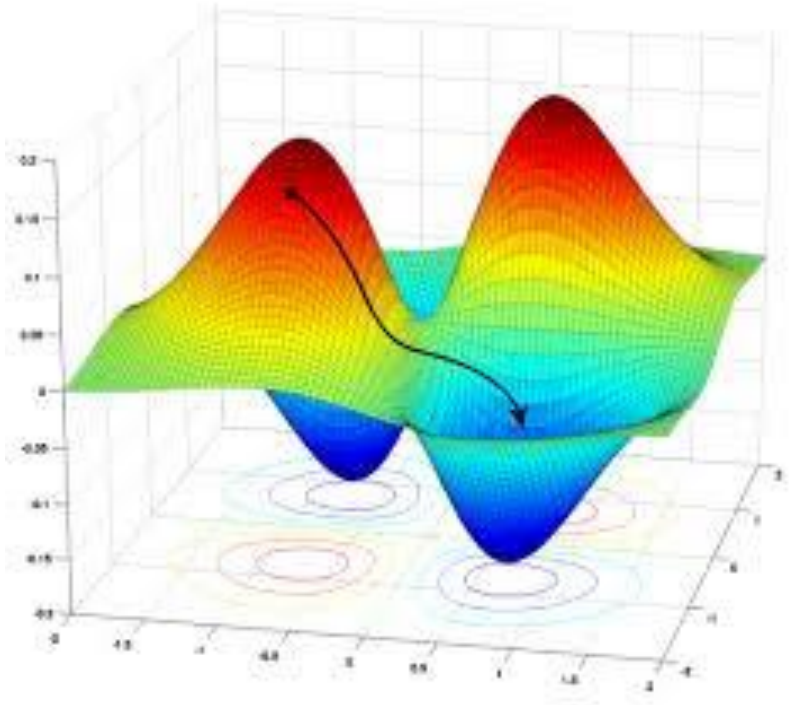
- Beware, when dealing with multiple input parameters, need simultaneous update:

temp_x1 $:= x_1 - \alpha \frac{\partial}{\partial_{x_1}} f\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right)$

temp_x2 $:= x_2 - \alpha \frac{\partial}{\partial_{x_2}} f\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right)$

