UNIVERSITY OF CAMBRIDGE

Adversarial learning meets graphs

. . . and why should you care?

Petar Veličković

Artificial Intelligence Group
Department of Computer Science and Technology, University of Cambridge, UK

MILA Adversarial Learning Reading Group Meeting **25 September 2018**

Introduction

- In this talk, I will attempt to give you a comprehensive overview of the current state-of-the-art on the intersection of **graph-based neural networks** and **adversarial learning**.
- The discussion will span the following research directions:
  - *Graph convolutional networks*;
  - *Generative models of graphs*;
  - *Semi-supervised adversarial learning on graphs*;
  - *Graph-based adversarial defence*.
- I assume *no prior knowledge of graph neural networks*.
  - Please stop me whenever necessary!
  - Disclaimer: I'm a graph person, *not a GAN person*.

Roadmap for today

Graph convolutional networks

- GCN (Kipf & Welling, ICLR 2017)
- MPNN (Gilmer *et al.*, ICML 2017)
- GAT (Veličković *et al.*, ICLR 2018)

Generative models of graphs

- MolGAN (De Cao & Kipf, ICML TADGM 2018)
- GCPN (You *et al.*, NIPS 2018)

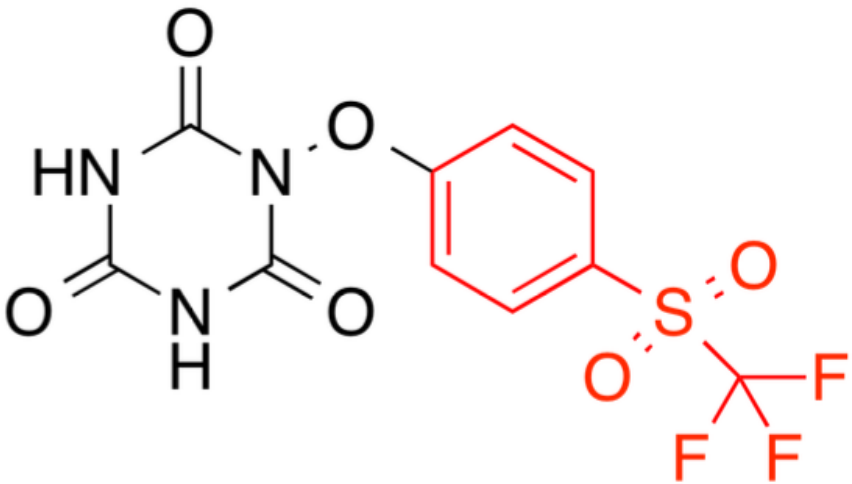
Semi-supervised adversarial learning on graphs

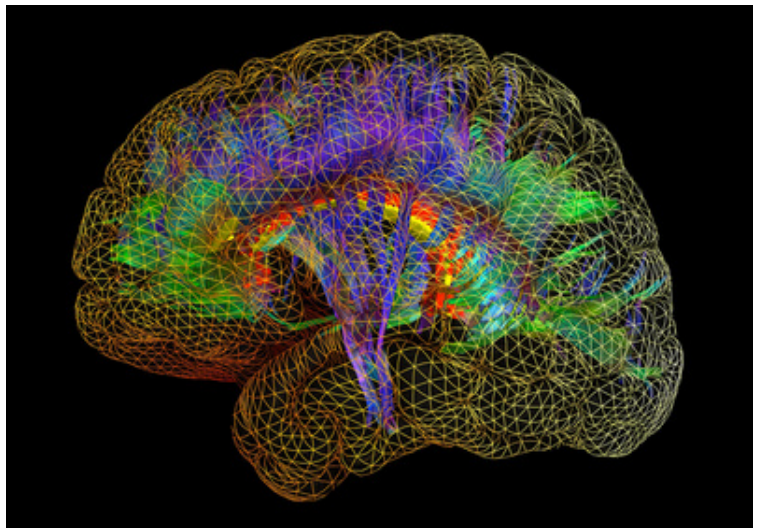
- GraphSGAN (Ding *et al.*, CIKM 2018)

Graph-based adversarial defence

- PeerNets (Svoboda *et al.*, 2018)

Graphs are **everywhere**!





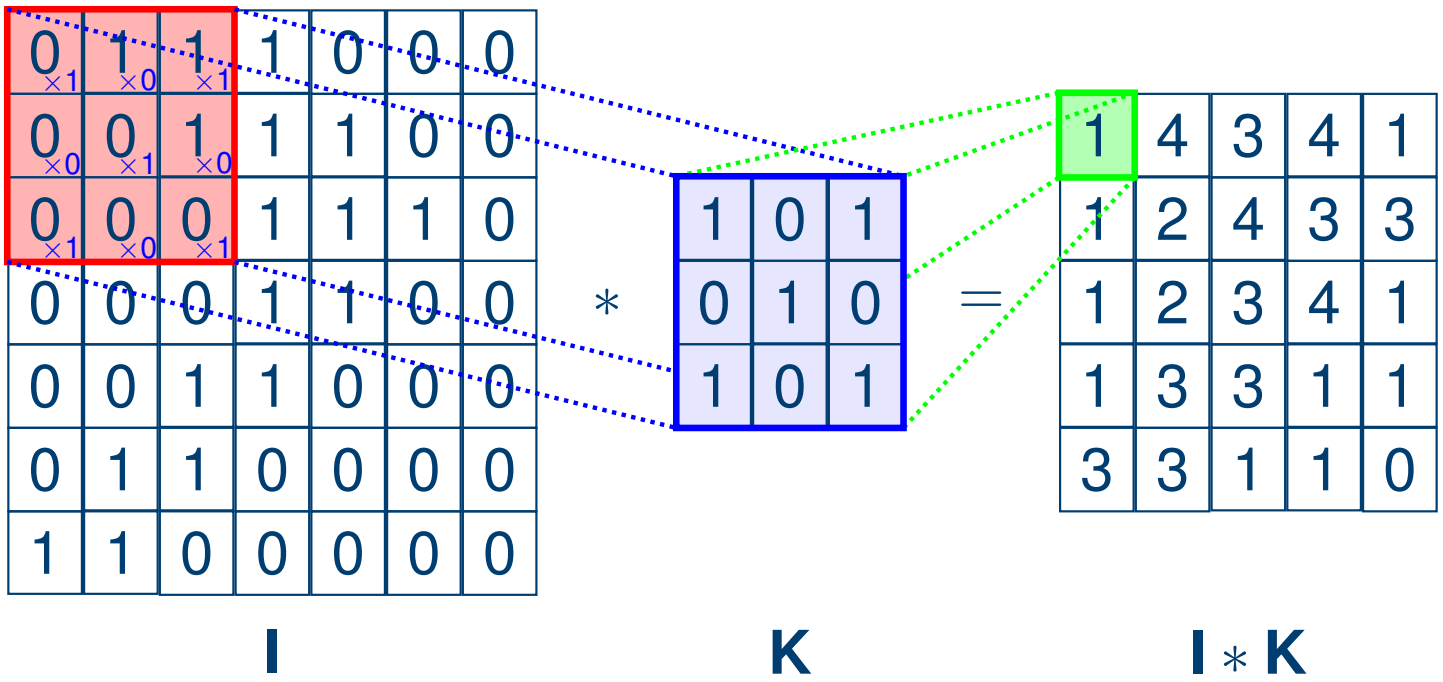
Mathematical setup

- We will usually think of graphs, $G = (V, E)$, in terms of sets of *nodes*, V, and *edges*, E, between them.
- Each node may contain a certain set of features. This is represented by a *node feature matrix*, $\mathbf{F} \in \mathbb{R}^{N \times F}$, with F features in each of the N nodes.
- We will represent edges as an *adjacency matrix*, $\mathbf{A} \in \mathbb{R}^{N \times N}$.
  - The entries of $\mathbf{A}$ may be binary, or real-valued—or even consist of arbitrary *edge features*! Also will often assume *sparsity*.
- We will denote the *neighbourhood* of node i by $\mathcal{N}_i$. It will usually consist of all of i's first-order neighbours, including i itself, i.e. $\mathcal{N}_i = \{j \mid i = j \vee \mathbf{A}_{ij} \neq 0\}$.

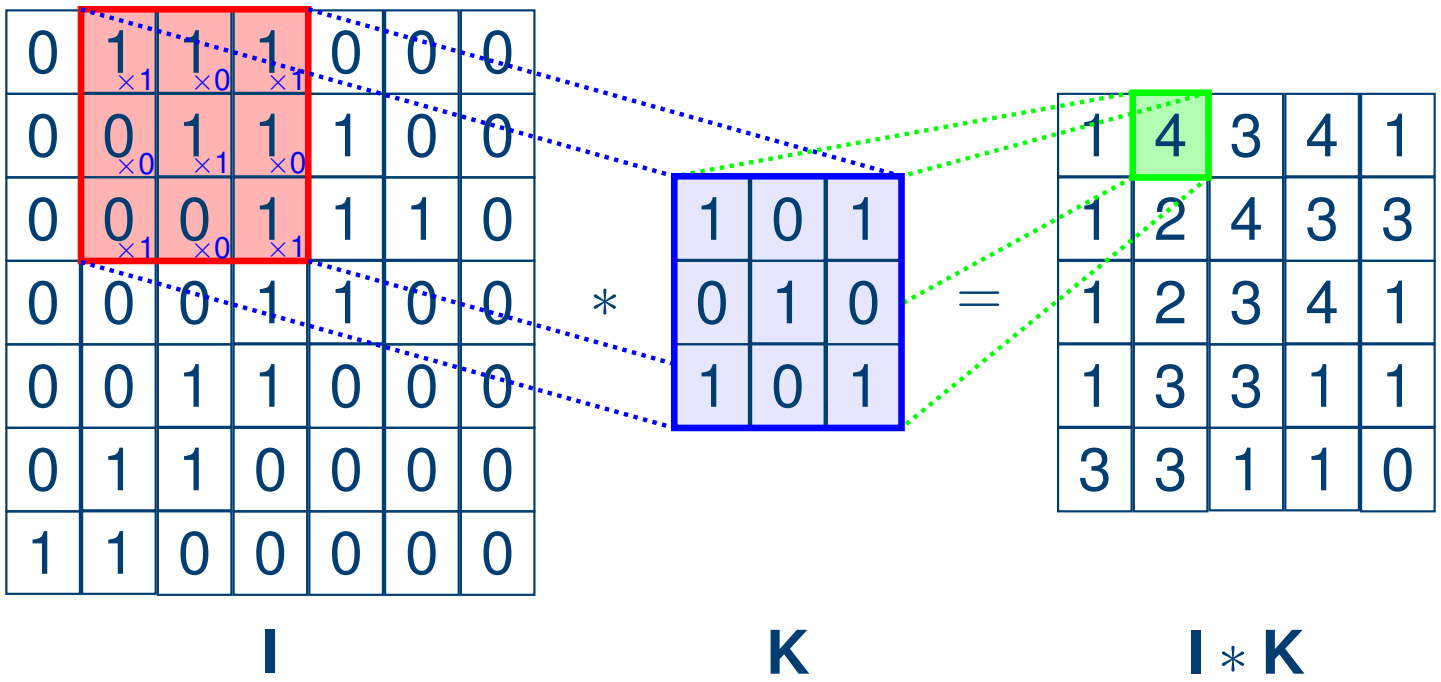
The silver bullet—a *convolutional* layer

- Graphs can be seen as a strict *generalisation* of **images.**
  - Can represent any image as a “grid graph” (every node incident to its four neighbours) with pixel values as node features.
- It would be, therefore, highly appropriate if we could somehow generalise the *convolutional operator* (as used in CNNs) to operate on arbitrary graphs!
- This will eventually result in a *graph convolutional network*.

