CS/DS 541: Class 21

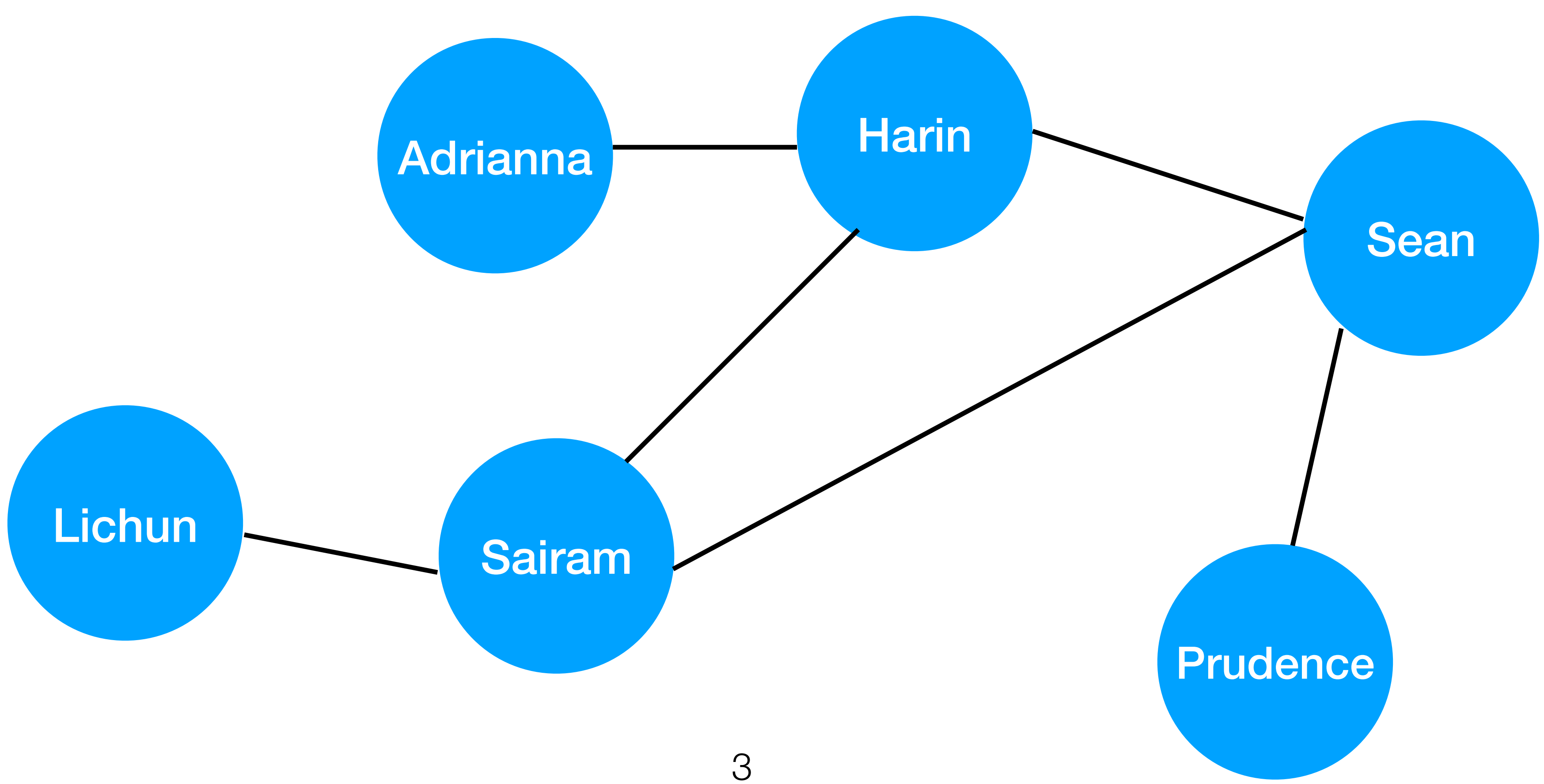
Jacob Whitehill

Machine learning on graphs

Machine learning on graphs

- Consider an undirected graph where each node is associated with a feature vector.

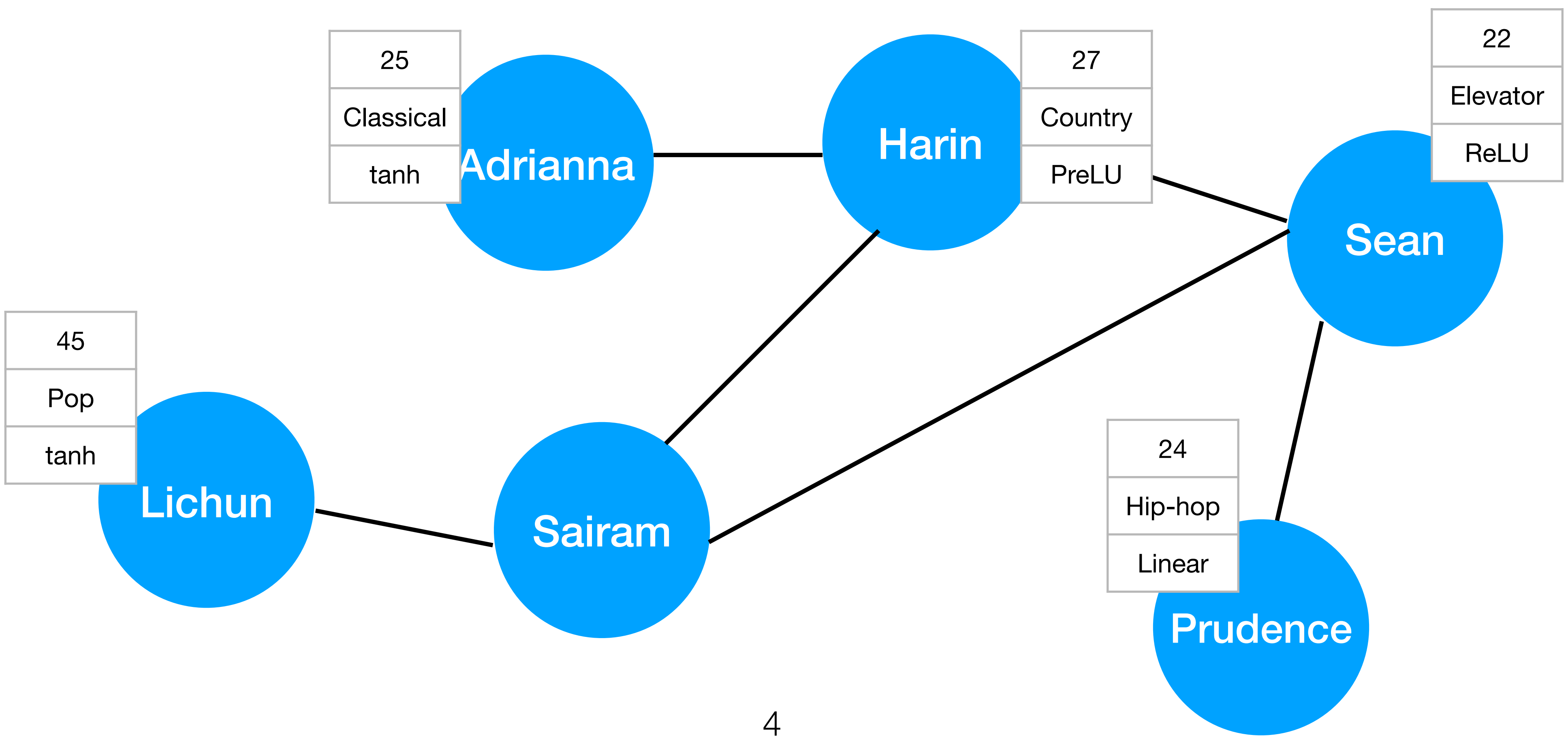
- Example: social network of CS/DS 541.



Machine learning on graphs

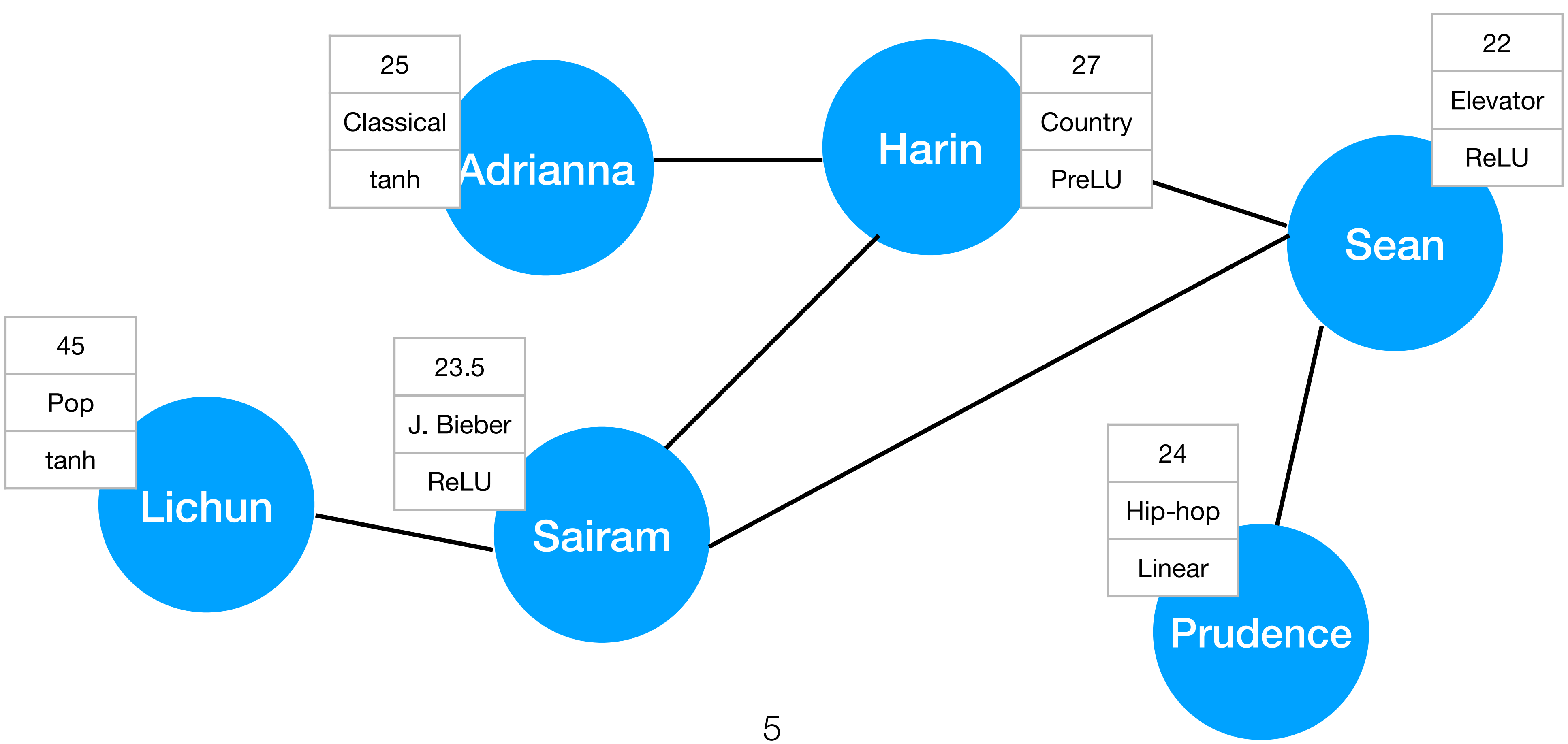
- Consider an undirected graph where each node is associated with a feature vector.