$x_1 :=$ temp_x1

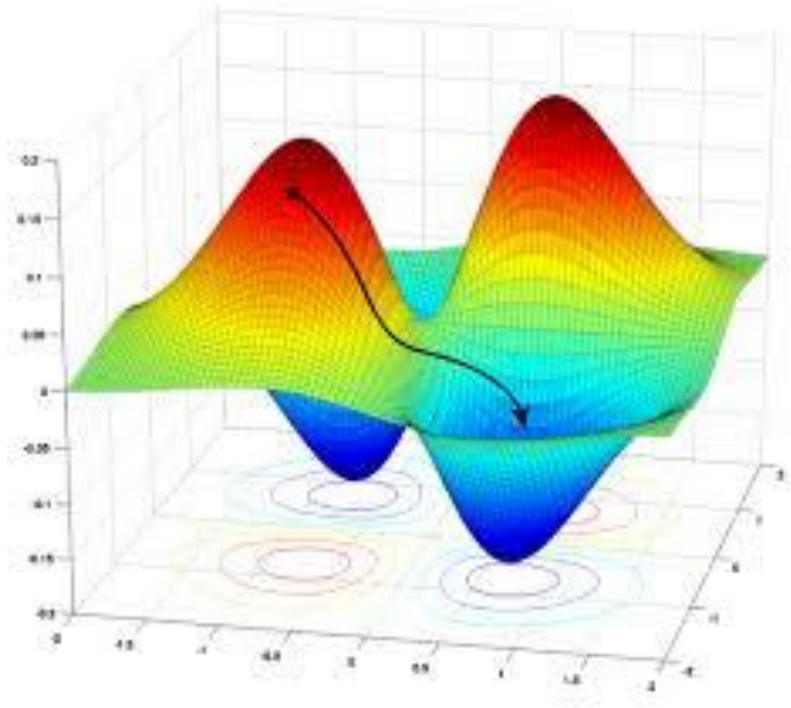
$x_2 :=$ temp_x2

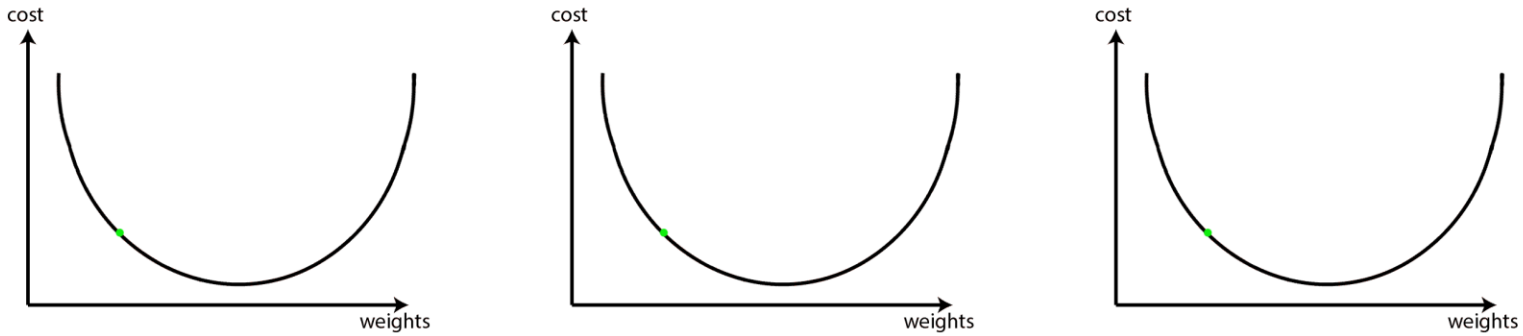


Do not worry about this part for assignement 2. This will be useful for later assignements to avoid bugs.

Gradient descent implementation: choosing α

- The learning rate **step** in assignment 2, or α in our formula indicates how much we are moving in the direction of the path of greatest descent.

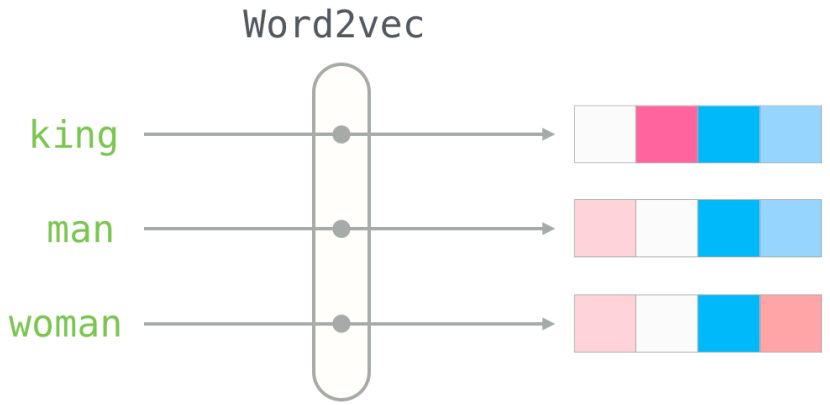




- See video from Andrew Ng: https://www.youtube.com/watch?v=rIVLE3condE

Word2Vec Coding Tutorial

HOW TO CODE ASSIGNMENT 2



Source of Word2Vec's pictures: http://jalammar.github.io/illustrated-word2vec/

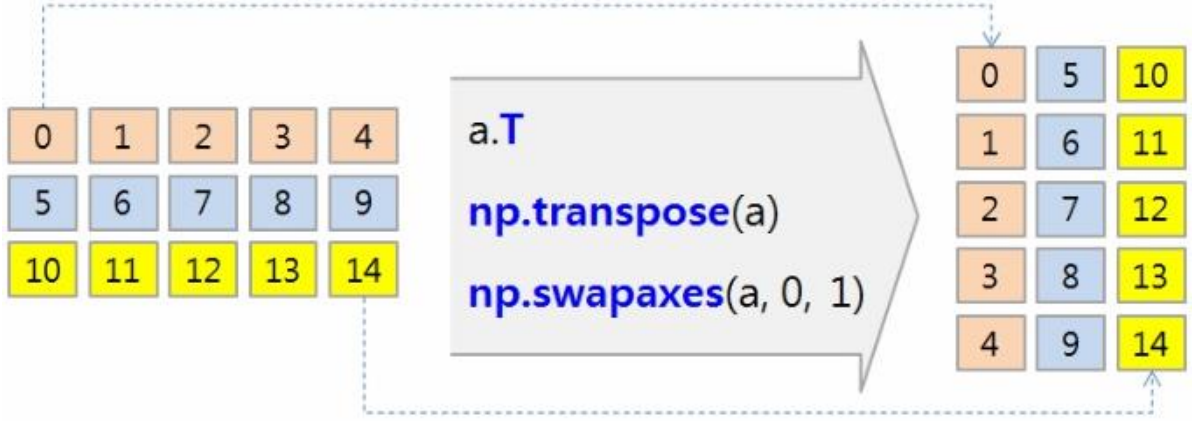
Coding: Implementing word2vec

- **Sigmoid**: use np.exp()rather than math.exp() → as **input is a vector and not a scalar**