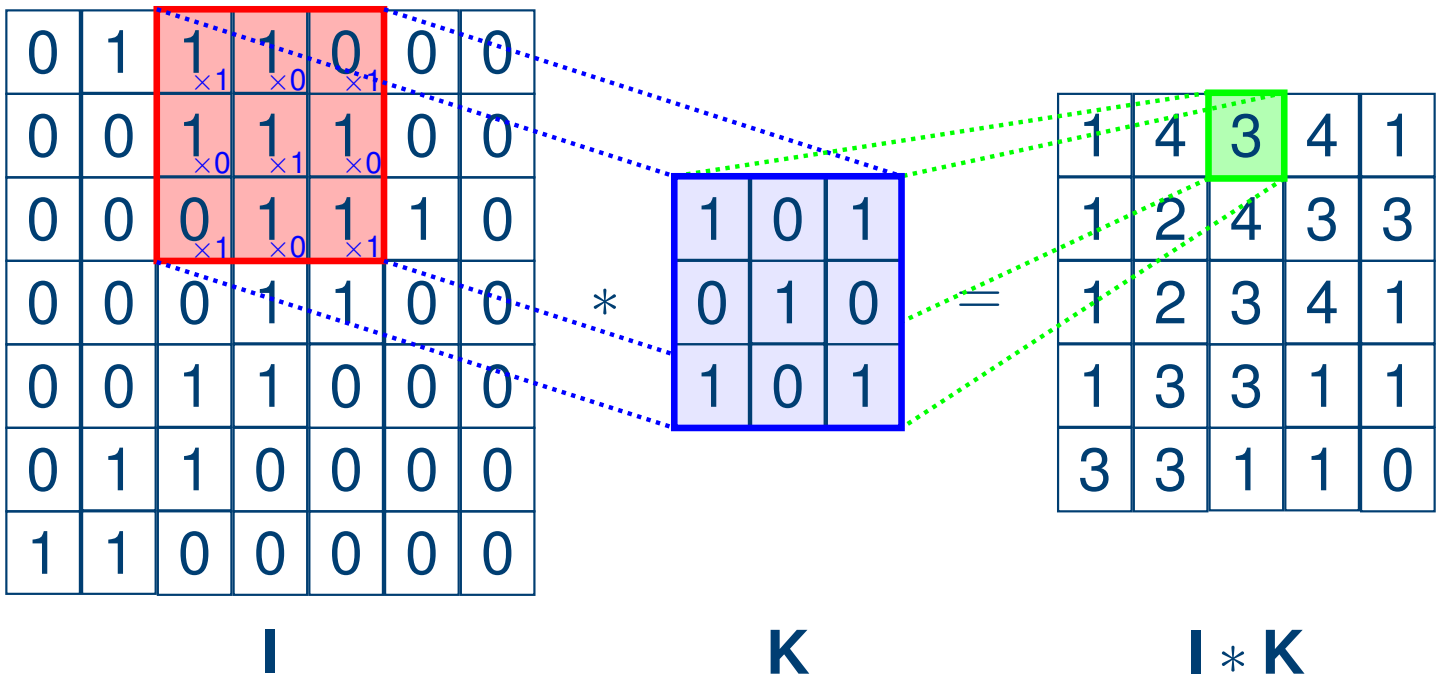
Convolution on images



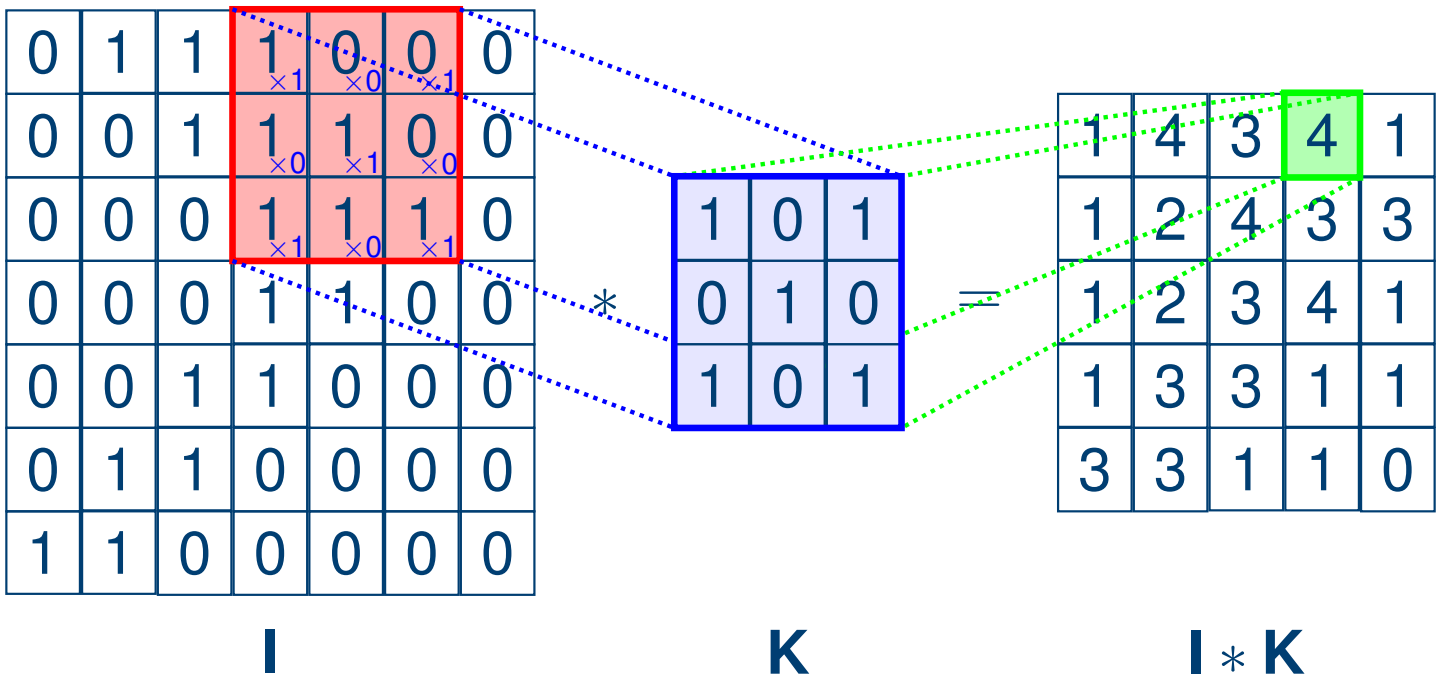
Convolution on images



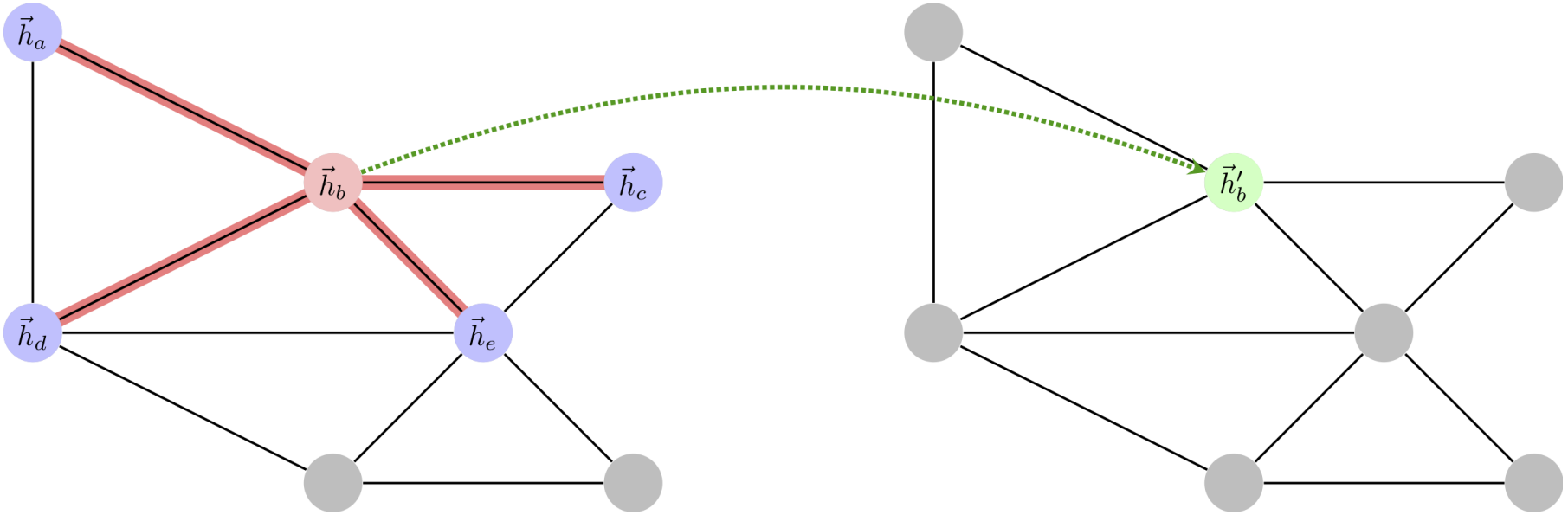
Convolution on images



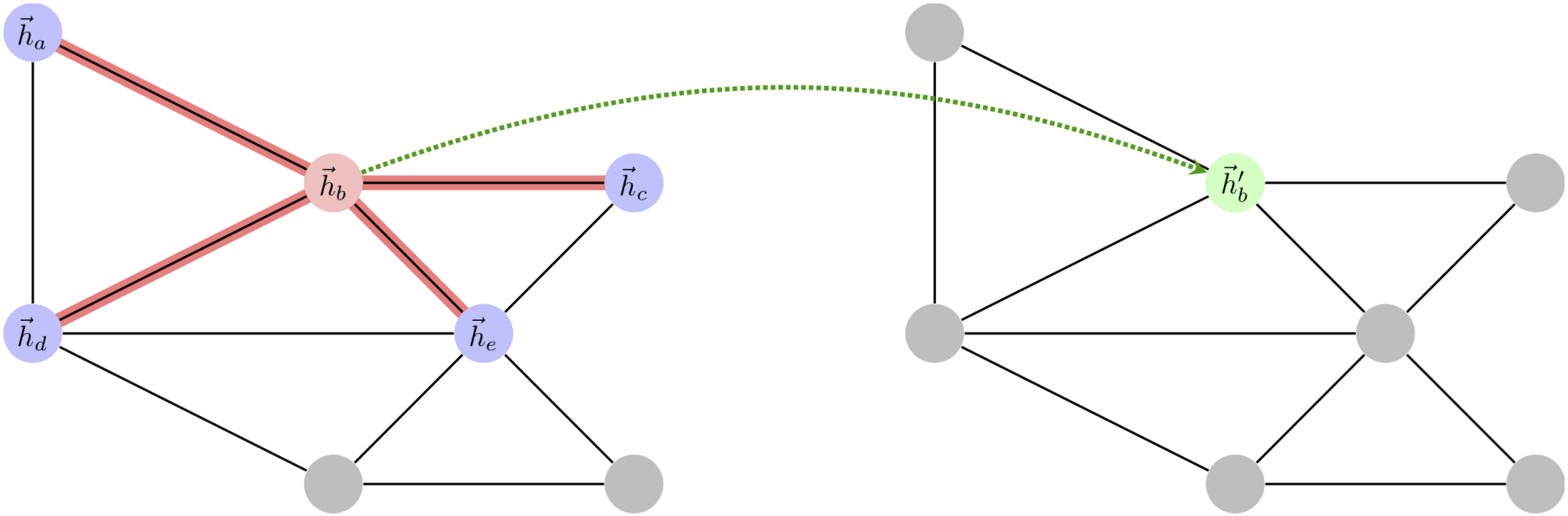
Convolution on images



Graph convolutional network?



Graph convolutional network?



In a nutshell, obtain higher-level representations of a node *i* by leveraging its *neighbourhood*, $\mathcal{N}_i$!

$$\vec{h}_i^{\ell+1} = g^{\ell}(\vec{h}_a^{\ell}, \vec{h}_b^{\ell}, \vec{h}_c^{\ell}, \dots) \;\; (a, b, c, \cdots \in \mathcal{N}_i)$$

where g^{ℓ} is the ℓ-th *graph convolutional layer*.

Challenges with graph convolutions

- Desirable properties for a graph convolutional layer:
  - **Computational and storage efficiency** ($\sim O(V + E)$);
  - **Fixed** number of parameters (independent of input size);
  - **Localisation** (acts on a *local neighbourhood* of a node);
  - Specifying **different importances** to different neighbours;
  - Applicability to **inductive problems**.
- Fortunately, images have a highly rigid and regular connectivity pattern, making such an operator trivial to deploy (as a small kernel matrix which is slided across).
- Arbitrary graphs are a **much harder** challenge!

Roadmap for today

Graph convolutional networks

- **GCN (Kipf & Welling, ICLR 2017)**
- **MPNN (Gilmer *et al.*, ICML 2017)**
- **GAT (Veličković *et al.*, ICLR 2018)**

Generative models of graphs

- MolGAN (De Cao & Kipf, ICML TADGM 2018)
- GCPN (You *et al.*, NIPS 2018)

Semi-supervised adversarial learning on graphs

- GraphSGAN (Ding *et al.*, CIKM 2018)

Graph-based adversarial defence

- PeerNets (Svoboda *et al.*, 2018)

Towards a simple update rule

- Let's assume our graph is *unweighted* and *undirected*.
  - That is, $A_{ij} = A_{ji} = \begin{cases} 1 & i \leftrightarrow j \\ 0 & \textit{otherwise} \end{cases}$
- We can then easily aggregate neighbourhoods through multiplying by the adjacency matrix!

$$\mathbf{H}' = \sigma\left(\mathbf{AHW}\right)$$

  where $\mathbf{W}$ is a learnable node-wise shared linear transformation, and σ is a nonlinearity.
- A few things need to be fixed...

Towards a simple update rule, *cont'd*

- Firstly, this update rule discards the central node. Provide a simple correction:

$$\mathbf{H}' = \sigma\left(\tilde{\mathbf{A}}\mathbf{H}\mathbf{W}\right)$$

  where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$.

- The update rule can now be rewritten, node-wise, as:

$$\vec{h}_i' = \sigma\left(\sum_{j \in \mathcal{N}_i} \mathbf{W}\vec{h}_j\right)$$

The mean-pooling update rule

- Secondly, multiplication by $\mathbf{A}$ may increase the scale of the output features—we need to normalise appropriately, e.g. by

$$\mathbf{H}' = \sigma\left(\tilde{\mathbf{D}}^{-1}\tilde{\mathbf{A}}\mathbf{H}\mathbf{W}\right)$$

  where $\tilde{\mathbf{D}}$ is the degree matrix of $\tilde{\mathbf{A}}$, i.e. $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$.
- We arrive at the *mean-pooling* update rule:

$$\vec{h}'_i = \sigma\left(\sum_{j \in \mathcal{N}_i} \frac{1}{|\mathcal{N}_i|}\mathbf{W}\vec{h}_j\right)$$

  which is simple but versatile (common for *inductive* problems!).