- Example: social network of CS/DS 541.



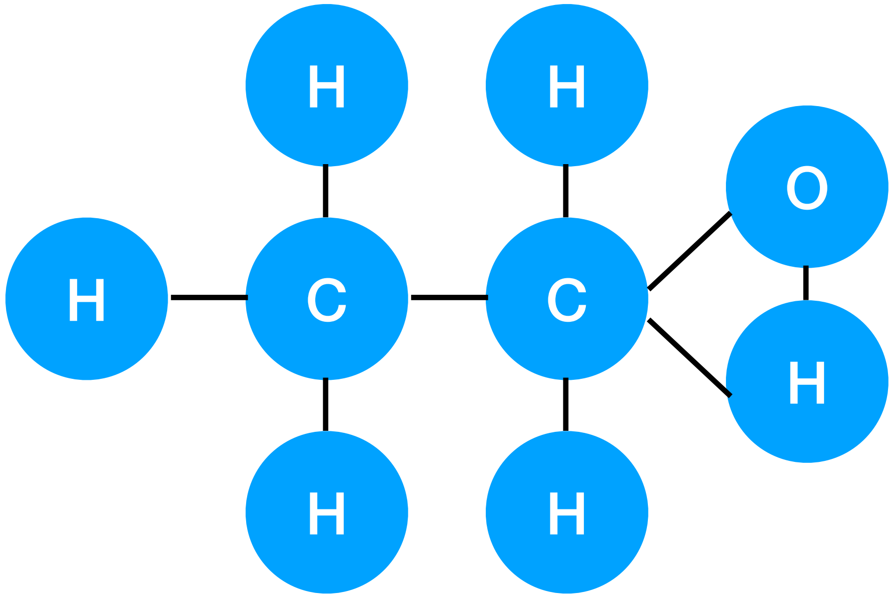
Machine learning on graphs

- Based on the graph topology and set of observed feature vectors, can we infer the unobserved feature vector?
  - This is a semi-supervised ML problem.



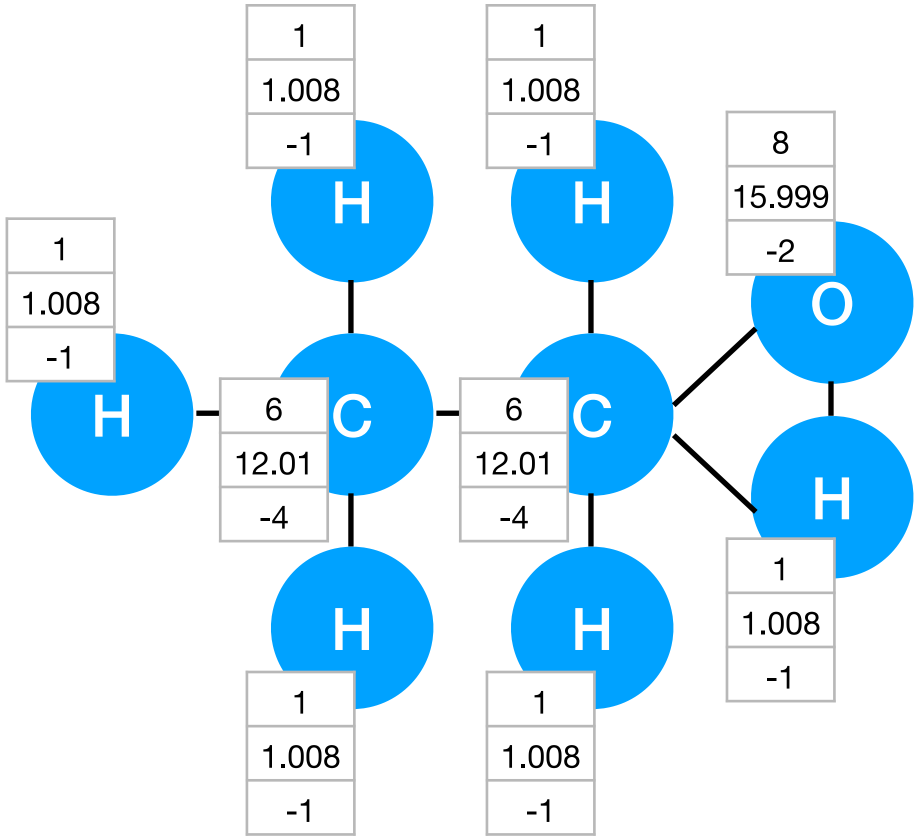
Machine learning on graphs

- Example: computational chemistry.



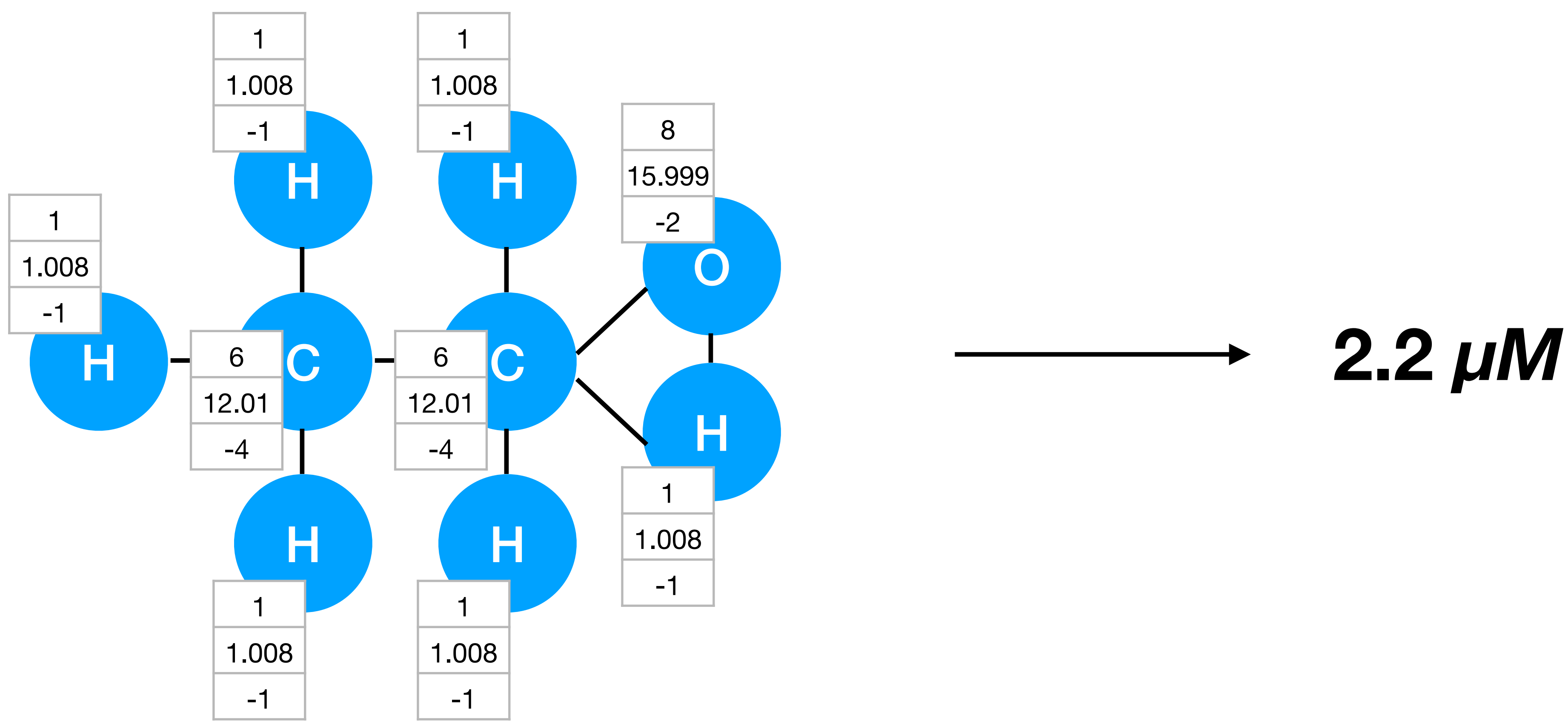
Machine learning on graphs

- Example: computational chemistry.



Machine learning on graphs

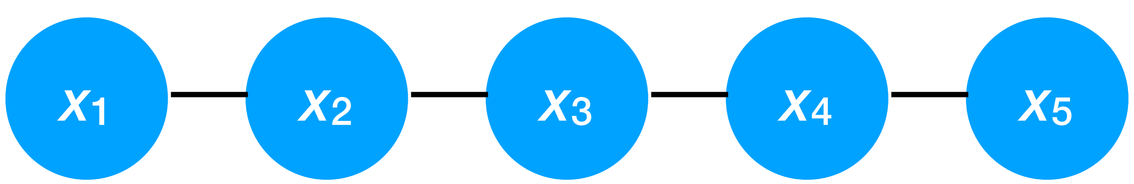
- Can we predict the binding affinity of a molecule to a particular receptor given its molecular structure?



Machine learning on graphs

- In contrast to vectors and images, the topology of graphs is **variable**.