- **Selecting an element**:
  - in a 1D Numpy array called vector at position index: use vector[index]

- **Create a matrix or array with zeros:**
  - np.zeros((m,n)) ← **note the double quotes for that 2D array.**
  - np.zeros(d)
  - Same dimension: y = np.zeros_like(y_hat)

- **Transposing a matrix:** for example, transpose a matrix **a** of dimension (3, 5) to (5, 3)

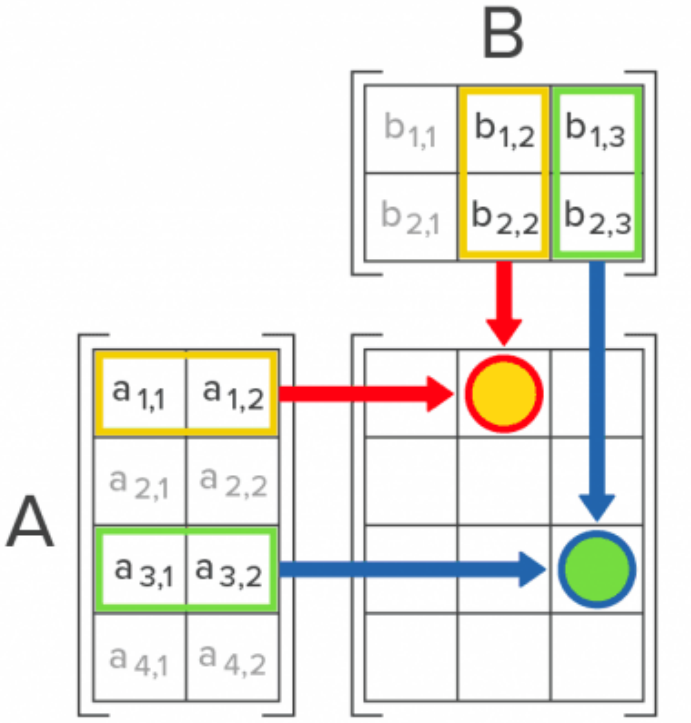


Coding: Implementing word2vec

- **Matrix multiplications**:
  - Check the size the **dimensions** of your matrices and vectors.
  - Understand whether data are in **rows** or **columns**.
  - Write in your **comments** the dimensions of your matrices.

 # (4,3) (4,2)(2,3) (5,5) (5,1)(1,5)

- **X = np.dot(A, B)** **X = np.outer(A, B)**



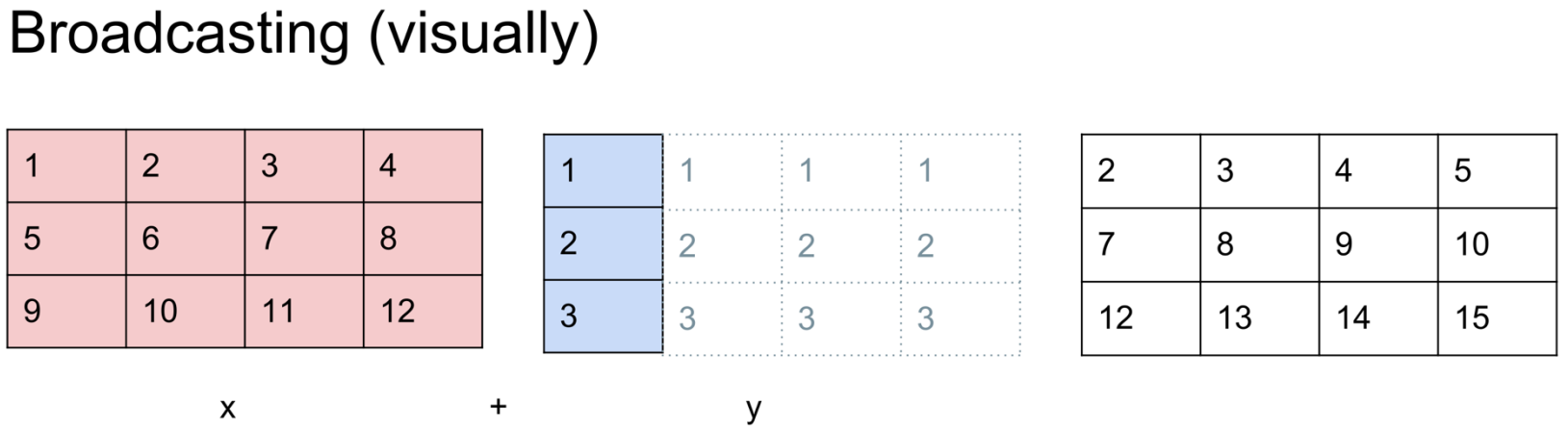
$$a \otimes b = ab^T = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix}_{(n \times 1)} \begin{bmatrix} b_1 & b_2 & b_3 & b_4 & b_5 \end{bmatrix}_{(1 \times n)} = \begin{Bmatrix} a_1b_1 & a_1b_2 & a_1b_3 & a_1b_4 & a_1b_5 \\ a_2b_1 & a_2b_2 & a_2b_3 & a_2b_4 & a_2b_5 \\ a_3b_1 & a_3b_2 & a_3b_3 & a_3b_4 & a_3b_5 \\ a_4b_1 & a_4b_2 & a_4b_3 & a_4b_4 & a_4b_5 \\ a_5b_1 & a_5b_2 & a_5b_3 & a_5b_4 & a_5b_5 \end{Bmatrix}_{(n \times n)}$$