GCN (Kipf & Welling, ICLR 2017)

- If we instead use *symmetric normalisation*:

$$\mathbf{H}' = \sigma\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}\mathbf{W}\right)$$

  we obtain the *graph convolutional network* (**GCN**) update rule!

- Node-wise, this can be written as follows:

$$\vec{h}'_i = \sigma\left(\sum_{j\in\mathcal{N}_i}\frac{1}{\sqrt{|\mathcal{N}_i||\mathcal{N}_j|}}\mathbf{W}\vec{h}_j\right)$$

  and it represents the currently most popular graph convolutional layer (simple and powerful, albeit not inductive).

Towards a more general update rule

- The GCN model only indirectly supports *edge features*.
- One way to correct this is to instead focus on *edge-wise* mechanisms: most generally, nodes can send *messages* (arbitrary vectors) along edges of the graph!
  - These messages can be *conditioned* by edge features.
- Then, a node can aggregate all messages sent to it.

MPNN (Gilmer *et al.*, ICML 2017)

- Let $\vec{m}_{ij}$ be the message sent across edge $i \to j$, computed using a message function $f_e : \mathbb{R}^N \times \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}^K$:

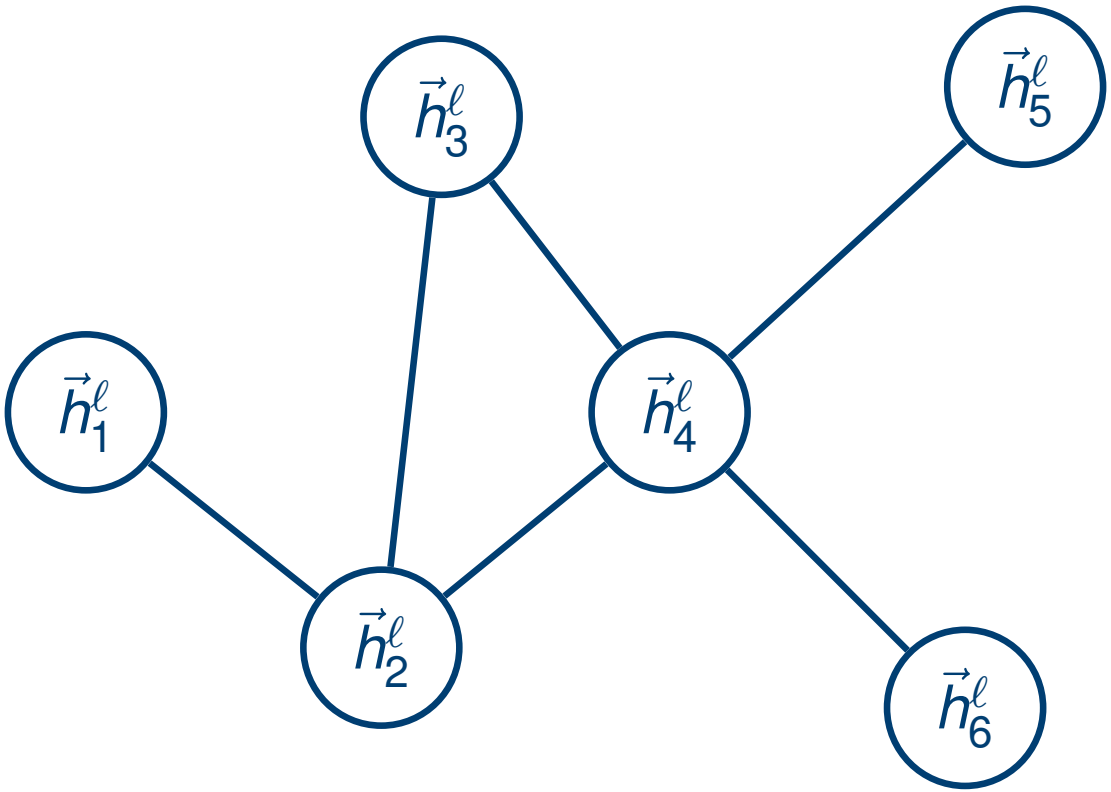
$$\vec{m}_{ij} = f_e\left(\vec{h}_i, \vec{h}_j, \vec{e}_{ij}\right)$$

- Now, aggregating all messages entering a node, transformed using an aggregation function $f_v : \mathbb{R}^N \times \mathbb{R}^K \to \mathbb{R}^N$:

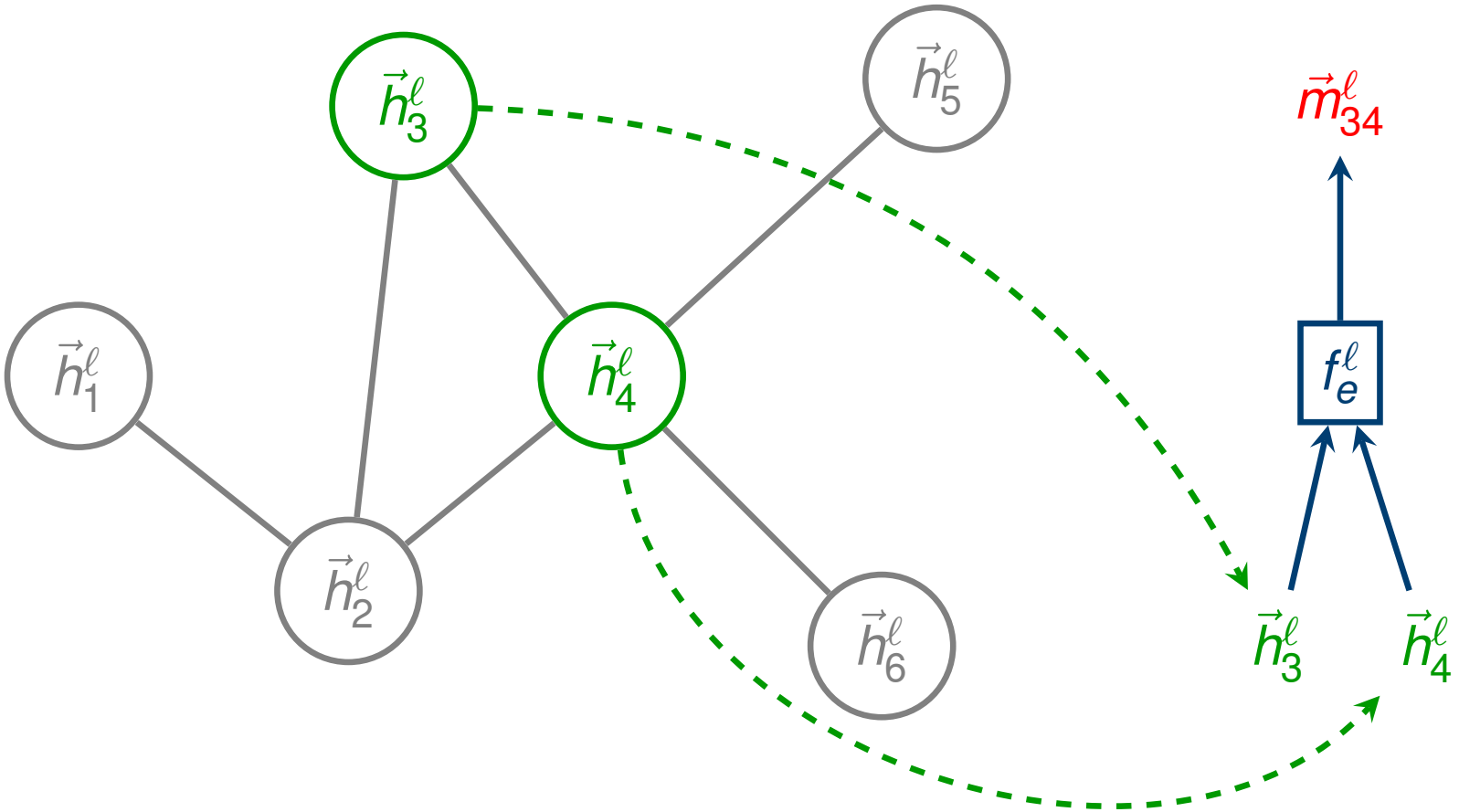
$$\vec{h}'_i = f_v\left(\vec{h}_i, \sum_{j \in \mathcal{N}_i} \vec{m}_{ij}\right)$$

  we arrive at the *message-passing neural network* (**MPNN**)!
  - f_e and f_v are usually (small) MLPs.