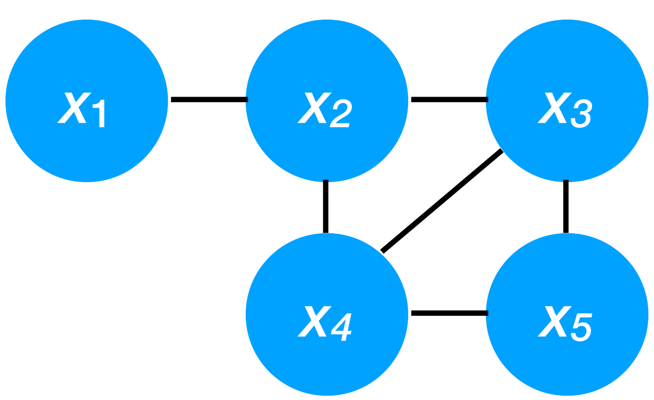
- E.g., while each element of an *n*-vector is connected to 1-2 neighbors



Machine learning on graphs

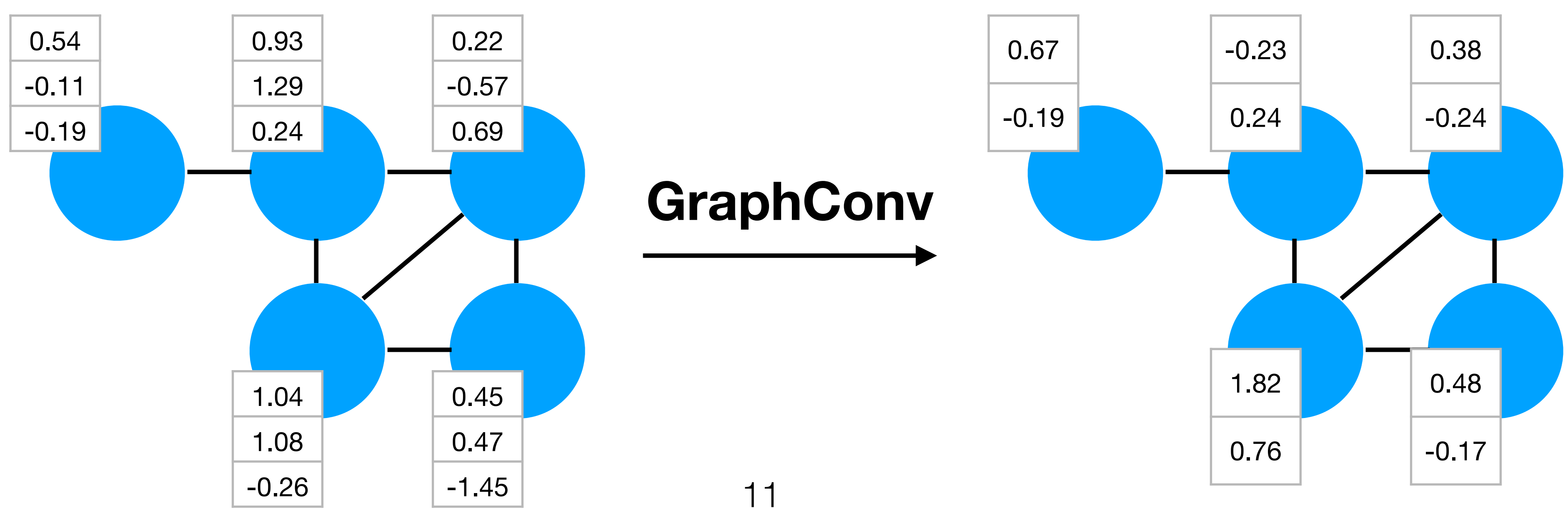
- In contrast to vectors and images, the topology of graphs is **variable**.

- E.g., while each element of an *n*-vector is connected to 1-2 neighbors, each graph node has up to *n*-1 neighbors.



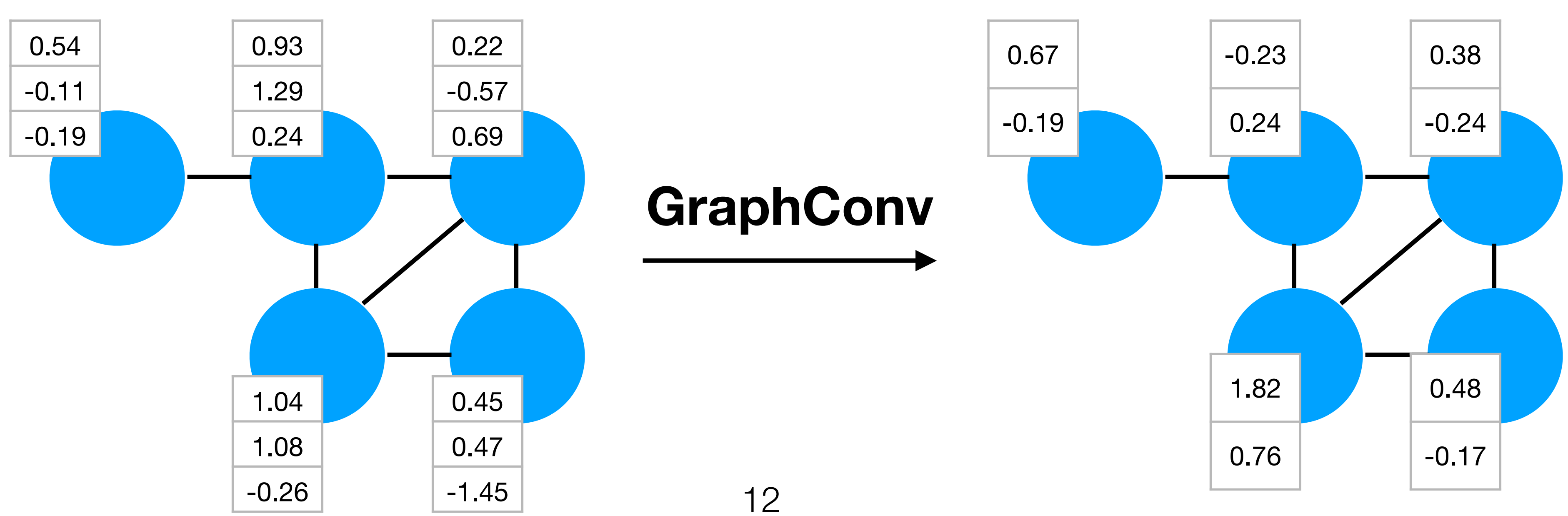
Machine learning on graphs

- With GCNs, the feature vector at each node is transformed (by each GraphConv layer) based on the feature vectors of nearby nodes.



Machine learning on graphs

- With GCNs, the feature vector at each node is transformed (by each GraphConv layer) based on the feature vectors of nearby nodes.
- With *true* graph convolution, “nearby” actually means *all* nodes.
- However, most GCNs actually perform *approximate* convolution where the set of influencing nodes is local.

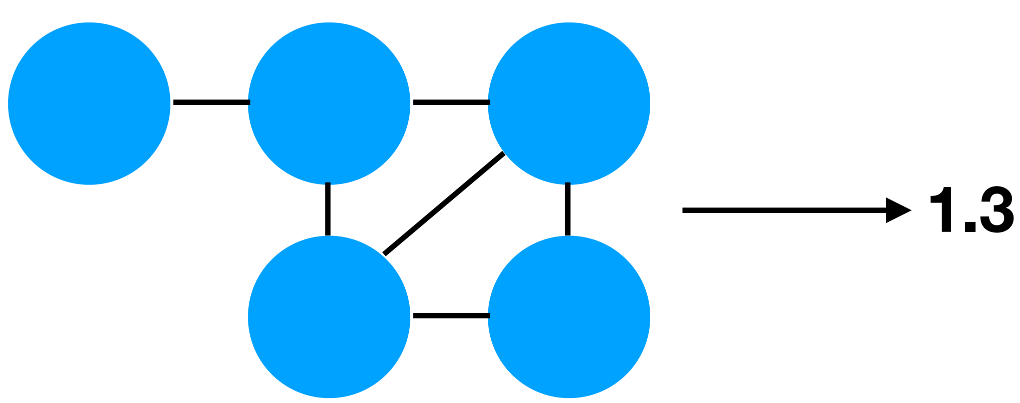


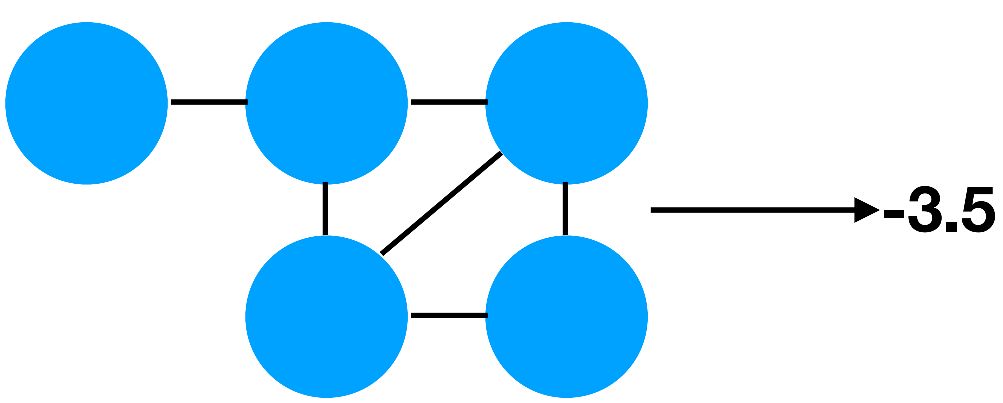
Machine learning on graphs

- When doing ML on graphs, we usually either:

  1. Restrict the topology of all input graphs to be the same (but with different feature vectors):

  - Supervised: apply a tailored aggregation function that depends on a specific graph topology.
  - Semi-supervised: classify the unlabeled nodes.