Outer Product

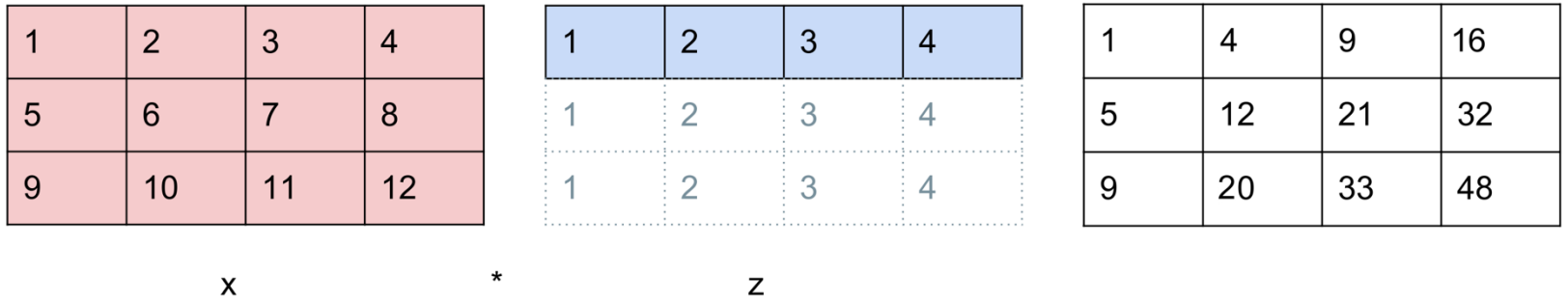
Coding: Implementing word2vec - Broadcasting

- **Matrix multiplication with Broadcasting** :

Rule 1: If the two arrays differ in their number of dimensions, the shape of the array with fewer dimensions is padded with ones on its leading (left) side.