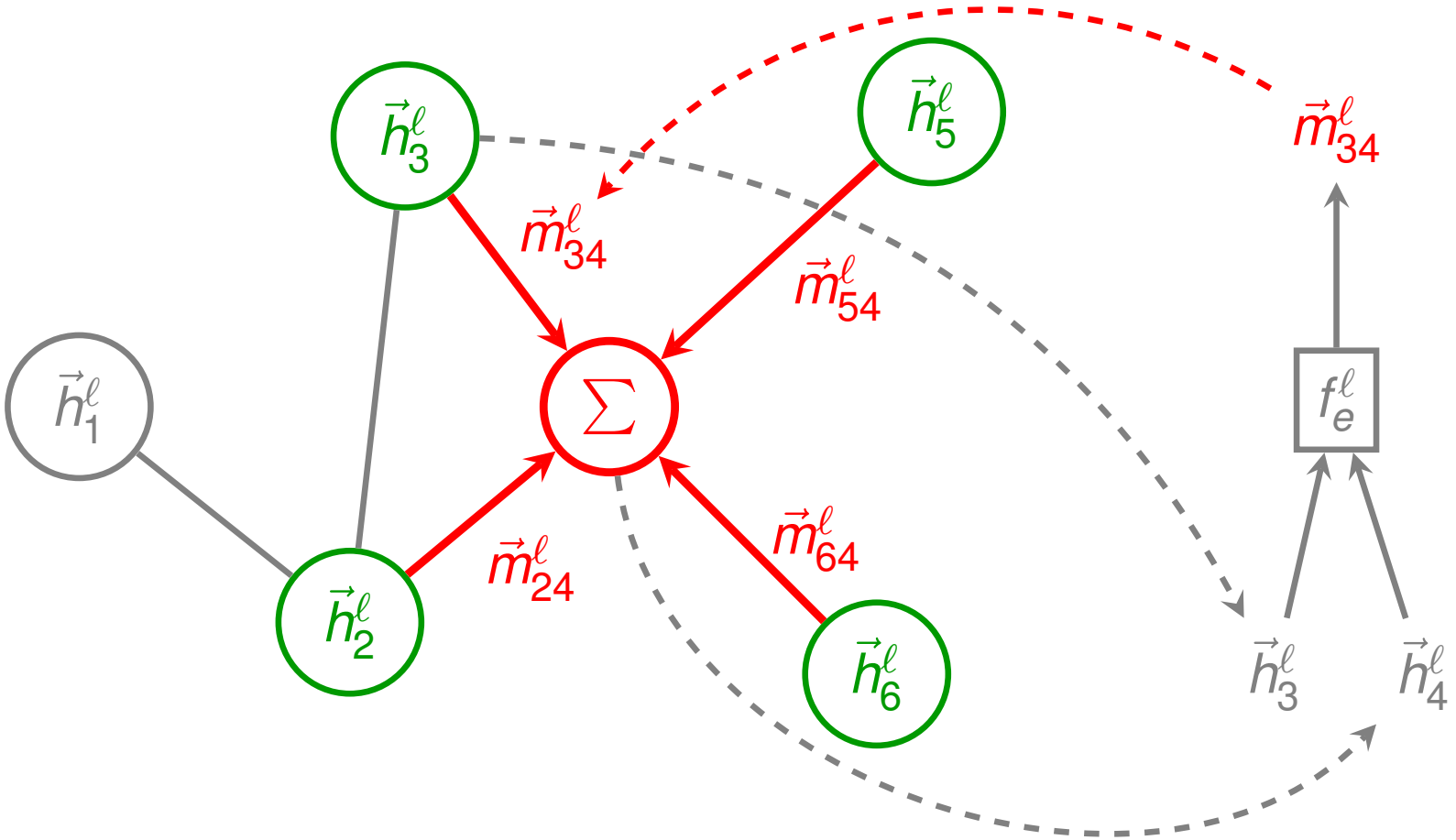
MPNN: initial setup



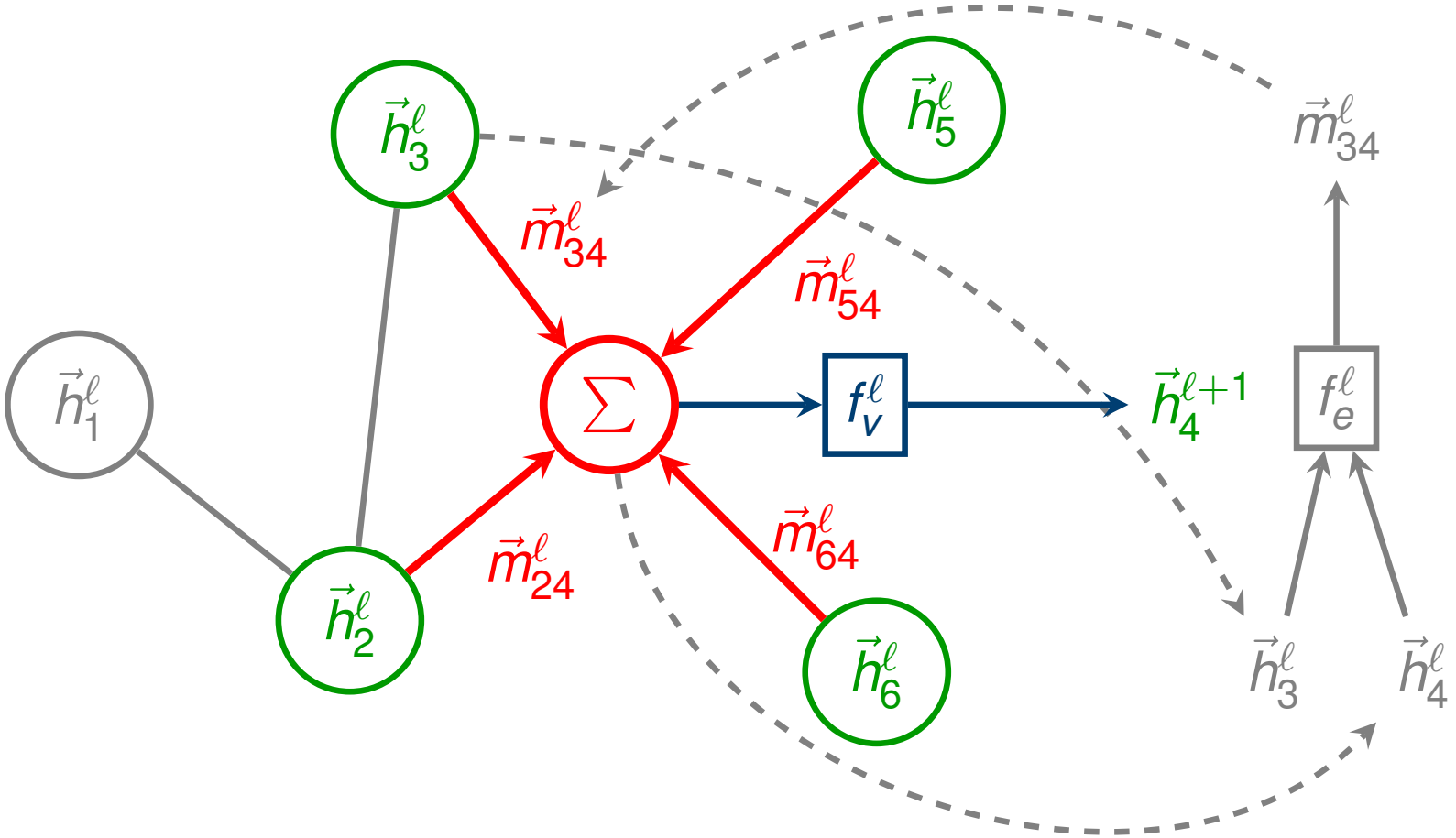
MPNN, computing messages



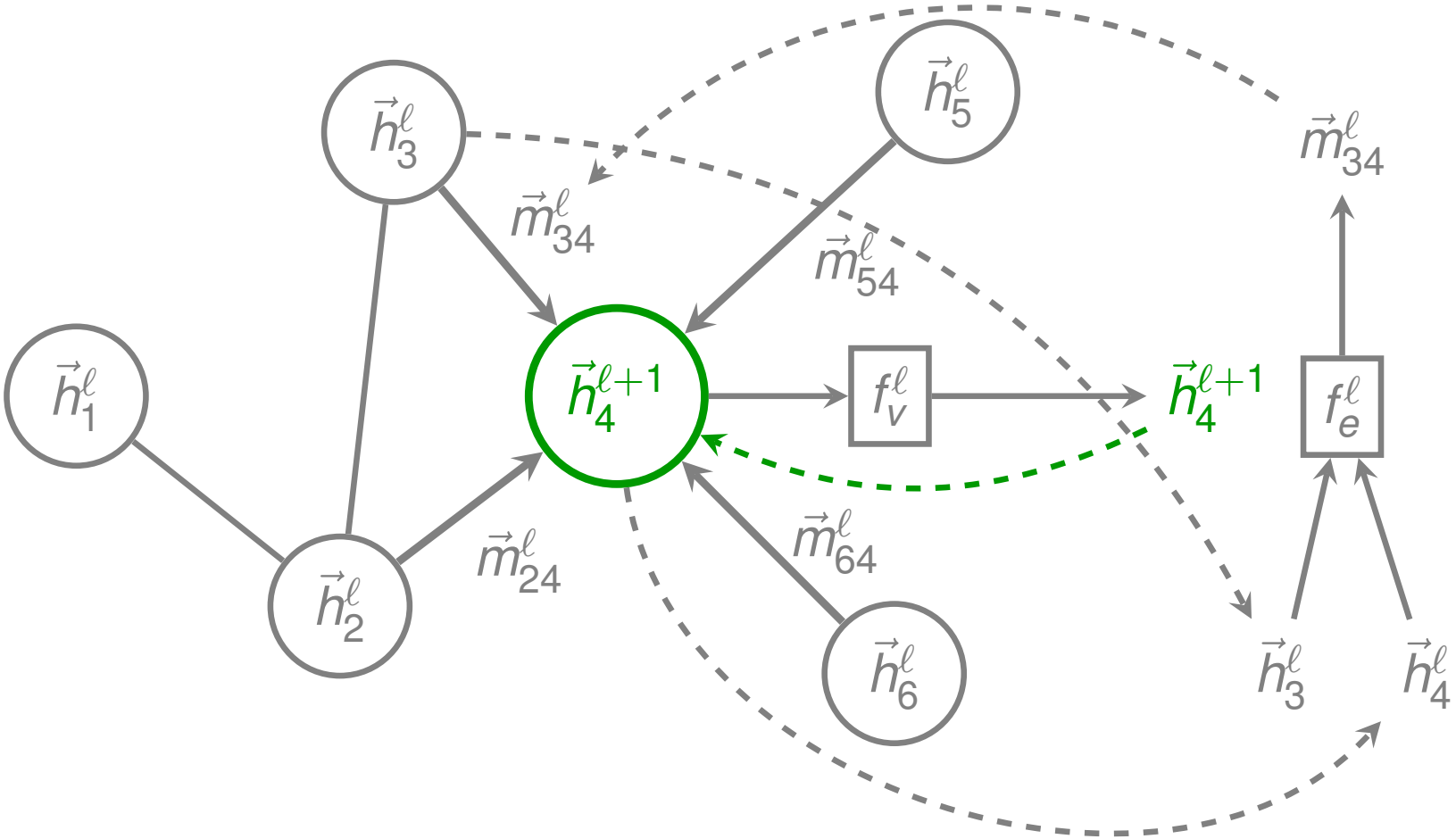
MPNN, aggregating messages



MPNN, computing node features



MPNN, next-level features



Towards a “golden middle”

- The MPNN model is the *most potent* graph convolutional layer, but requires storage and manipulation of *edge messages*, which quickly becomes troublesome memory-wise.
  - In practice, only applicable to *small graphs*.
  - Can think of them as “MLPs” of the graph domain.
- As an intermediate approach, let’s consider a more general form of the GCN update rule:

$$\vec{h}'_i = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} \vec{h}_j\right)$$

The GCN’s shortcomings lied in making α_{ij} **explicit** (based on graph structure).

GAT (Veličković *et al.*, ICLR 2018)

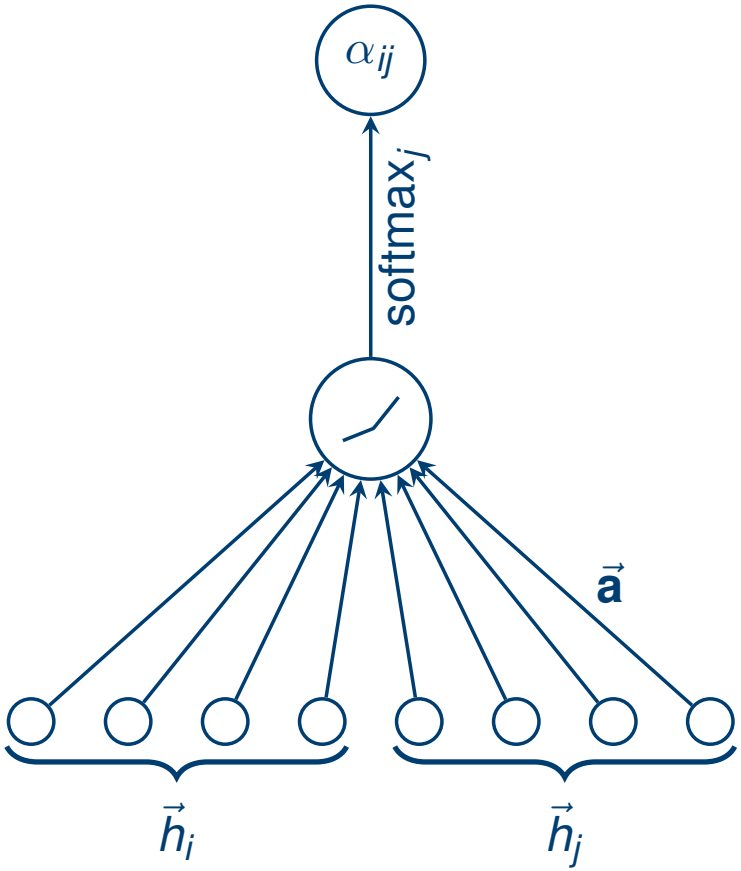
- Instead, if we let α_{ij} be computed *implicitly*...

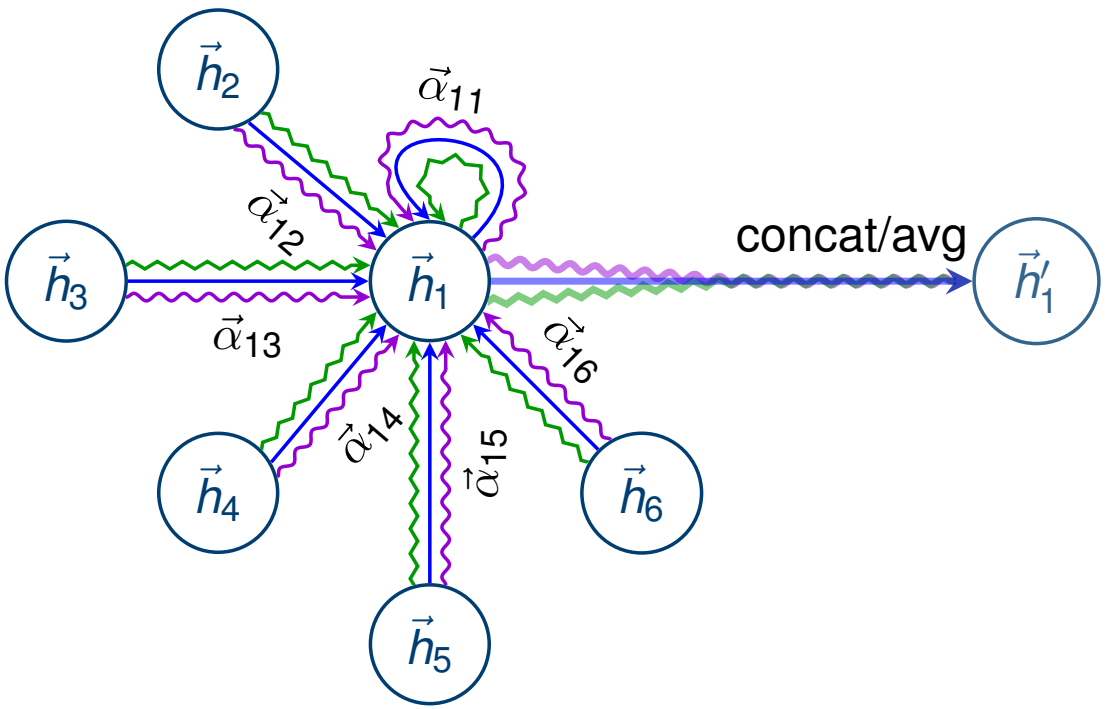
$$e_{ij} = a(\vec{h}_i, \vec{h}_j, \vec{e}_{ij})$$
$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}$$

  ...where $a : \mathbb{R}^N \times \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}$ is a learnable, shared, *self-attention mechanism* (e.g. Transformer attention)...

- ...we arrive at the *graph attention network* (**GAT**) update rule!
  - In practice, significantly stablised through *multi-head attention*.
  - Probably not as general as MPNNs, but trivially scalable.

A single GAT step, visualised





Roadmap for today

Graph convolutional networks

- GCN (Kipf & Welling, ICLR 2017)
- MPNN (Gilmer *et al.*, ICML 2017)
- GAT (Veličković *et al.*, ICLR 2018)

Generative models of graphs

- **MolGAN (De Cao & Kipf, ICML TADGM 2018)**
- **GCPN (You *et al.*, NIPS 2018)**

Semi-supervised adversarial learning on graphs

- GraphSGAN (Ding *et al.*, CIKM 2018)

Graph-based adversarial defence

- PeerNets (Svoboda *et al.*, 2018)

Generative models of graphs

- We seek a model capable of producing graphs that “capture” the empirical properties of a given distribution of graphs (usually via a training set).
- Very challenging! Thus far majority of results in the domain of *small chemicals* ($<$ 50 nodes!).
- Still, even in this domain, immense potential for important tasks, such as **drug discovery**!