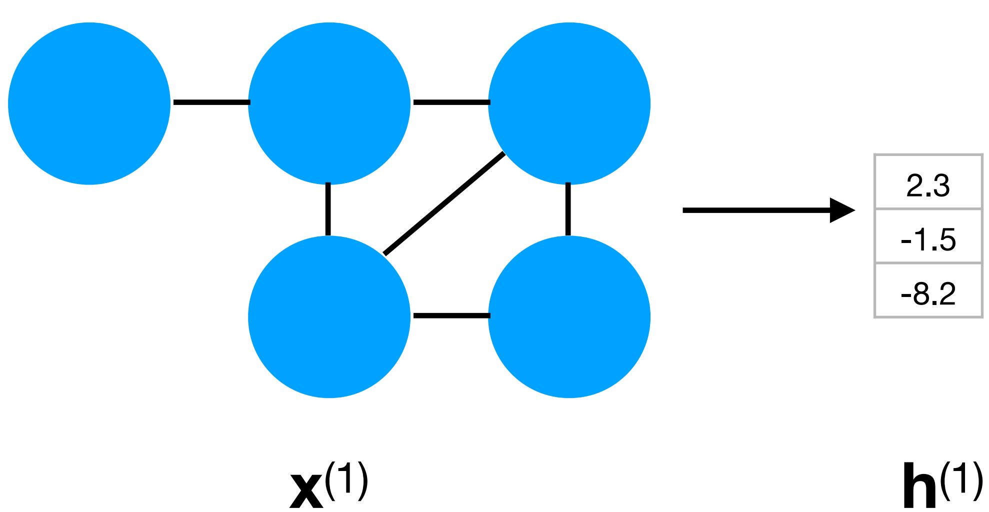


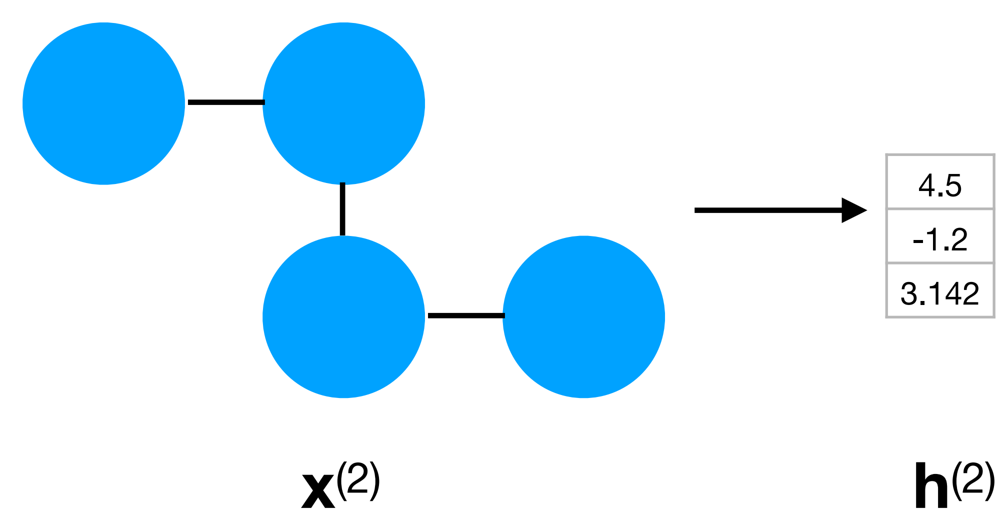
$\mathbf{x}^{(1)}$ $\hat{y}^{(1)}$ $\mathbf{x}^{(2)}$ $\hat{y}^{(2)}$

Machine learning on graphs

- …or:

  2. Apply pooling to convert the graph into a fixed-length feature vector for downstream processing.





GCN: theory

Convolution

- What are two nice properties of convolution?

Convolution

- What are two nice properties of convolution?

  1. Apply the same weights to everywhere in the input regardless of location:

     - Weight-sharing

     - Locality

  2. Convolution theorem.

Convolution theorem

- According to the **convolution theorem**:

$$\mathbf{x} * \mathbf{w} = \mathcal{F}^{-1}(\mathcal{F}(\mathbf{x}) \odot \mathcal{F}(\mathbf{w}))$$

where * represents convolution, and $\mathcal{F}$ is the Fourier transform.

Convolution theorem

- According to the **convolution theorem**:

$$\mathbf{x} * \mathbf{w} = \mathcal{F}^{-1}(\mathcal{F}(\mathbf{x}) \odot \mathcal{F}(\mathbf{w}))$$

where * represents convolution, and $\mathcal{F}$ is the Fourier transform.

- In other words, convolution can be implemented by:
  1. Applying a Fourier transform to **x** and **w**;
  2. Multiplying the transformed vectors element-wise; and
  3. Applying an inverse Fourier transform to the result.

Convolution theorem

- Demo.

Fourier transform of functions

- The Fourier transform of a vector **x** can be computed by multiplying by a **Fourier basis matrix V** (though the FFT is faster):

$$\mathcal{F}(\mathbf{x}) = \mathbf{V}^\top \mathbf{x}$$

Fourier transform of functions

- Where does the basis **V** come from?

$$\mathbf{v}_k = \frac{1}{\sqrt{n}} \begin{bmatrix} e^{j2\pi 0k/n} \\ \vdots \\ e^{j2\pi(n-1)k/n} \end{bmatrix}$$

- The Fourier bases are eigenfunctions of the Laplacian operator for real functions.

Fourier transform of graphs

- Let graph function *x*(*i*) be a function from graph nodes to scalar values; equivalently, we can let $\mathbf{x} \in \mathbb{R}^m$.

- For graphs, the Fourier basis matrix **V** consists of the eigenvectors of the **normalized graph Laplacian matrix L**:

$$\mathbf{L} = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$$

$$\tilde{\mathbf{x}} = \mathcal{F}(\mathbf{x}) = \mathbf{V}^\top \mathbf{x}$$

$$\mathbf{x} = \mathcal{F}^{-1}(\tilde{\mathbf{x}}) = \mathbf{V}\tilde{\mathbf{x}}$$

A: adjacency matrix
D: degree matrix

Graph convolution

- With this machinery, we can now implement convolution of a graph function **x** with a filter **w**:

$$\mathcal{F}^{-1}(\mathcal{F}(\mathbf{w}) \odot \mathcal{F}(\mathbf{x}))$$

Graph convolution

- With this machinery, we can now implement convolution of a graph function **x** with a filter **w**:

$$\mathcal{F}^{-1}(\mathcal{F}(\mathbf{w}) \odot \mathcal{F}(\mathbf{x})) = \mathbf{V}(\mathbf{V}^\top \mathbf{w} \odot \mathbf{V}^\top \mathbf{x})$$

Graph convolution

- With this machinery, we can now implement convolution of a graph function **x** with a filter **w**:

$$
\begin{aligned}
\mathcal{F}^{-1}(\mathcal{F}(\mathbf{w}) \odot \mathcal{F}(\mathbf{x})) &= \mathbf{V}(\mathbf{V}^\top \mathbf{w} \odot \mathbf{V}^\top \mathbf{x}) \\
&= \mathbf{V}\theta\mathbf{V}^\top \mathbf{x}
\end{aligned}
$$

where we define diagonal matrix θ to be equivalent to multiplying $V^\top x$ element-wise by $V^\top w$.

Approximation

- Here, θ is an n-dimensional vector of parameters; this performs a true graph convolution.
- However, this also implies that the convolution output, for each node in the graph, depends on the entire graph -- not just a local subgraph.
- It is also slow, since we must multiply by potentially large (n x n) matrices.
- In Kipf & Welling (2017), they instead make an approximation…

Approximation

- Here, θ is an n-dimensional vector of parameters; this performs a true graph convolution.

- However, this also implies that the convolution output, for each node in the graph, depends on the entire graph -- not just a local subgraph.

- It is also slow, since we must multiply by potentially large (n x n) matrices.

- In Kipf & Welling (2017), they instead make an approximation (and wrap it with σ) that yields the result:

$$\mathbf{h} = \sigma\left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{X}\mathbf{W}\right)$$

Approximation

- **X**: *n* x *k* feature matrix (for a graph with *n* nodes, and k feature channels).
- **W**: *k* x *m* filter matrix
- **A**: graph adjacency matrix
- **D**: graph degree matrix

Linear transformation from *k* features to *m* features

$$\mathbf{h} = \sigma\left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\boxed{\mathbf{X}\mathbf{W}}\right)$$

Approximation

- **X**: *n* x *k* feature matrix (for a graph with *n* nodes, and k feature channels).
- **W**: *k* x *m* filter matrix
- **A**: graph adjacency matrix
- **D**: graph degree matrix

Weighted sum over the features of neighboring nodes

$$\mathbf{h} = \sigma\left(\boxed{\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}}\mathbf{X}\mathbf{W}\right)$$

Approximation

- **X**: *n* x *k* feature matrix (for a graph with *n* nodes, and k feature channels).
- **W**: *k* x *m* filter matrix
- **A**: graph adjacency matrix
- **D**: graph degree matrix

$$\mathbf{h} = \sigma\left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{X}\mathbf{W}\right)$$

Relationship to FC layer

- Special cases:
  - If **A**=**I** (i.e., each node is adjacent only to itself), then the Graph Convolution (GC) layer simplifies to just a regular Fully Connected (FC) layer.

$$\mathbf{h} = \sigma\left(\qquad\qquad \mathbf{XW} \right)$$

Relationship to FC layer

- Special cases:
  - If **A**=**I** (i.e., each node is adjacent only to itself), then the Graph Convolution (GC) layer simplifies to just a regular Fully Connected (FC) layer.
  - If **A**=**1**/n (i.e., each node is adjacent to every node in the graph), then the GC layer performs average pooling.

$$\mathbf{h} = \sigma\left(\quad \mathbf{1/n} \quad \mathbf{XW} \right)$$

Connection between graph convolution and image convolution

Convolution is linear

- Recall that convolution is linear and can thus be implemented by multiplication with a matrix **W**, e.g.:

1
2
-2
0
3

Image

1
2
3

Filter

-1
-2
7

Feature map

$$\underbrace{\begin{bmatrix} 1 & 2 & 3 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 0 & 1 & 2 & 3 \end{bmatrix}}_{\mathbf{W}} \underbrace{\begin{bmatrix} 1 \\ 2 \\ -2 \\ 0 \\ 3 \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} -1 \\ -2 \\ 7 \end{bmatrix}}_{\mathbf{h}}$$

Wx = h

Review of 3D convolution

- Recall how 3D convolution (3x3 filter) on a stack of input images works.