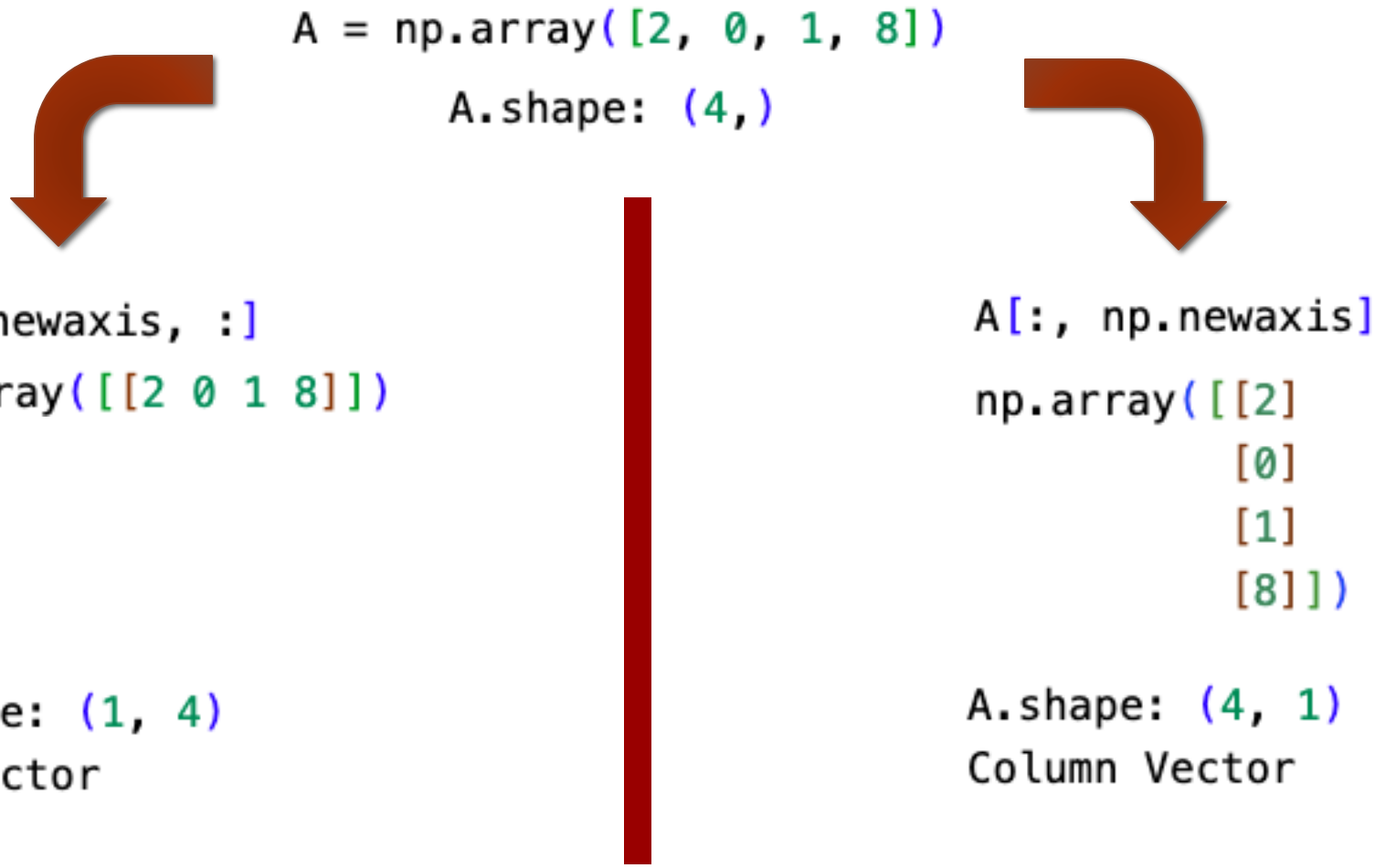
Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.



Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

Coding: Implementing word2vec

- **Transform 1D array (vector) into a 2D matrix: increase the dimension of an existing array**



2 Coding: Implementing word2vec

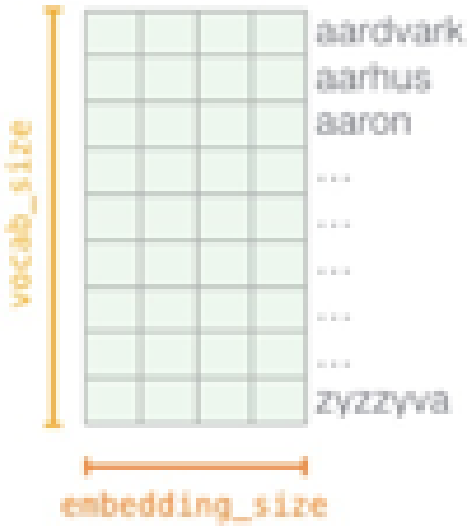
In `word2vec`, the conditional probability distribution is given by taking vector dot-products and applying the softmax function:

$$P(O = o|C = c) = \frac{\exp(\boldsymbol{u_o}^\top \boldsymbol{v_c})}{\sum_{w \in \text{Vocab}} \exp(\boldsymbol{u_w}^\top \boldsymbol{v_c})} \tag{1}$$

Here, $\boldsymbol{u}_o$ is the 'outside' vector representing outside word o, and $\boldsymbol{v}_c$ is the 'center' vector representing center word c. To contain these parameters, we have two matrices, $\boldsymbol{U}$ and $\boldsymbol{V}$. The columns of $\boldsymbol{U}$ are all the 'outside' vectors $\boldsymbol{u}_w$. The columns of $\boldsymbol{V}$ are all of the 'center' vectors $\boldsymbol{v}_w$. Both $\boldsymbol{U}$ and $\boldsymbol{V}$ contain a vector for every $w \in$ Vocabulary.[1]