Issues with generative modelling of graphs

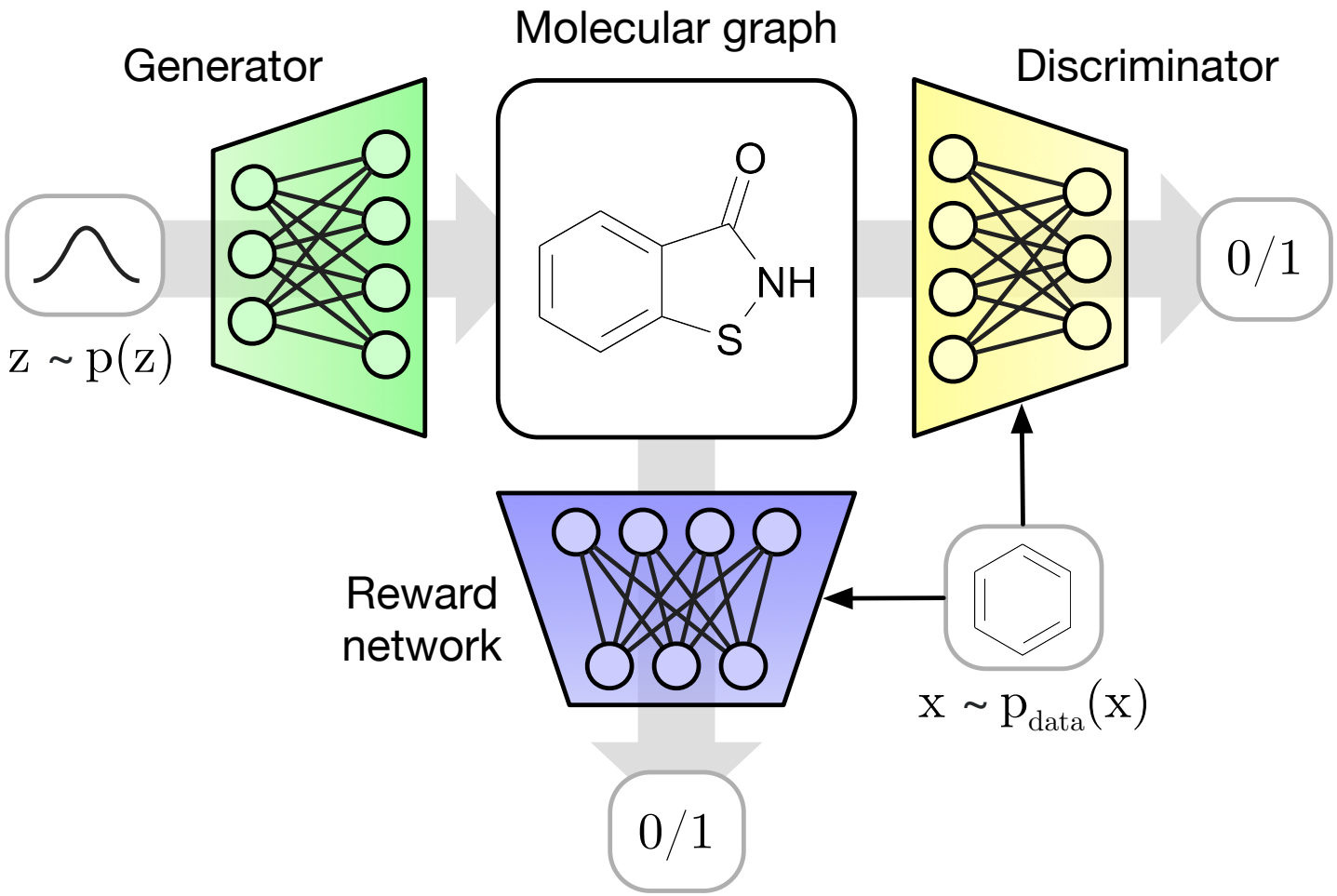
- We have the option of directly generating graphs from a given noise input, $\vec{z}$, either:
  - Sequentially (GraphRNN; You *et al.*, ICML 2018); or
  - All-at-once (GraphVAE; Simonovsky *et al.*, 2018)

  using a likelihood-based objective...
- ...but this is troublesome, as it requires either a *fixed node ordering* or (expensive) *graph matching*!
- *GANs to the rescue!*

GANs to the rescue!

- **Implicit models** such as GANs alleviate the previous issues.
- However, generating a graph involves *discrete choices* (e.g. choosing edges)—as such, cannot directly backpropagate through the generator.
- Requires *reinforcement learning* or the *Gumbel softmax* in practice.
  - RL can also be beneficial to, e.g., enforce certain chemical properties through reward signals.
- In this space, we may once again generate graphs all-at-once, or sequentially. I will now present one example of each—both concurrently published.

MolGAN (De Cao & Kipf, ICML TADGM 2018)



MolGAN: Generator

- The authors upfront limit the number of atoms in the generated molecule to 9 (to match their dataset).
- This allows generating the *node annotation matrix* $\mathbf{X} \in \mathbb{R}^{N \times T}$ (probability of each of T atom types) and *adjacency tensor* $\mathbf{A} \in \mathbb{R}^{N \times N \times K}$ (probability of each of K bond types), all-at-once.
- That is, the MolGAN generator, G_θ, computes $\mathbf{X}$ and $\mathbf{A}$ through an MLP applied to $\vec{z} \sim \mathcal{N}(\vec{0}, \mathbf{I})$.
- Sample one-hot versions ($\tilde{\mathbf{X}}$ and $\tilde{\mathbf{A}}$) using the *Gumbel softmax* trick, to allow backpropagating through the generator.

MolGAN: Discriminator/Reward network

- The discriminator, D_ϕ, and reward network, R_ψ, receive as input a graph $(\tilde{\mathbf{X}}, \tilde{\mathbf{A}})$, either real or generated.

- Compute several GCN-style updates over the nodes:

$$\vec{h}'_i = \tanh\left(f_s(\vec{h}_i, \vec{\tilde{x}}_i) + \sum_{j=1}^{N} \sum_{k=1}^{K} \frac{\tilde{\mathbf{A}}_{ijk}}{|\mathcal{N}_i|} f_k(\vec{h}_j, \vec{\tilde{x}}_i) \right)$$

  where f_s and f_k are MLPs (one per each bond type!).

MolGAN: Discriminator/Reward network, *cont'd*

- Then, summarise the obtained features, $\vec{h}_i$, into a graph representation, $\vec{h}_G$, as follows (Li *et al.*, ICLR 2016):

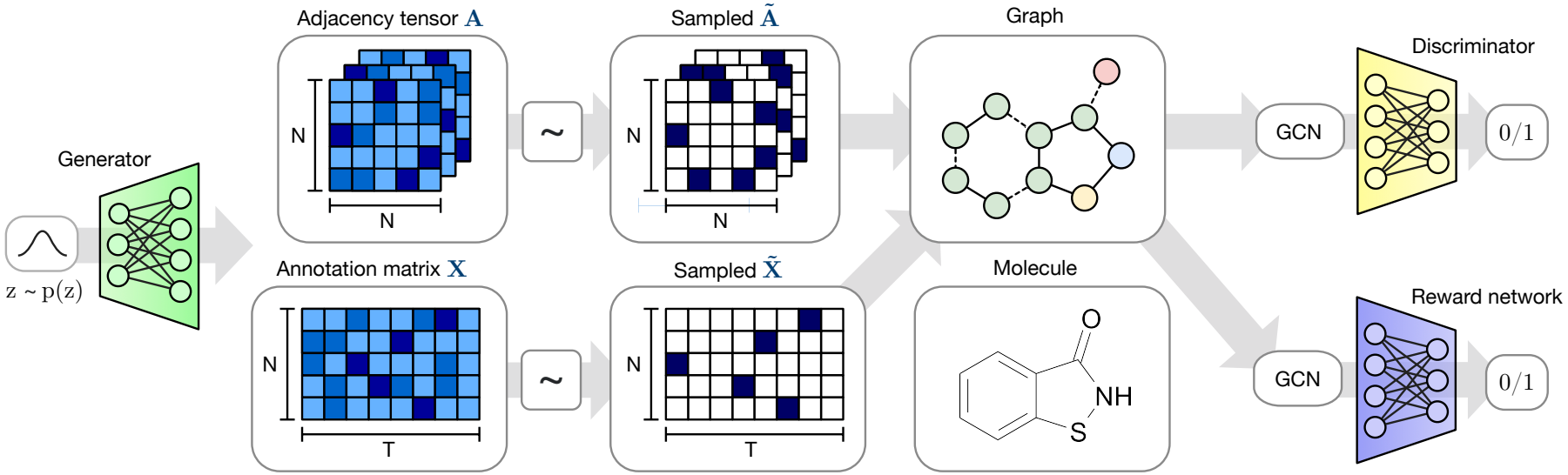
$$\vec{h}_G = \tanh\left(\sum_i \sigma(a(\vec{h}_i, \vec{\tilde{x}}_i)) \odot \tanh(b(\vec{h}_i, \vec{\tilde{x}}_i))\right)$$

  where a and b are MLPs, and σ is the logistic sigmoid.

- Finally, $\vec{h}_G$ is used as input to the discriminator/reward MLPs (for obtaining the relevant scores).

- The discriminator and reward networks are entirely disjoint!

MolGAN, *expanded*



MolGAN: Training

- The reward network is optimised to match a chemical property score which is output by an external software.
  - Specially, the reward is zero for *invalid* molecules.

- The discriminator, D_ϕ, is optimised using the *Improved WGAN* (Gulrajani *et al.*, 2017) objective:

$$\mathcal{L}(\vec{x}_i, G_\theta; \phi) = -D_\phi(\vec{x}_i) + D_\phi(G_\theta(\vec{z}_i)) + \alpha \left(\|\nabla_{\vec{\hat{x}}_i} D_\phi(\vec{\hat{x}}_i)\| - 1 \right)^2$$

where $\vec{x}_i \sim p_{data}(\vec{x})$, $\vec{z}_i \sim p_{\vec{z}}(\vec{z})$,
and $\vec{\hat{x}}_i = \varepsilon \vec{x}_i + (1 - \varepsilon) G_\theta(\vec{z}_i)$ $(\varepsilon \sim U(0, 1))$.

MolGAN: Generator training

- Since we made a continuous approximation to the sampling operation (i.e. Gumbel softmax), we can directly propagate gradients through the generator!
- The generator, G_θ, is trained using a combination of two objectives:
  - the original WGAN objective (Arjovsky *et al.*, 2017); and
  - the policy gradients obtained from DDPG (Lillicrap *et al.*, 2015), using the reward function to obtain rewards.

$$\mathcal{L}(\theta) = \lambda \mathcal{L}_{WGAN} + (1 - \lambda)\mathcal{L}_{DDPG}$$

- For the purposes of the RL setup, the generator is the policy network, and the actions correspond to the generated graphs.

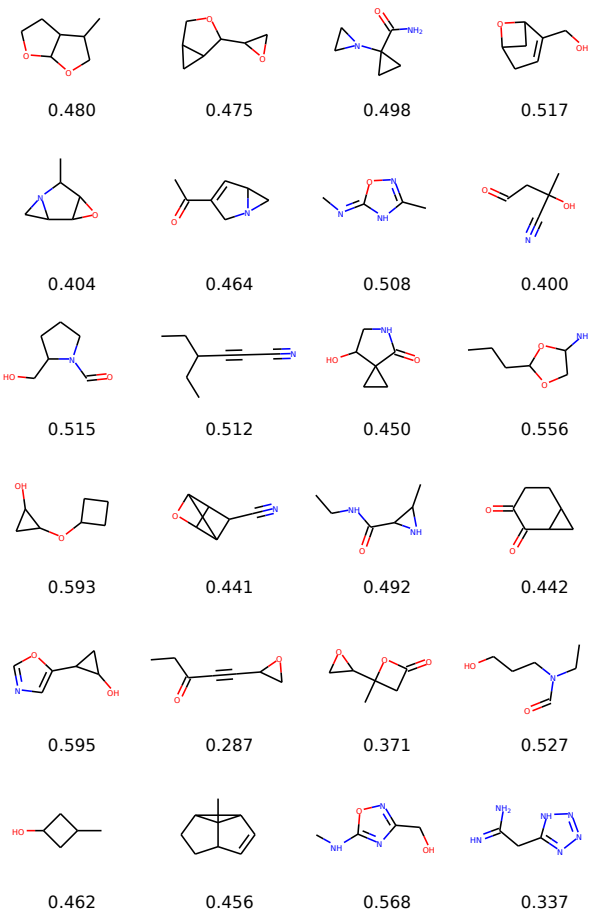
Results: Quantitative

Comparative evaluation against ORGAN (Guimaraes *et al.*, 2017).