R

1	9	5	-2
2	8	**6**	5
3	7	-5	-3
2	3	8	1

G

2	1	9	0
-3	0	**-7**	3
3	7	1	2
4	2	0	1

B

6	4	2	1
-2	3	**9**	-2
7	7	3	2
3	8	7	2

Multi-channel input

0	1	1
2	-1	4
-3	3	1

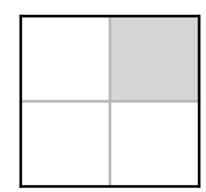
+

0	1	1
2	-1	2
0	-2	1

+

1	1	0
2	-1	4
2	3	-3

Filter

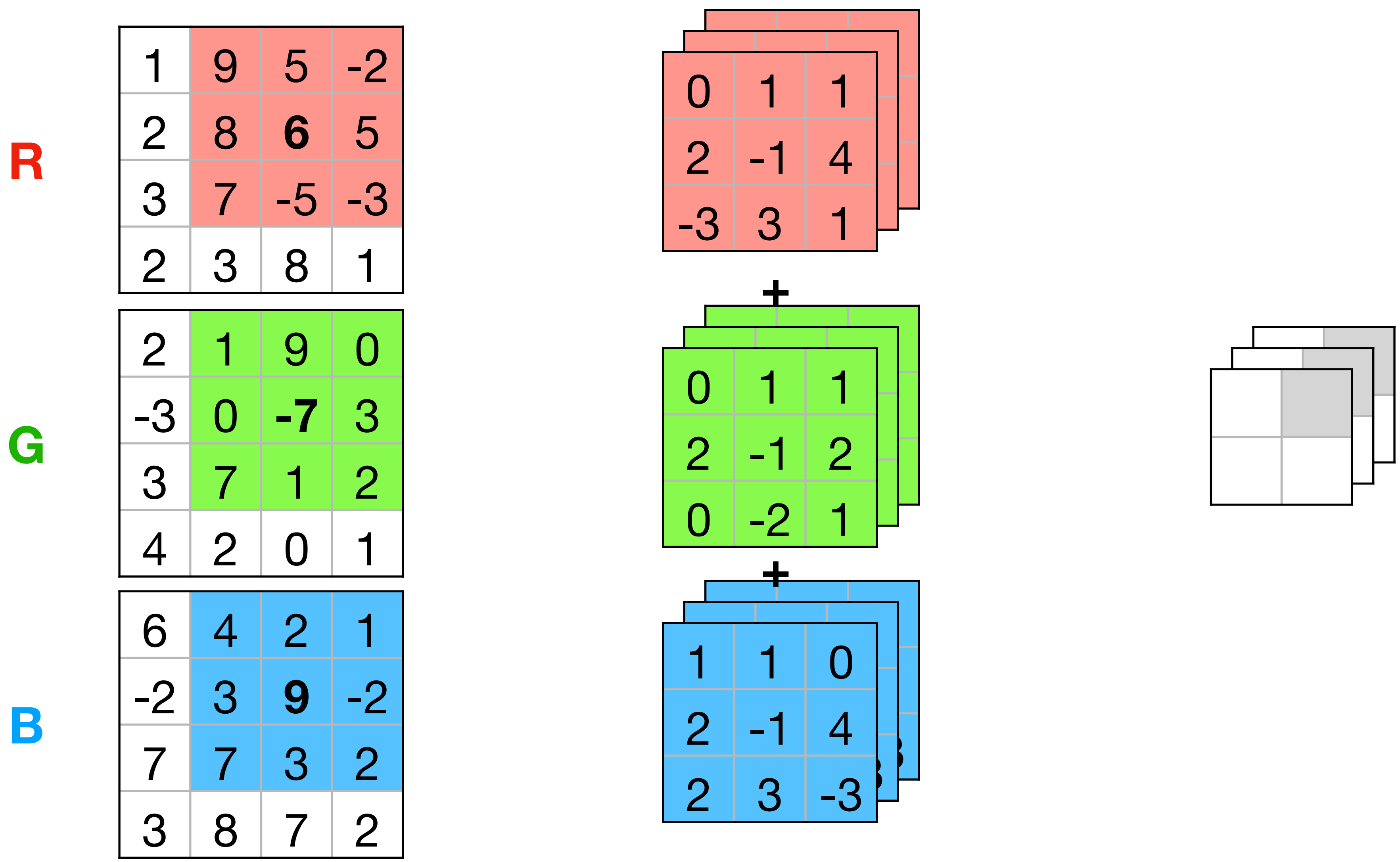


Output feature map

Jacob Whitehill, WPI

Review of 3D convolution

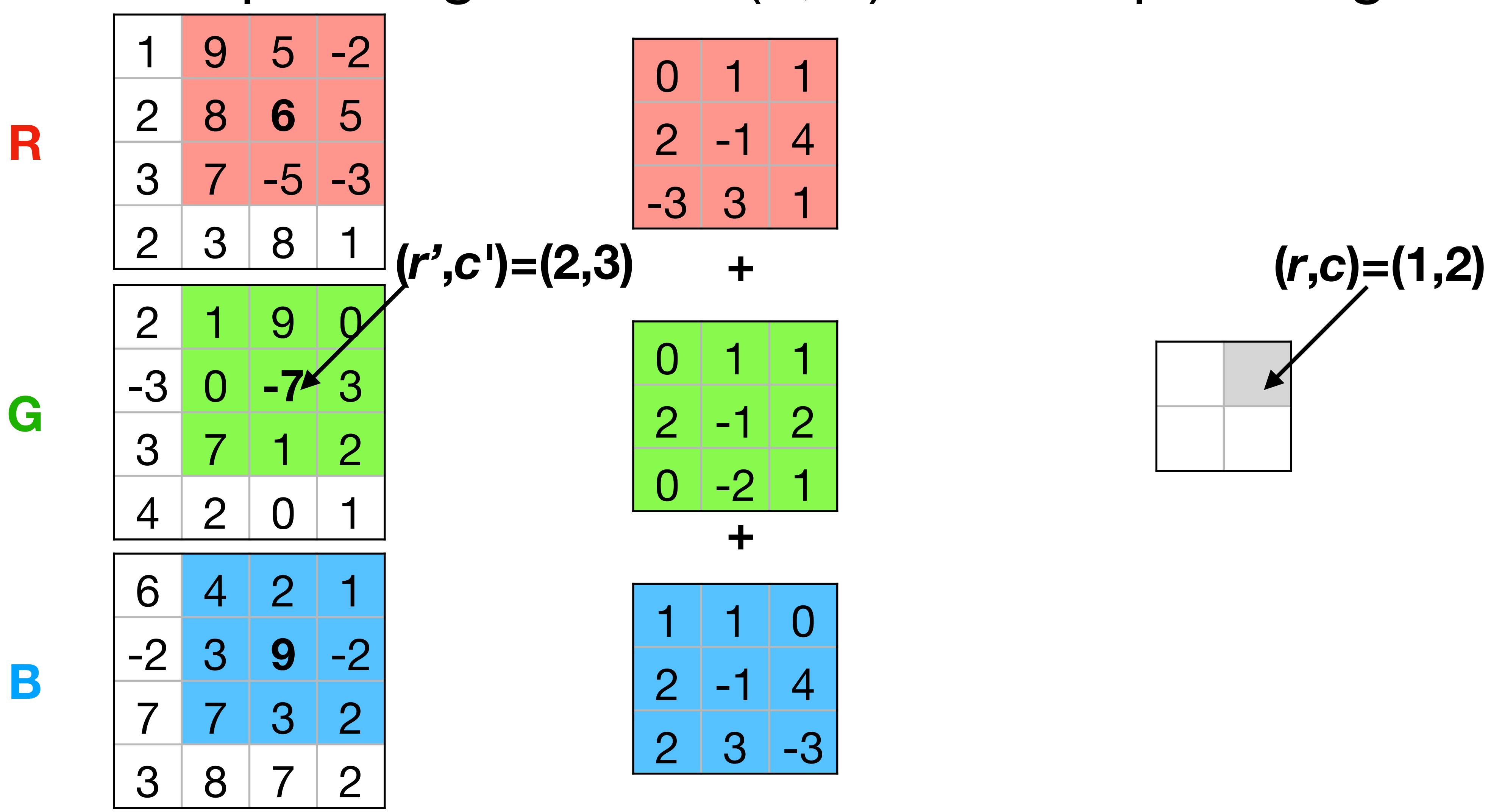
- We can also apply multiple such filters to obtain a stack of output feature maps.



Multi-channel input **Filters** **Output feature maps**

Review of 3D convolution

- The value of each element (r,c) of the output feature map depends on the set of (k*k) **neighbors** of the corresponding element (r',c') in the input image.



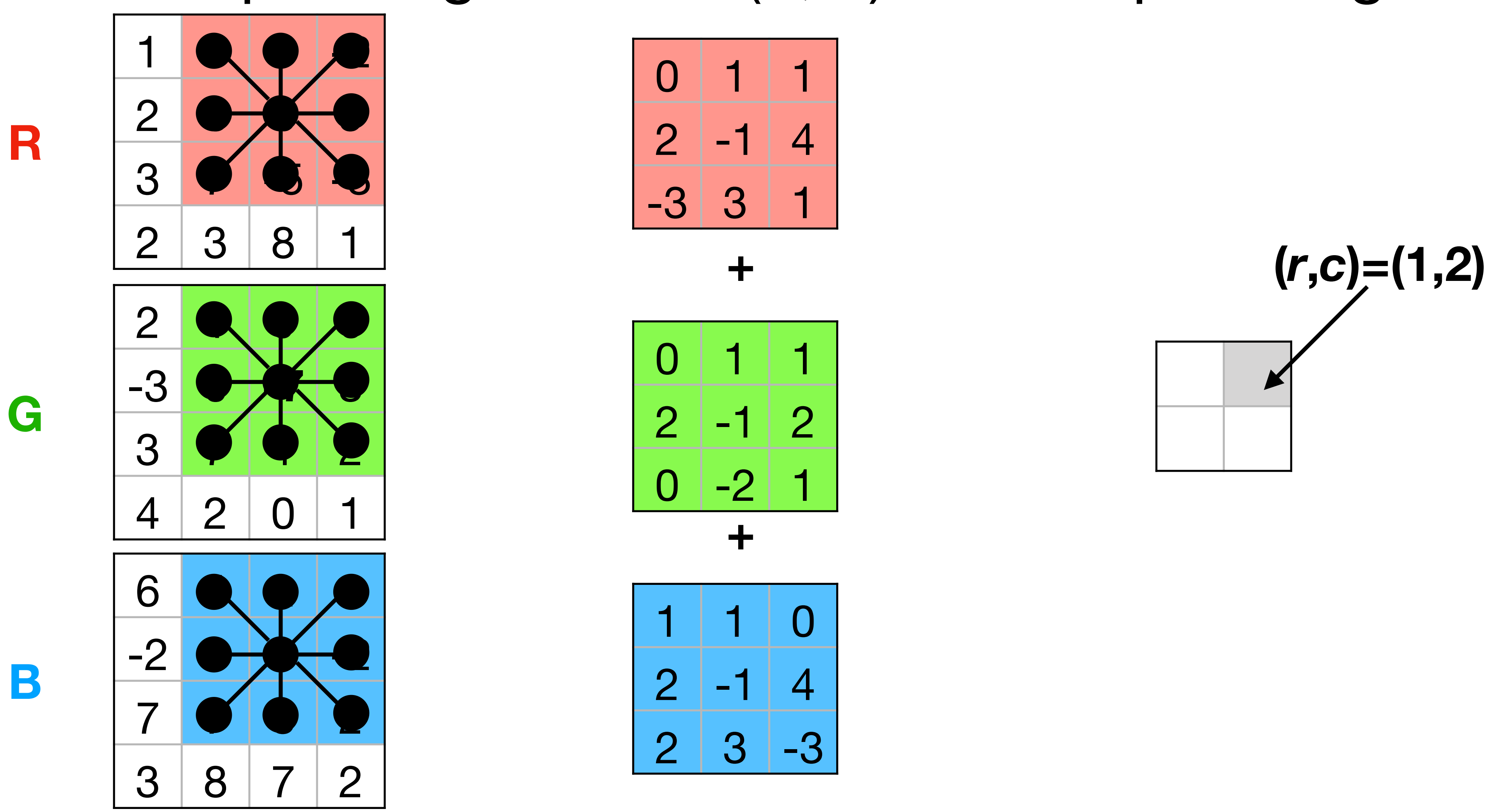
Multi-channel input

Filter

Output feature map

Review of 3D convolution

- The value of each element (r,c) of the output feature map depends on the set of (k*k) **neighbors** of the corresponding element (r',c') in the input image.