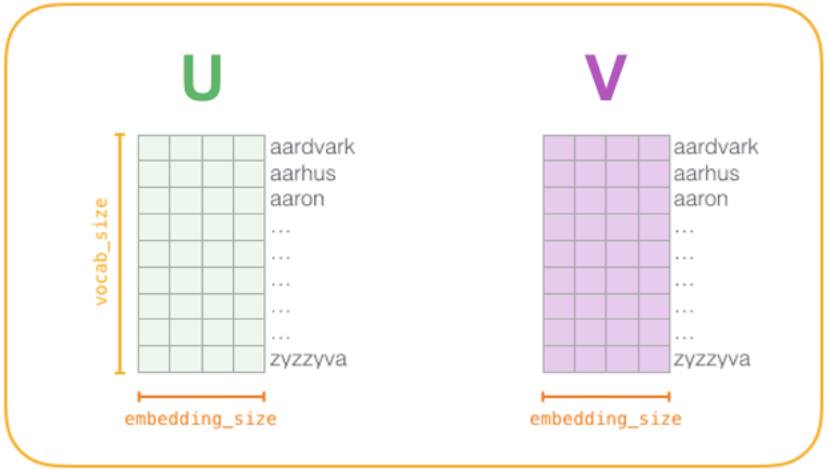
$\hat{\boldsymbol{y}} =$

$\boldsymbol{y} =$



y = np.zeros_like(y_hat)

Coding: Implementing word2vec

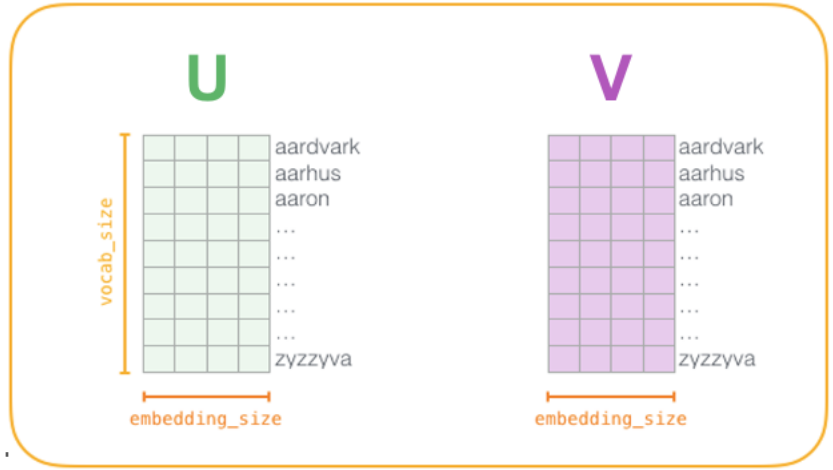


- **Matrix multiplications**: check the **dimensions** and whether data are in **rows** or **columns**

- **We usually use column vector convention (i.e., vectors are in column form) for vectors in matrix U and V (in the handout), but for implementation/programming we usually use row vectors (representing vectors in row form).**

- **For python run.py, on a Mac M4, it takes 15 m, while it can take 3 hours on a windows machine→ Use the softmax() function provided (utils/utils.py)**

Coding: Implementing word2vec



- **Matrix multiplications**: check the **dimensions** and whether data are in **rows** or **columns**
  - **(2) (5) naive-softmax loss** given in Equation(2) is the same as the cross-entropy loss betweer $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ in Equation (5):

$$\boldsymbol{J}_{\text{naive-softmax}}(\boldsymbol{v}_c, o, \boldsymbol{U}) = -\log P(O = o|C = c) = -\sum y_w \log(\hat{y}_w) = -\log(\hat{y}_o)$$

Hint: you are given outside_word_idx, creating y_hat and y isn't too difficult.

  - **(3) partial derivative of J with respect to the center word vector**

$$\frac{\partial J}{\partial \boldsymbol{v}_c} = \boldsymbol{U}(\hat{\boldsymbol{y}} - \boldsymbol{y}) \qquad (3)$$

(Vocab,)

(Vocab, nb_of_embeddings)

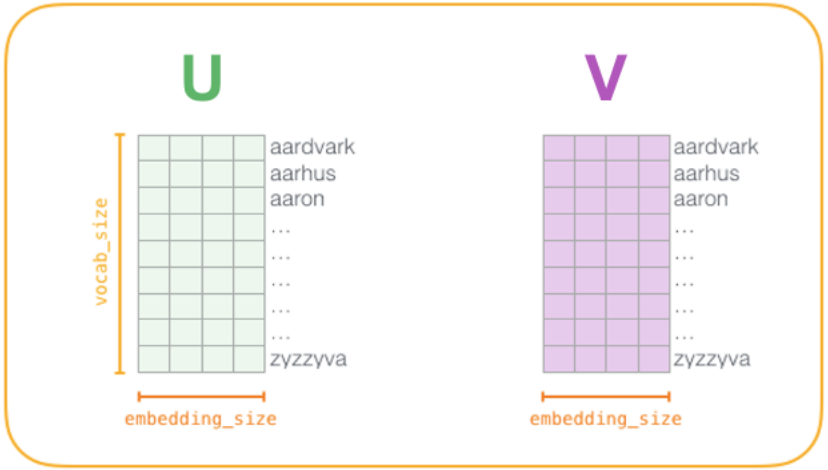
(nb_of_embeddings,)