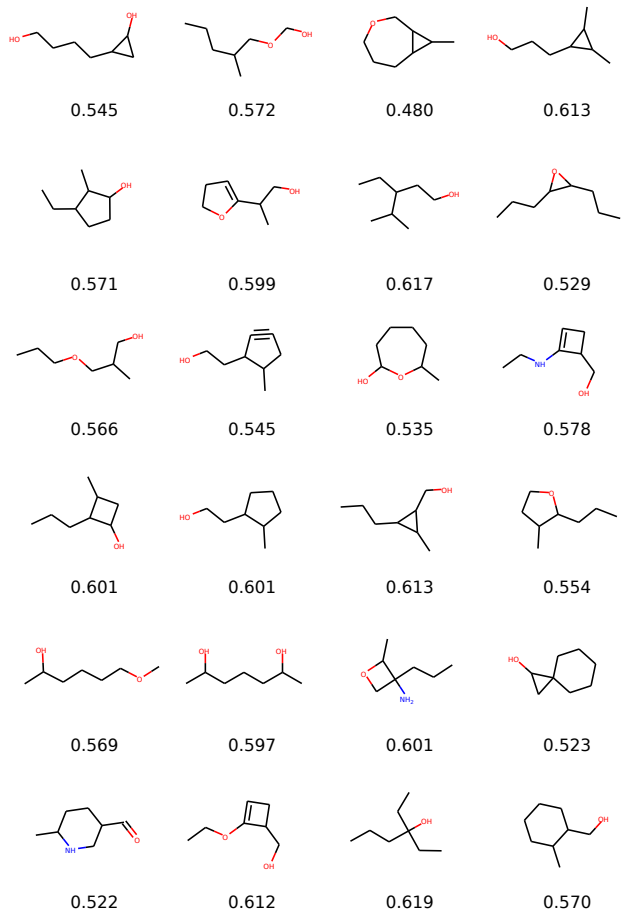
Objective	Algorithm	Valid (%)	Unique (%)	Time (h)	Diversity	Druglikeliness	Synthesizability	Solubility
Druglikeliness	ORGAN	88.2	69.4*	9.63*	0.55	0.52	0.32	0.35
	OR(W)GAN	85.0	8.2*	10.06*	0.95	0.60	0.54	0.47
	Naive RL	97.1	54.0*	9.39*	0.80	0.57	0.53	0.50
	MolGAN	**99.9**	2.0	1.66	0.95	**0.61**	0.68	0.52
	MolGAN (QM9)	**100.0**	2.2	4.12	**0.97**	**0.62**	0.59	0.53
Synthesizability	ORGAN	96.5	45.9*	8.66*	0.92	0.51	0.83	0.45
	OR(W)GAN	97.6	30.7*	9.60*	**1.00**	0.20	0.75	0.84
	Naive RL	97.7	13.6*	10.60*	0.96	0.52	0.83	0.46
	MolGAN	**99.4**	2.1	1.04	0.75	0.52	**0.90**	0.67
	MolGAN (QM9)	**100.0**	2.1	2.49	0.95	0.53	**0.95**	0.68
Solubility	ORGAN	94.7	54.3*	8.65*	0.76	0.50	0.63	0.55
	OR(W)GAN	94.1	20.8*	9.21*	0.90	0.42	0.66	0.54
	Naive RL	92.7	100.0*	10.51*	0.75	0.49	0.70	0.78
	MolGAN	**99.8**	2.3	0.58	**0.97**	0.45	0.42	**0.86**
	MolGAN (QM9)	**99.8**	2.0	1.62	**0.99**	0.44	0.22	**0.89**
All/Alternated	ORGAN	96.1	97.2*	10.2*	0.92	**0.52**	0.71	0.53
All/Simultaneously	*MolGAN*	**97.4**	2.4	2.12	0.91	0.47	**0.84**	**0.65**
All/Simultaneously	*MolGAN (QM9)*	**98.0**	2.3	5.83	**0.93**	0.51	**0.82**	**0.69**

Trains *much faster*, but suffers from **mode collapse**!

Results: Qualitative



Dataset (QM9)

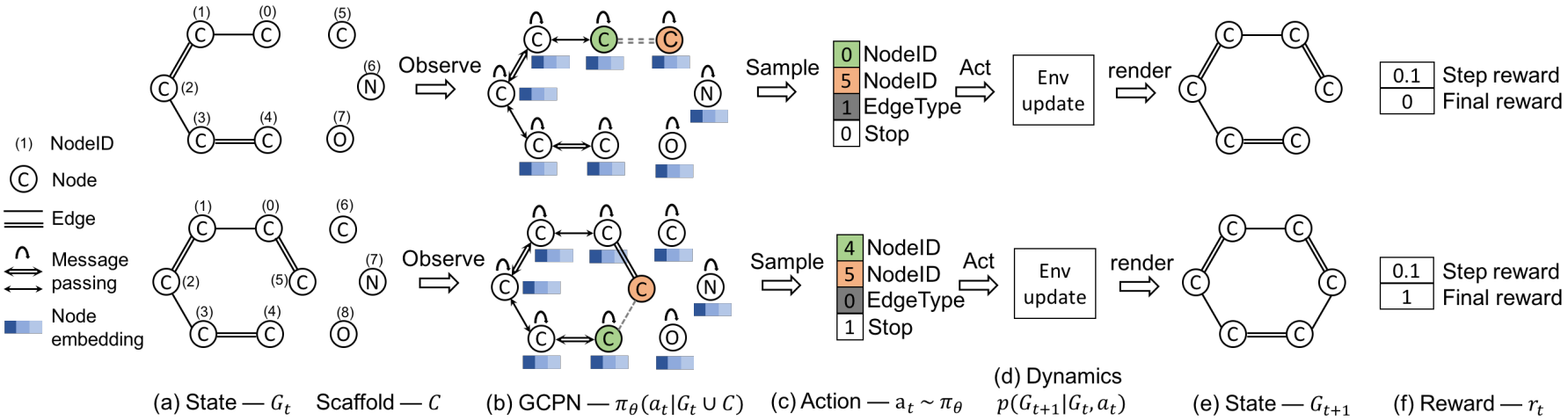


MolGAN

GCPN (You *et al.*, NIPS 2018)

- The *graph convolutional policy network* (**GCPN**) represents, conversely, a *sequential* generation method.
- At each step, the generator takes a graph of a *partially constructed molecule* ($\mathcal{G}_t$) and a set of *scaffolds* ($\mathcal{C}_t$), and makes discrete decisions on:
  - **Two atoms** to connect (one must be in $\mathcal{G}_t$);
  - **Bond type** between them;
  - Whether or not to **stop**.
- Partial molecules are checked by chemical software for validity and properties; invalid molecules give negative reward and nullify the action! (N.B. no explicit reward network this time.)

GCPN



Total reward given to the generator is $\mathcal{R}_{chem} - V(G_\theta, D_\phi)$, where $V(G_\theta, D_\phi)$ is the usual GAN value function (Goodfellow *et al.*, 2014):

$$V(G_\theta, D_\phi) = \mathbb{E}_{\vec{x}\sim p_{data}}[\log D_\phi(\vec{x})] + \mathbb{E}_{\vec{x}\sim G_\theta}[\log(1 - D_\phi(\vec{x}))]$$

Policy gradients obtained using PPO (Schulman *et al.*, 2017).

GCPN embedding computation

- Both the generator and discriminator compute node embeddings (from either a (partially-)generated chemical or a real example) based on a GCN update rule:

$$\mathbf{H}' = \sum_{k=1}^{K} ReLU\left(\tilde{\mathbf{D}}_k^{-\frac{1}{2}} \tilde{\mathbf{A}}_k \tilde{\mathbf{D}}_k^{-\frac{1}{2}} \mathbf{H}\mathbf{W}_k\right)$$

  decoupling each of the K bond types.
- These can be averaged into a *graph embedding*:

$$\vec{h}_{\mathcal{G}} = \frac{1}{N} \sum_{i=1}^{N} \vec{h}_i$$

GCPN: Policy and discriminator outputs

- Each action consists of four choices: nodes i and j, bond type k, and stopping decision t. Obtained using appropriate MLPs, $m_{first}, m_{second}, m_{edge} and m_{stop}$:

$$
\begin{aligned}
i &\sim softmax(m_{first}(\mathbf{H}^{(\mathcal{G}_t)})) \\
j &\sim softmax(m_{second}(\vec{h}_i, \mathbf{H}^{(\mathcal{G}_t \cup \mathcal{C})})) \\
k &\sim softmax(m_{edge}(\vec{h}_i, \vec{h}_j)) \\
t &\sim softmax(m_{stop}(\vec{h}_{\mathcal{G}_t})) \\
a_{t+1} &= [i, j, k, t]
\end{aligned}
$$

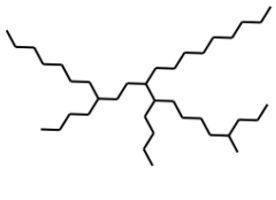
- The discriminator produces a score based on the output of an MLP applied to $\vec{h}_{\mathcal{G}}$.

GCPN training

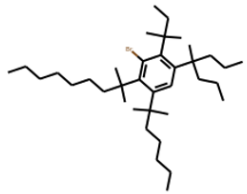
- The discriminator network is trained by SGD on the original GAN loss.
- The generator network is first pre-trained in an MLE fashion by using (partially-constructed) real molecules to obtain ground-truth actions.
- Afterwards, use PPO on the obtained chemical score and discriminator output to train via policy gradients.

Results: Property optimisation

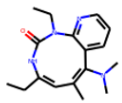
Method	Penalized logP				QED			
	1st	2nd	3rd	Validity	1st	2nd	3rd	Validity
ZINC	4.52	4.30	4.23	100.0%	0.948	0.948	0.948	100.0%
ORGAN	3.63	3.49	3.44	0.4%	0.896	0.824	0.820	2.2%
JT-VAE	5.30	4.93	4.49	100.0%	0.925	0.911	0.910	100.0%
GCPN	**7.98**	**7.85**	**7.80**	**100.0%**	**0.948**	**0.947**	**0.946**	**100.0%**



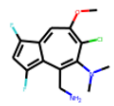
7.98



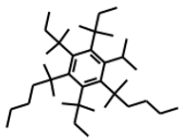
7.48



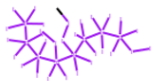
0.948



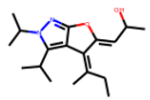
0.945



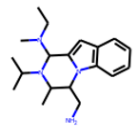
7.12



23.88*



0.944



0.941

(a) Penalized logP optimization

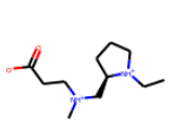
(b) QED optimization

Results: Property targeting

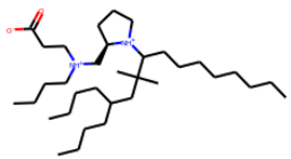
Method	$-2.5 \leq \text{logP} \leq -2$		$5 \leq \text{logP} \leq 5.5$		$150 \leq \text{MW} \leq 200$		$500 \leq \text{MW} \leq 550$	
	Success	Diversity	Success	Diversity	Success	Diversity	Success	Diversity
ZINC	0.3%	0.919	1.3%	0.909	1.7%	0.938	0	–
JT-VAE	11.3%	**0.846**	7.6%	0.907	0.7%	0.824	16.0%	0.898
ORGAN	0	–	0.2%	**0.909**	15.1%	0.759	0.1%	0.907
GCPN	**85.5%**	0.392	**54.7%**	0.855	**76.1%**	**0.921**	**74.1%**	**0.920**

Results: Constrained optimisation

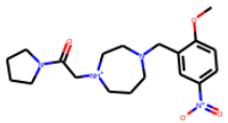
δ	JT-VAE			GCPN		
	Improvement	Similarity	Success	Improvement	Similarity	Success
0.0	1.91 ± 2.04	0.28 ± 0.15	97.5%	$\mathbf{4.20 \pm 1.28}$	$\mathbf{0.32 \pm 0.12}$	**100.0%**
0.2	1.68 ± 1.85	0.33 ± 0.13	97.1%	$\mathbf{4.12 \pm 1.19}$	$\mathbf{0.34 \pm 0.11}$	**100.0%**
0.4	0.84 ± 1.45	$\mathbf{0.51 \pm 0.10}$	83.6%	$\mathbf{2.49 \pm 1.30}$	0.47 ± 0.08	**100.0%**
0.6	0.21 ± 0.71	0.69 ± 0.06	46.4%	$\mathbf{0.79 \pm 0.63}$	$\mathbf{0.68 \pm 0.08}$	**100.0%**



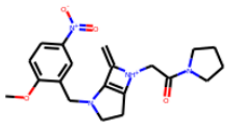
-8.32



-0.71



-5.55



-1.78

(c) Constrained optimization of penalized logP

Conclusions: Graph generative models

- Despite promising results, still a lot of work left to be done!
  - Enforcing validity without resorting to “hard stopping” is one such property in the chemical domain.
  - Scaling up to larger graphs is also paramount!

- Most of the push thus far has been by graph neural network researchers—in my opinion, far more input (and likely bespoke objectives for graphs) is needed from GAN experts.