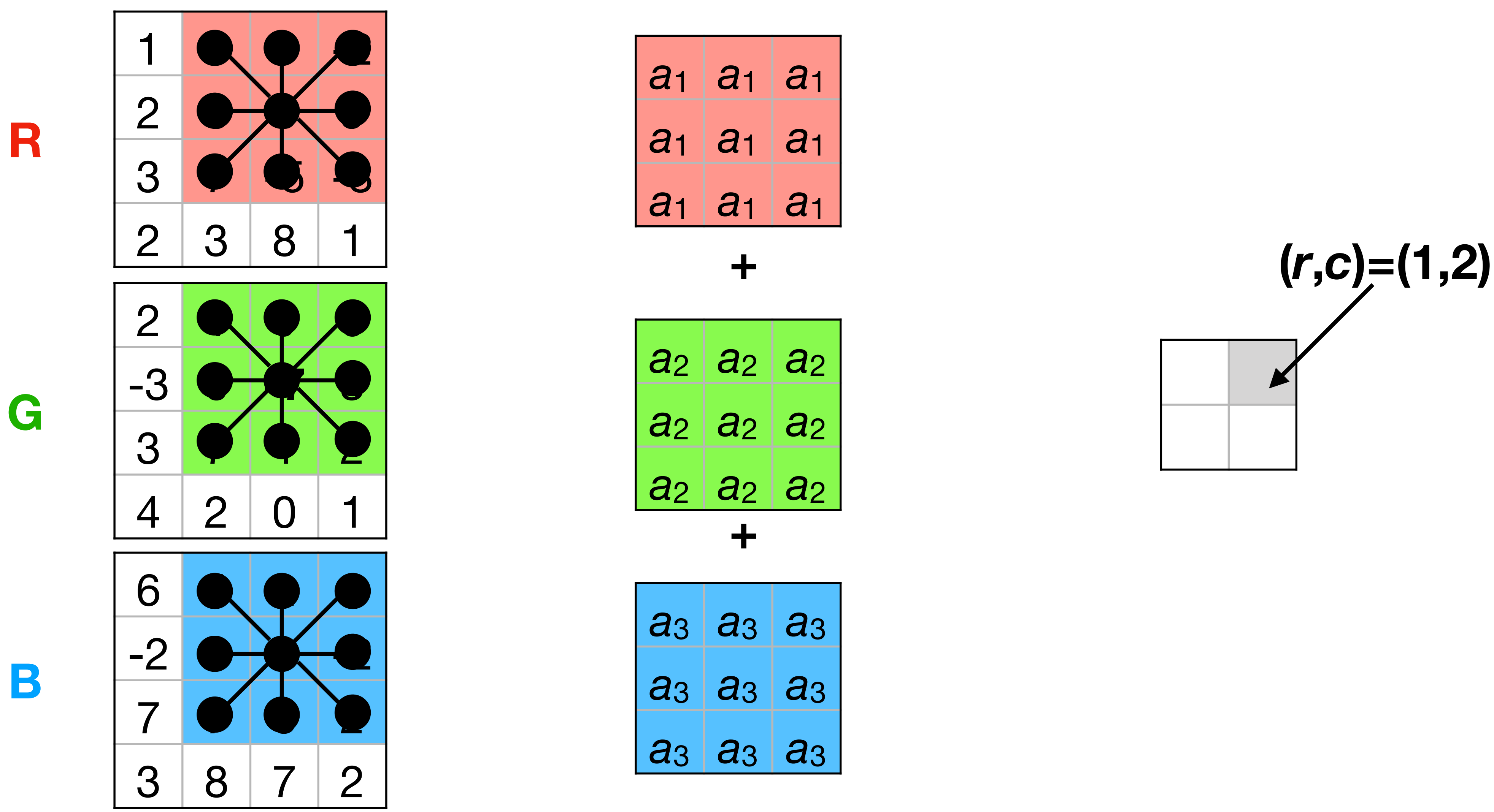
Multi-channel input

Filter

Output feature map

Special case

- Let's consider a special case where all the elements of each channel of the convolution filter are equal, i.e., there are just 3 learned parameters a_1, a_2, a_3.