How to resolve the dimension's issue with **Transposition**:

U.T **(y_hat – y)**

(nb_of_embeddings, Vocab) **(Vocab,)**

Coding: Implementing word2vec



- **(6) partial derivative of J with respect to each of the outside word vectors**

$$\frac{\partial J}{\partial U} = \boldsymbol{v}_c(\hat{\boldsymbol{y}} - \boldsymbol{y})^\top \qquad (6)$$

(Vocab, number of embeddings) **(number embeddings,)** **(Vocab,)**

How to resolve the dimension issue?

- Use np.outer on **(Vocab,)(number of embeddings,)**
- Use np.dot with reshaped vectors using **[:, np.newaxis]** and **[np.newaxis, :]**

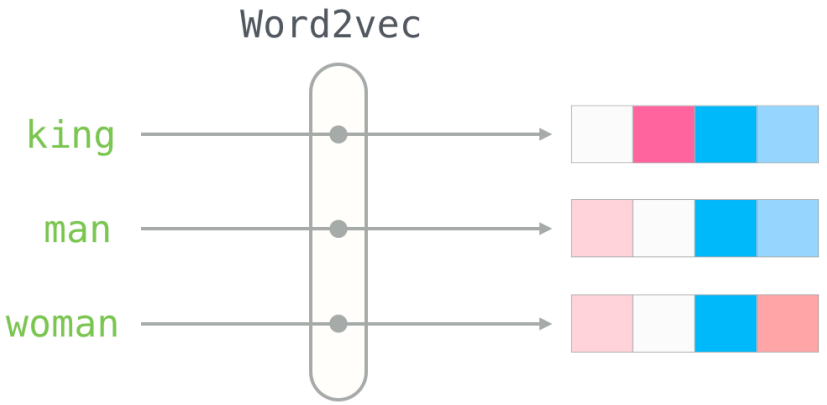
Debugging word2vec

```
def skipgram(current_center_word, wind .. word2vec_loss_and_gradient=neg_sampling_loss_and_gradient):
    """ Skip-gram model in word2vec
```

- Are you calling the wrong function in skipgram?
- Difference in updating between grad_outside_vectors and grad_center_vecs
- Accumulating the gradients for centers words and outside words += and not making an assignment.

Questions?

PLEASE POST ON SLACK OR CONTACT YOUR COURSE FACILITATOR.



What are the differences between np.dot and np.matmul?

Functionality:

np.dot: Primarily designed for calculating the dot product of two arrays. It can handle vectors, matrices, and higher-dimensional arrays.

np.matmul: Specifically designed for matrix multiplication. It's generally more efficient for this task compared to np.dot. However, it has stricter requirements for input shapes.

Key Differences:

1. Scalar multiplication:

np.dot allows multiplication of an array by a scalar value.

np.matmul does not support scalar multiplication directly.

2. Broadcasting:

np.dot follows NumPy's broadcasting rules for element-wise operations when dimensions don't match exactly.

np.matmul performs matrix multiplication, which has specific rules for broadcasting dimensions.

3. Performance:

np.dot might be slightly less efficient for large matrices due to its broader functionality.

np.matmul is generally optimized for matrix multiplication and often faster for that specific task.

Choosing the Right Function:

Use **np.dot** (i) for calculating the dot product of vectors or matrices or when you need scalar multiplication or NumPy's broadcasting flexibility.

Use **np.matmul** (i) for performing matrix multiplication efficiently. Or (ii) when you know the input arrays have compatible shapes for matrix multiplication.

What is an appropriate embedding size (pre-trained word vectors' dimension)?

- Word2Vec (2013): standard dimension 300, but Word2Vec guidance suggests between 100 and 1000.
- Glove (2014): 25, 50, 100, 200 or 300.
- BERT (2018): BERT model `BERT-Base` generates 768-length embedding vector compared to the smaller BERT model which generates 128 length embedding vector.
- Elmo (2018): 1'024.
- GPT-2 (2019): 1'600
- GPT-3 (2020): 12'288 (not using words, but employs a subword-based representation where words are broken down into smaller units like morphemes or prefixes)
- LLaMA 2 (2023): 3,204

How can I practice Python and Numpy?