Roadmap for today

Graph convolutional networks

- GCN (Kipf & Welling, ICLR 2017)
- MPNN (Gilmer *et al.*, ICML 2017)
- GAT (Veličković *et al.*, ICLR 2018)

Generative models of graphs

- MolGAN (De Cao & Kipf, ICML TADGM 2018)
- GCPN (You *et al.*, NIPS 2018)

Semi-supervised adversarial learning on graphs

- **GraphSGAN (Ding *et al.*, CIKM 2018)**

Graph-based adversarial defence

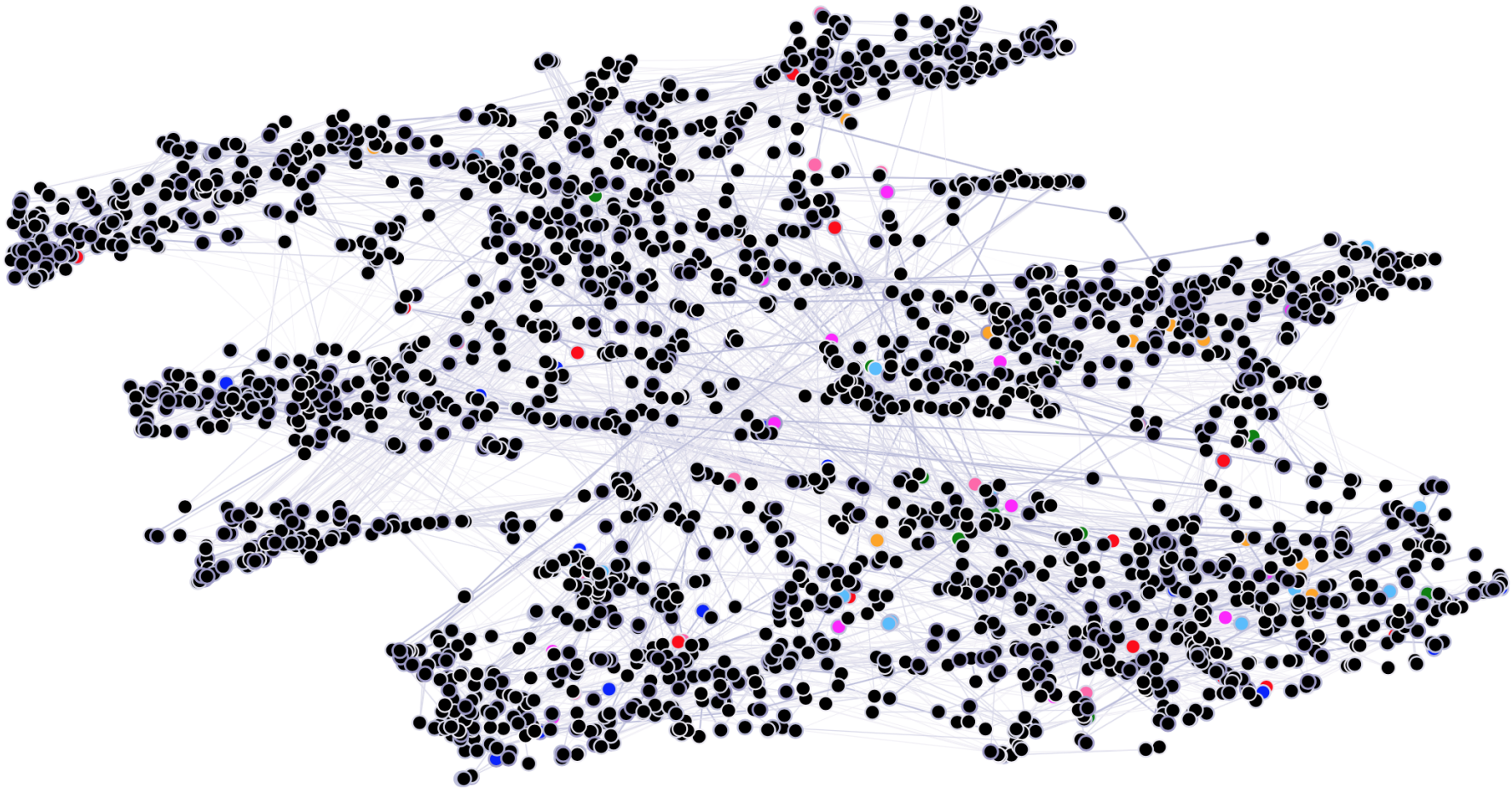
- PeerNets (Svoboda *et al.*, 2018)

Node classification

- We will now focus on the **node classification** problem:
  - **Input**: a matrix of *node features*, $\mathbf{X} \in \mathbb{R}^{N \times F}$, with F features in each of the N nodes, and an *adjacency matrix*, $\mathbf{A} \in \mathbb{R}^{N \times N}$.
  - **Output**: a matrix of *node class probabilities*, $\mathbf{Y} \in \mathbb{R}^{N \times C}$, such that $Y_{ij} = \mathbb{P}(\textit{Node } i \in \textit{Class } j)$.

- Can be seen as *traditional deep learning*, with *relations between training set elements*.

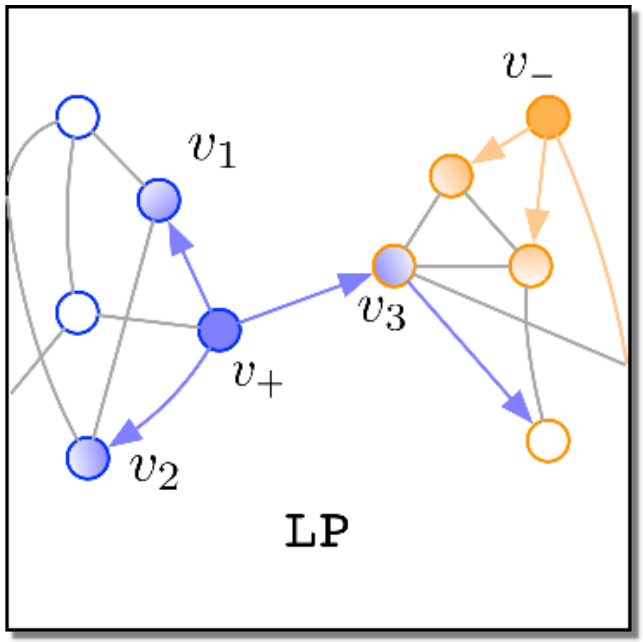
Transductive learning on graphs



Training algorithm sees *all features* (**including test nodes**)!

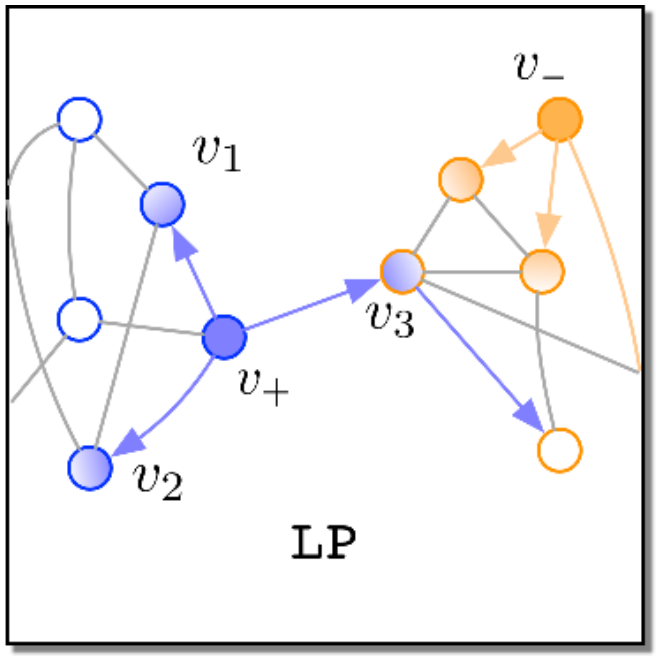
Towards a generative solution

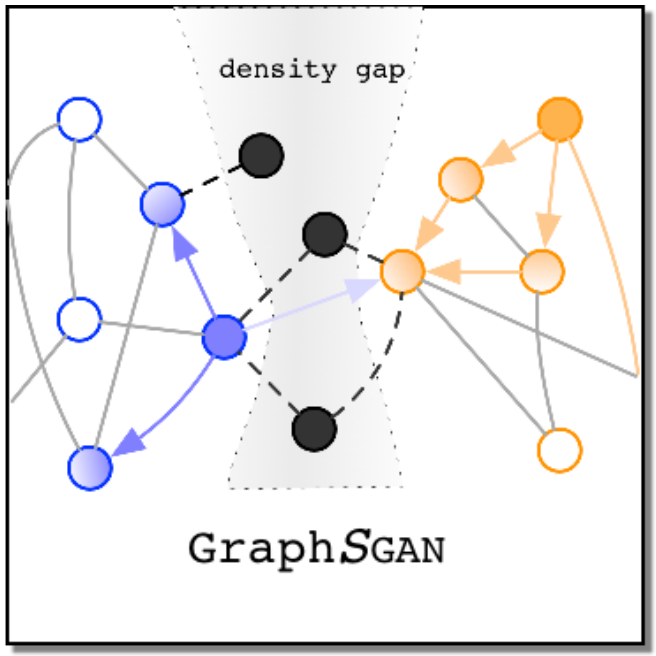
- The predictions of the model may be highly dependent on *where the labelled nodes are in the graph*!
- Consider this failure case of simple label propagation:



The algorithm can be very sensitive to *bridges*!

GraphSGAN (Ding *et al.*, CIKM 2018)





- Generate *artificial nodes* to fill the *density gap*!
- **Problem:** *How to **connect** the new nodes?*
- To solve this, we will need to make a brief detour. . .

Inserting structure: *DeepWalk*

- An alternative to using a graph convolutional network is first learning some **structural features**, $\vec{\Phi}_i$, for each node i (these will not depend on $\vec{x}_i$, but on the graph structure)!
- Then, use $\vec{h}_i = \vec{x}_i \| \vec{\Phi}_i$ as the input to a shared classifier (where $\|$ is concatenation).
- Typically, **random walks** are used as the primary input for analysing the structural information of each node.
- The first method to leverage random walks efficiently is *DeepWalk* by Perozzi *et al.* (KDD 2014)

Overview of DeepWalk

- Start by random features $\vec{\Phi}_i$ for each node i.
- Sample a random walk $\mathcal{W}_i$, starting from node i.
- For node x at step j, $x = \mathcal{W}_i[j]$, and a node y at step $k \in [j - w, j + w]$, $y = \mathcal{W}_i[k]$, modify $\vec{\Phi}_x$ to maximise $\log \mathbb{P}(y|\vec{\Phi}_x)$ (obtained from a neural network classifier).
- Inspired by **skip-gram models** in natural language processing: to obtain a good vector representation of a word, its vector should allow us to easily predict the words that *surround* it.

Overview of DeepWalk, *cont'd*

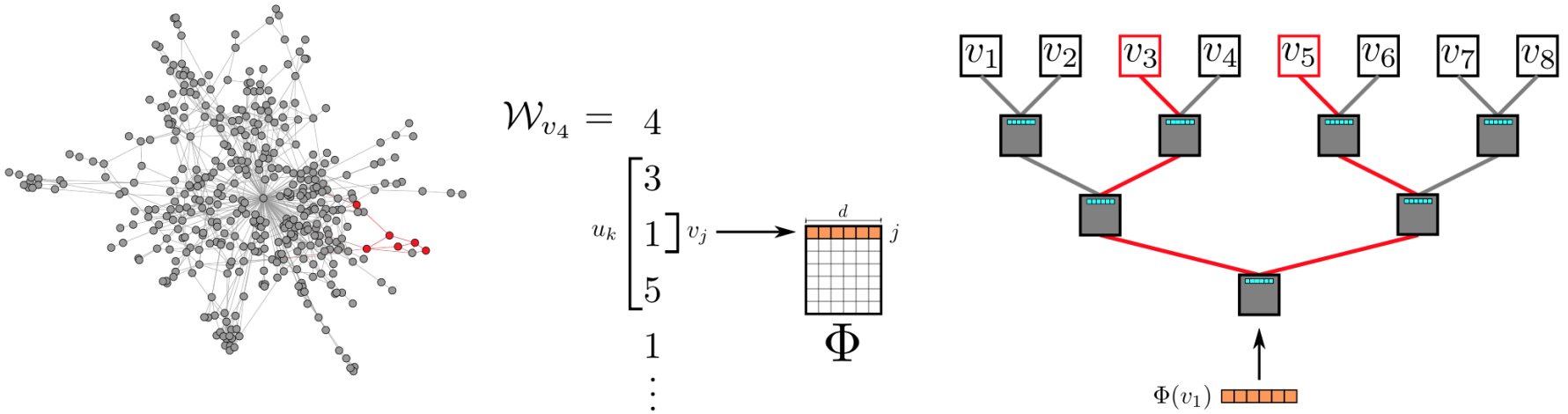
- Expressing the full $\mathbb{P}(y|\vec{\Phi}_x)$ distribution directly, even for a single layer neural network, where

$$\mathbb{P}(y|\vec{\Phi}_x) = softmax(\vec{w}_y^T \vec{\Phi}_x) = \frac{\exp\left(\vec{w}_y^T \vec{\Phi}_x\right)}{\sum_z \exp\left(\vec{w}_z^T \vec{\Phi}_x\right)}$$

  is prohibitive for large graphs, as we need to normalise across the entire space of nodes—making most updates *vanish*.

- To rectify, DeepWalk expresses it as a *hierarchical softmax*—a tree of binary classifiers, each halving the node space.

DeepWalk in action



(a) Random walk generation. (b) Representation mapping. (c) Hierarchical Softmax.

Figure 3: Overview of DEEPWALK. We slide a window of length $2w + 1$ over the random walk $\mathcal{W}_{v_4}$, mapping the central vertex v_1 to its representation $\Phi(v_1)$. Hierarchical Softmax factors out $\Pr(v_3 \mid \Phi(v_1))$ and $\Pr(v_5 \mid \Phi(v_1))$ over sequences of probability distributions corresponding to the paths starting at the root and ending at v_3 and v_5. The representation Φ is updated to maximize the probability of v_1 co-occurring with its context $\{v_3, v_5\}$.

Later improved by *LINE* (Tang *et al.*, WWW 2015) and *node2vec* (Grover & Leskovec, KDD 2016), but main idea stays the same. GraphSGAN uses **LINE**.