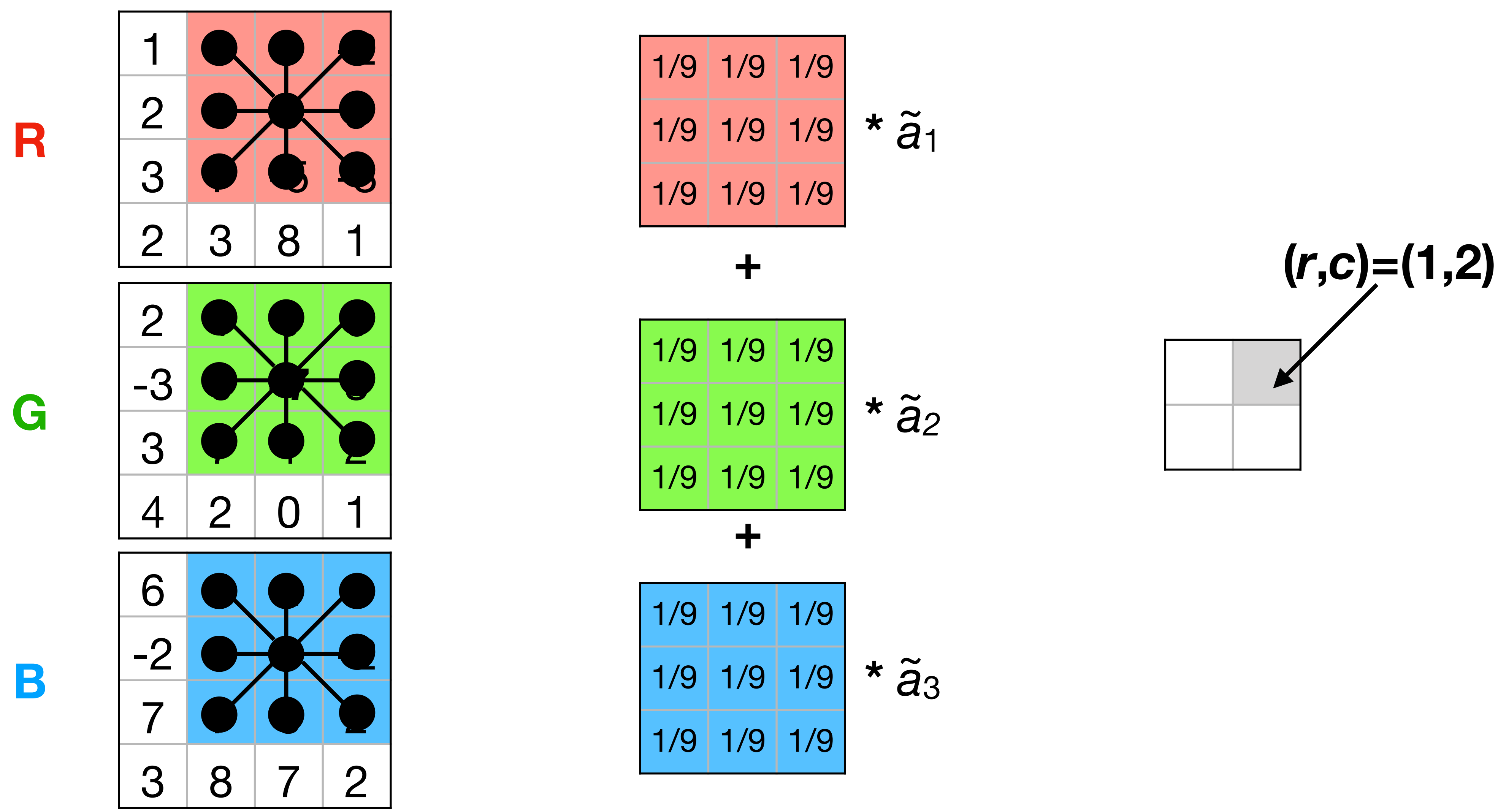
Multi-channel input

Filter

Output feature map

Special case

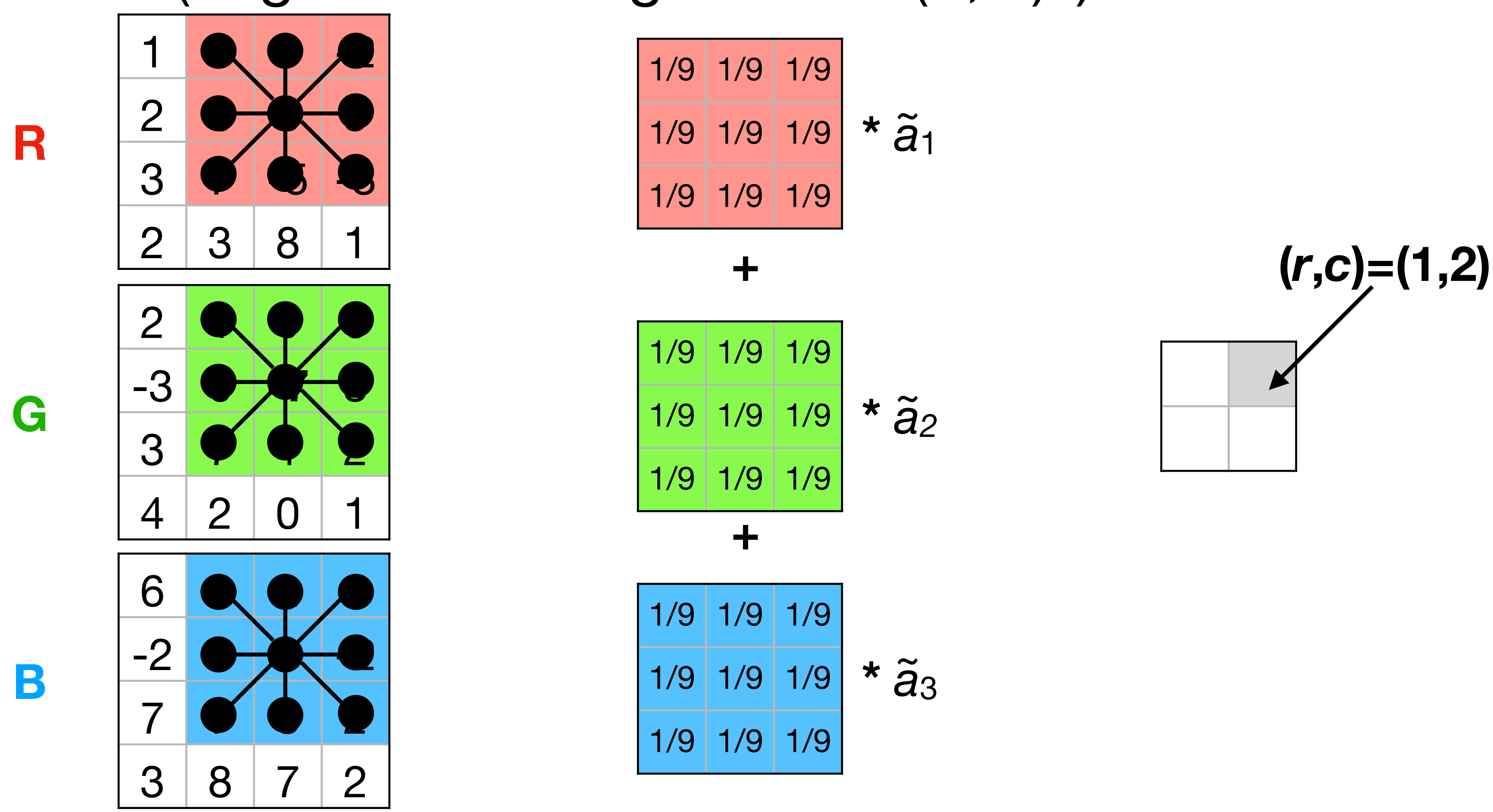
- Now let’s reparameterize it:



Multi-channel input **Filter** **Output feature map**

Special case

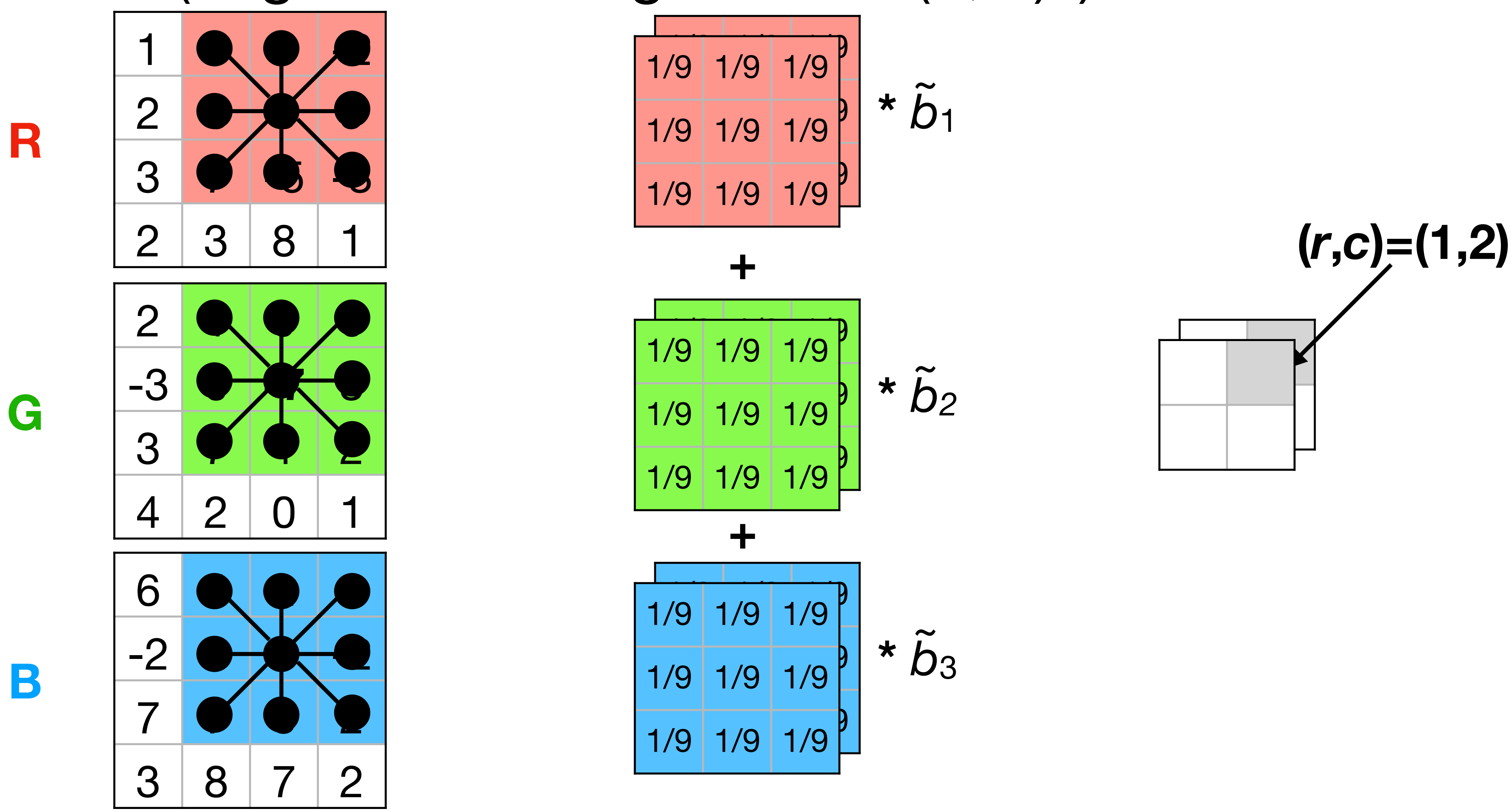
- In this case, the output feature map at (r,c) equals:
  $\tilde{a}_1$ * (Avg value of neighbors of $(r',c')^R$) +
  $\tilde{a}_2$ * (Avg value of neighbors of $(r',c')^G$) +
  $\tilde{a}_3$ * (Avg value of neighbors of $(r',c')^B$)



Multi-channel input **Filter** **Output feature map**

Special case

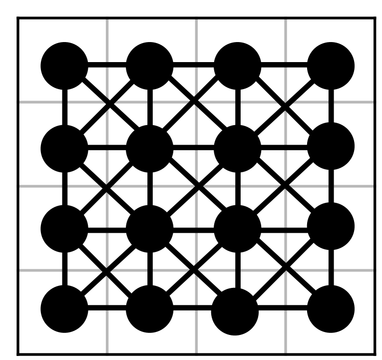
- For a second filter, the output feature map at (r,c) would equal:
  $\tilde{b}_1$ * (Avg value of neighbors of $(r',c')^R$) +
  $\tilde{b}_2$ * (Avg value of neighbors of $(r',c')^G$) +
  $\tilde{b}_3$ * (Avg value of neighbors of $(r',c')^B$)



Multi-channel input **Filter** **Output feature map**

Images as graphs

- Let us now represent an image as an undirected graph with:
  - One node per pixel.
  - An edge between each pair of adjacent pixels.



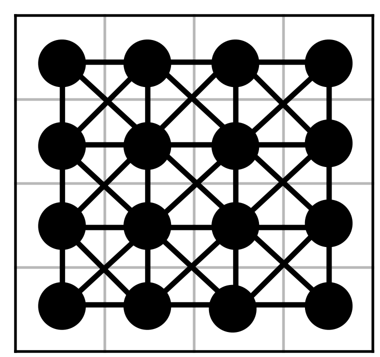
Images as graphs

- We can represent this graph using an **adjacency matrix.**
- In particular, for an image with *n* pixels (*n*=16 in this example), let $\mathbf{A} \in \{0, 1\}^{n \times n}$ where A_{ij}=1 iff pixel *i* neighbors pixel *j*.
- Also define diagonal matrix **D** to count the total number of neighbors of each node *i*: $D_{ii} = \sum_j A_{ij}$

  If we include self-connections (A_{ii}=1) then D_{ii} is 9 for interior pixels, 6 for edges, and 4 for corners in this example.)