Please find links to excellent tutorials and exercises:

Tutorials:

- https://cs231n.github.io/python-numpy-tutorial/
- https://cs229.stanford.edu/section/cs229_python_tutorial/cs229_python_friday.pdf
- https://numpy.org/doc/stable/user/basics.broadcasting.html

Practice:

- https://github.com/norvig/pytudes
- https://github.com/Asabeneh/30-Days-Of-Python

Hi could someone explain why the jacobian form of this derivative is a row vector? (slide 45 of module 2)

Similarly, $\frac{\partial s}{\partial \boldsymbol{b}} = \boldsymbol{h}^T \circ f'(\boldsymbol{z})$ is a row vector

- But shape convention says our gradient should be a column vector because ***b*** is a column vector …

Input x:

- Let x be an input vector of dimension $n \times 1$.

Weight Matrix W:

- Let W be a weight matrix of dimension $m \times n$, where m is the number of hidden units.

Bias b:

- Let b be a bias vector of dimension $m \times 1$.

Linear Transformation z:

- $z = Wx + b$
- The dimension of z is $m \times 1$.

Activation Function $h = f(z)$:

- h is the output of the activation function applied element-wise to z.
- The dimension of h remains $m \times 1$.

Output $s = u^T h$:

- Let u be a weight vector of dimension $m \times 1$.
- s is a scalar (dimension 1×1).

1. **Gradient of s with respect to h**:
   - $\frac{\partial s}{\partial h} = u^T$, where u^T is a row vector of dimension $1 \times m$.
2. **Gradient of h with respect to z**:
   - $\frac{\partial h}{\partial z}$ is a diagonal matrix of dimension $m \times m$, representing the derivative of the activation function $f'(z)$.
3. **Gradient of z with respect to b**:
   - $\frac{\partial z}{\partial b} = I$, where I is the identity matrix of dimension $m \times m$.

When you multiply these together:

$$\frac{\partial s}{\partial b} = u^T \cdot \text{diag}(f'(z)) \cdot I$$

- u^T is $1 \times m$.
- $\text{diag}(f'(z))$ is $m \times m$.
- I is $m \times m$.

The result of the multiplication $u^T \cdot \text{diag}(f'(z))$ is a row vector of dimension $1 \times m$, and multiplying by the identity matrix I does not change its dimensions.

Thus, $\frac{\partial s}{\partial b}$ is indeed a **row vector** of dimension $1 \times m$. This is consistent with the fact that the gradient of a scalar (in this case, s) with respect to a vector (in this case, b) is a row vector.

Understanding h and z:

z is a vector of dimension $m \times 1$ (i.e., $z = [z_1, z_2, \ldots, z_m]^T$).

h is the output of the activation function applied element-wise to z, so $h = f(z) = [f(z_1), f(z_2), \ldots, f(z_m)]^T$.

Thus, h is also a vector of dimension $m \times 1$.

Gradient $\frac{\partial h}{\partial z}$:

The gradient $\frac{\partial h}{\partial z}$ represents how each element of h changes with respect to each element of z.

Since h is a function of z applied element-wise, each $h_i = f(z_i)$ depends only on z_i, not on z_j for $j \neq i$.

Structure of $\frac{\partial h}{\partial z}$:

The gradient $\frac{\partial h}{\partial z}$ is a matrix where each entry (i, j) is the partial derivative $\frac{\partial h_i}{\partial z_j}$.

Because h_i depends only on z_i, $\frac{\partial h_i}{\partial z_j} = 0$ for $i \neq j$.

For $i = j$, $\frac{\partial h_i}{\partial z_i} = f'(z_i)$, where f' is the derivative of the activation function.

Diagonal Matrix:

This means that $\frac{\partial h}{\partial z}$ is a diagonal matrix of the form:

$$\frac{\partial h}{\partial z} = \begin{pmatrix} f'(z_1) & 0 & \cdots & 0 \\ 0 & f'(z_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'(z_m) \end{pmatrix}$$

The off-diagonal elements are zero because h_i does not depend on z_j for $i \neq j$.

The diagonal elements are $f'(z_i)$, the derivative of the activation function evaluated at each z_i.

Dimension:

Since z and h are both $m \times 1$ vectors, $\frac{\partial h}{\partial z}$ is an $m \times m$ matrix.

It is a square matrix with m rows and m columns.

Get familiar with Linux command lines

- Determine your current directory with **pwd**
- List files in your directory with **ls**
- Change directory with **cd**
- Look at https://web.stanford.edu/class/archive/cs/cs107/cs107.1216/resources/unix.html for more info