1	1			1	1										
1	1	1		1	1	1									
	1	1	1		1	1	1								
		1	1			1	1								
1	1			1	1			1	1						
1	1	1		1	1	1		1	1	1					
	1	1	1		1	1	1		1	1	1				
			1	1		1	1			1	1				
				1	1			1	1			1	1		
				1	1	1		1	1	1		1	1	1	
					1	1	1		1	1	1		1	1	1
						1	1			1	1			1	1
								1	1			1	1		
								1	1	1		1	1	1	
									1	1	1		1	1	1
										1	1			1	1

A



4															
	6														
		6													
			4												
				6											
					9										
						9									
							6								
								6							
									9						
										9					
											6				
												4			
													6		
														6	
															4

D

Images as graphs

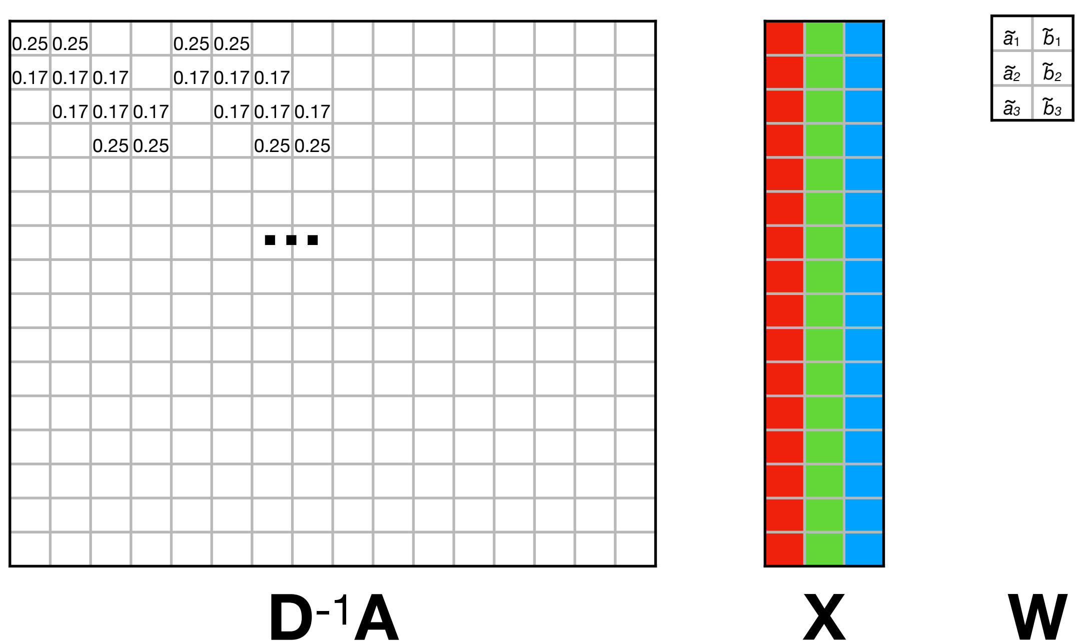
- By computing $\mathbf{D}^{-1}\mathbf{A}$, we can compute a normalized adjacency matrix:

0.25	0.25			0.25	0.25										
0.17	0.17	0.17		0.17	0.17	0.17									
	0.17	0.17	0.17		0.17	0.17	0.17								
		0.25	0.25			0.25	0.25								
							...								

$\mathbf{D}^{-1}\mathbf{A}$

Images as graphs

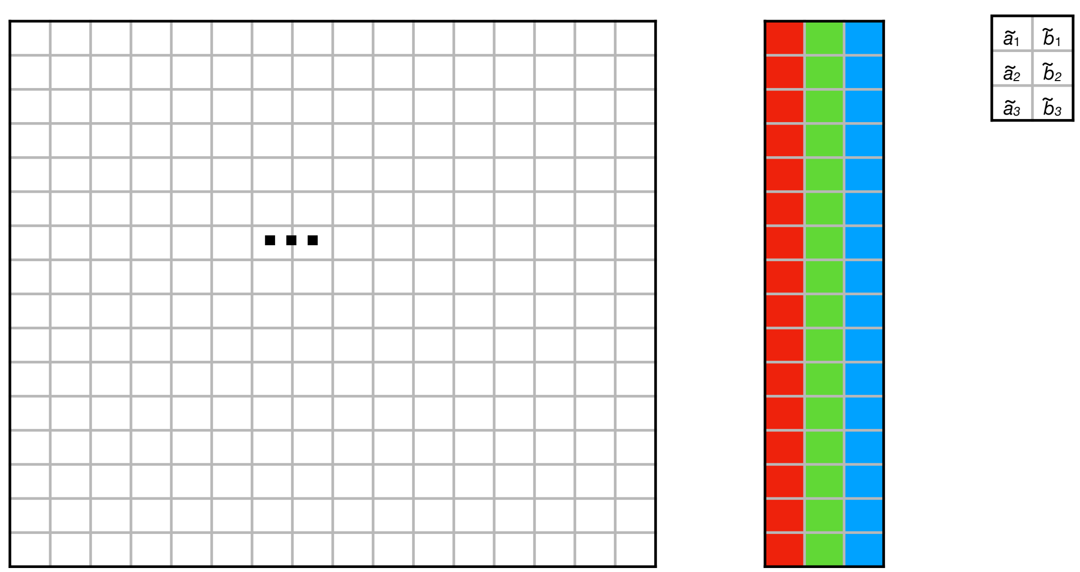
- We can now express this convolution as the product of the normalized adjacency matrix $\mathbf{D}^{-1}\mathbf{A}$ with the input features $\mathbf{X}$ and the filter weights $\mathbf{W}$:



Jacob Whitehill, WPI

Images as graphs

- We can now express this convolution as the product of the normalized adjacency matrix $\mathbf{D}^{-1}\mathbf{A}$ with the input features $\mathbf{X}$ and the filter weights $\mathbf{W}$:

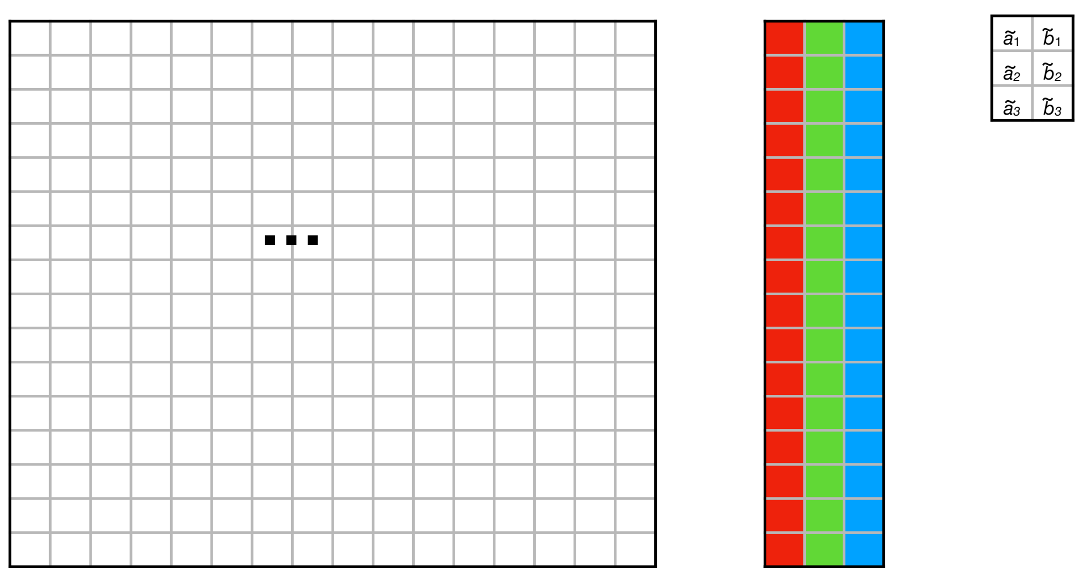


$\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ $\mathbf{X}$ $\mathbf{W}$

- A related matrix is given by $\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$.

Images as graphs

- We can now express this convolution as the product of the normalized adjacency matrix $\mathbf{D}^{-1}\mathbf{A}$ with the input features $\mathbf{X}$ and the filter weights $\mathbf{W}$:



$\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ $\quad$ $\mathbf{X}$ $\quad$ $\mathbf{W}$

$$\mathbf{h} = \sigma(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{X}\mathbf{W})$$

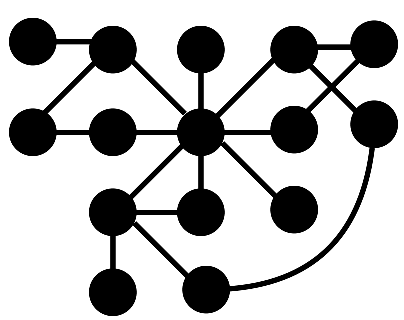
- A related matrix is given by $\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$.

- We can then apply a non-linear activation function σ.

Images as graphs

- We can apply the same procedure to general graphs.
- Moreover, the number of neighbors of each node does not have to be the same.

$$\mathbf{h} = \sigma\left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{X}\mathbf{W}\right)$$



- Intuitively, we convolve the features of each graph node with a fixed filter (with equal value for all spatial neighbors) and renormalize according to the neighborhood structure.
- This is the basic operation of a **graph convolutional neural network** (GCN) (as proposed by Kipf & Welling 2017).

Exercises

https://cs230.stanford.edu/files/cs230exam_fall20_soln.pdf

Exercise 1

(2 points) You are training a large feedforward neural network (100 layers) on a binary classification task, using a sigmoid activation in the final layer, and a mixture of *tanh* and *ReLU* activations for all other layers. You notice your weights to your a *subset* of your layers stop updating after the first epoch of training, even though your network has not yet converged. Deeper analysis reveals the *gradients* to these layers completely, or almost completely, go to zero very early on in training. Which of the following fixes could help? (You also note that your loss is still within a reasonable order of magnitude).

(i) Increase the size of your training set

(ii) Switch the ReLU activations with leaky ReLUs everywhere

(iii) Add Batch Normalization before every activation

(iv) Increase the learning rate

Exercise 2

(2 points) Which of the following would you consider to be valid activation functions (elementwise non-linearities) to train a neural net in practice?

(i) $f(x) = -\min(2, x)$

(ii) $f(x) = 0.9x + 1$

(iii) $f(x) = \begin{cases} \min(x, .1x) & \mid x >= 0 \\ \min(x, .1x) & \mid x < 0 \end{cases}$

(iv) $f(x) = \begin{cases} \max(x, .1x) & \mid x >= 0 \\ \min(x, .1x) & \mid x < 0 \end{cases}$

Exercise 3

(2 points) During backpropagation, as the gradient flows backward through a *sigmoid* non-linearity, the gradient will always:

(i) Increase in magnitude, maintain polarity

(ii) Increase in magnitude, reverse polarity

(iii) Decrease in magnitude, maintain polarity

(iv) Decrease in magnitude, reverse polarity

Exercise 4

(2 points) You are benchmarking runtimes for layers commonly encountered in CNNs. Which of the following would you expect to be the fastest (in terms of floating point operations)?

(i) Conv layer (convolution operation + bias addition)

(ii) Max pooling

(iii) Average pooling

(iv) Batch Normalization

Exercise 5

(2 points) You come across a nonlinear function that passes 1 if its input is nonnegative, else evaluates to 0, i.e.

$$f(x) = \begin{cases} 1 & | \; x >= 0 \\ 0 & | \; x < 0 \end{cases}$$

A friend recommends you use this non-linearity in your convolutional neural network with the Adam optimizer. Would you follow their advice? Why or why not?