DeepWalk meets GANs

- LINE features can be precomputed for every node in the original graph and attached to their main features.
- GraphSGAN avoids the issue of having to connect new nodes by *generating both node features **and** LINE-like features*!
- That is, its generator, G_θ, produces node features $\vec{\hat{h}}_i$ as follows:

$$\vec{z} \sim p(\vec{z})$$
$$\vec{\hat{h}}_i = G_\theta(\vec{z})$$

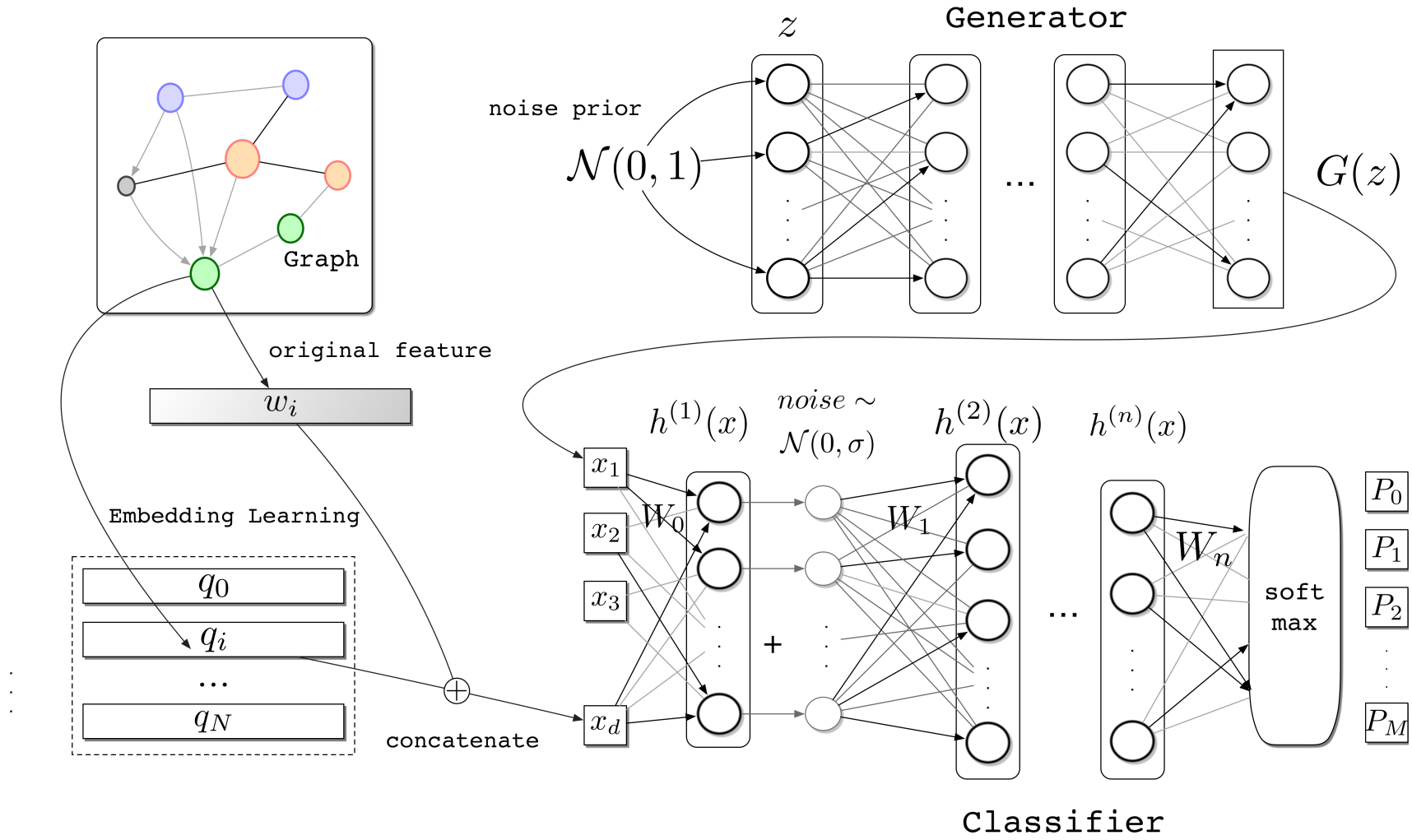
where $\vec{\hat{h}}_i = \vec{\hat{x}}_i \| \vec{\hat{\Phi}}_i$, so the generator *encodes the local graph structure of i without having to explicitly generate it*!

GraphSGAN generator/discriminator

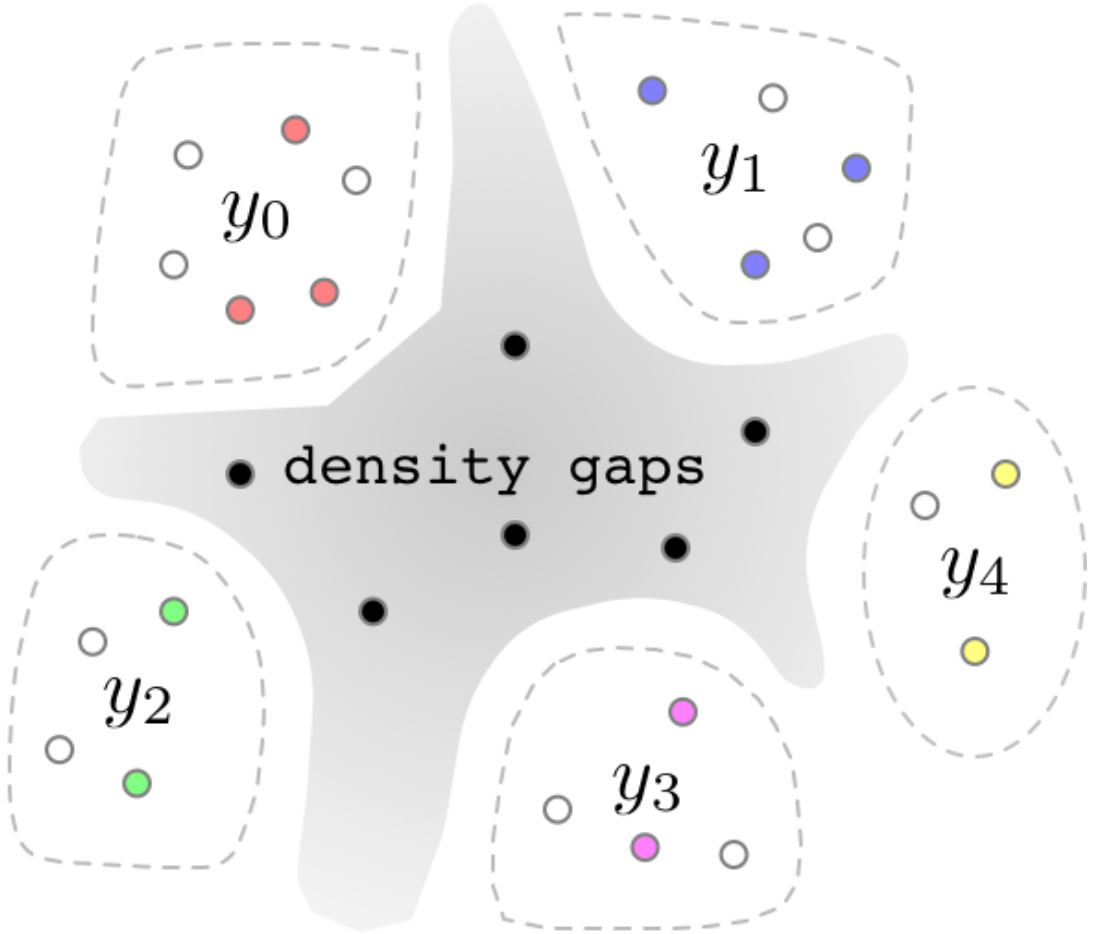
- As all nodes are represented as flat feature vectors (which encodes the graph structure), we do not need to use graph convolutional networks!

- Indeed, GraphSGAN uses MLPs for G_θ and D_ϕ. The discriminator shares all of its parameters with the downstream node classifier (by adding an extra class, P_M, for "fake")!

Overview of GraphSGAN



Expected outcome of training



Choose objectives to enforce such behaviour!

Objectives on the discriminator

- Map differently-classed nodes to different clusters.
  - Naturally optimised using the standard *cross-entropy* loss on the labelled nodes:

$$\mathcal{L}_{CE} = -\mathbb{E}_{\vec{h}_i \sim p_{data}} \log \mathbb{P}(y_i | \vec{h}_i, y_i < M)$$

- Real nodes should **not** be mapped into the "central" area (to expose the gaps).
  - Assuming the generator *perfectly* generates samples in the density gap, then the original GAN loss will enforce this:

$$\mathcal{L}_{GAN} = -\mathbb{E}_{\vec{h}_i \sim p_{data}} \log(1 - \mathbb{P}(M | \vec{h}_i)) - \mathbb{E}_{\vec{\hat{h}}_i \sim G(\vec{z})} \log \mathbb{P}(M | \vec{\hat{h}}_i)$$

Objectives on the discriminator, *cont'd*

- Each unlabelled node should be mapped into one cluster.
  - Enforced by requiring a "confident" classifier; i.e. penalising the *entropy* over the non-fake classes:

$$\mathcal{L}_{ent} = -\mathbb{E}_{\vec{h}_i \sim p_{data}} \sum_{y=0}^{M-1} \mathbb{P}(y|\vec{h}_i, y_i < M) \log \mathbb{P}(y|\vec{h}_i, y_i < M)$$

- Clusters should be as distant from each other as possible.
  - Add a *pull-away* penalty within a batch of m examples (as used in EBGAN; Zhao *et al.*, 2017) to enforce this:

$$\mathcal{L}_{pull} = \frac{1}{m(m-1)} \sum_{i=1}^{m} \sum_{j \neq i} \left(\frac{\vec{h}_i^T \vec{h}_j}{\|\vec{h}_i\| \|\vec{h}_j\|} \right)^2$$

GraphSGAN discriminator objective

- Finally, the GraphSGAN discriminator/classifier is optimised with respect to all four of these losses simultaneously:

$$\mathcal{L}_D = \mathcal{L}_{CE} + \lambda_0 \mathcal{L}_{GAN} + \lambda_1 \mathcal{L}_{ent} + \lambda_2 \mathcal{L}_{pull}$$

- We now turn our attention to the generator...

Objectives on the generator

- Map fake samples into the “central area”.
  - Enforce with a *feature matching* loss; make generated nodes not deviate too far from the **centroid** of real nodes:

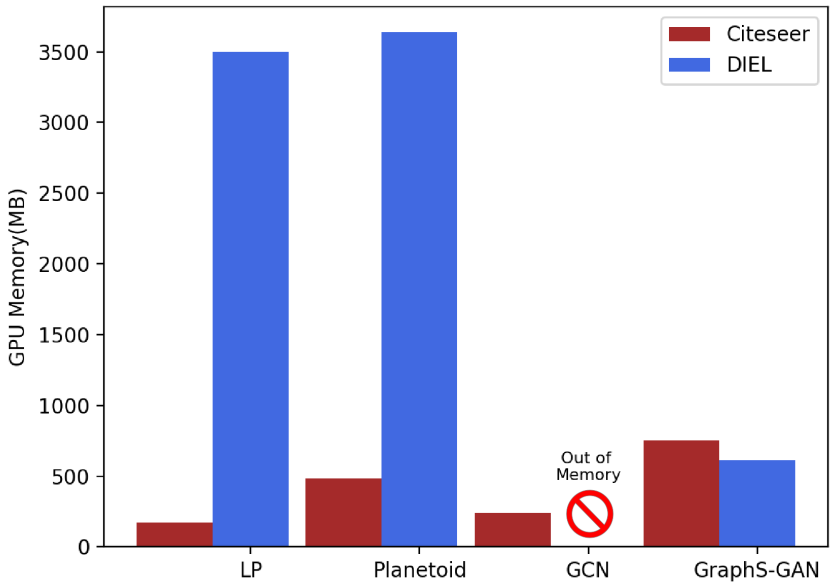
$$\mathcal{L}_{fm} = -\|\mathbb{E}_{\vec{h}_i \sim p_{data}} \vec{h}_i - \mathbb{E}_{\vec{\hat{h}}_j \sim G(\vec{z})} \vec{\hat{h}}_j\|^2$$

- Fake samples should not overfit to the centroid!
  - Use another pull-away term to enforce this.
- The generator is optimised using these two additional losses:

$$\mathcal{L}_G = \mathcal{L}_{fm} + \lambda \mathcal{L}_{pull}$$

Datasets used

Dataset	Nodes	Edges	Features	Classes	Labels
Cora	2,708	5,429	1,433	7	140
Citeseer	3,327	4,732	3,703	6	120
DIEL	4,373,008	4,464,261	1,233,597	4	3413.8



Results on Cora/Citeseer

Method	Cora	Citeseer
LP	68.0	45.3
ICA	75.1	69.1
ManiReg	59.5	60.1
DeepWalk	67.2	43.2
SemiEmb	59.0	59.6
Planetoid	75.7	64.7
Chebyshev	81.2	69.8
GCN	80.1 ± 0.5	67.9 ± 0.5
GAT	$\mathbf{83.0 \pm 0.7}$	72.5 ± 0.7
GraphSGAN	$\mathbf{83.0 \pm 1.3}$	$\mathbf{73.1 \pm 1.8}$

Results on DIEL

Method	Recall@K
LP	16.2
ManiReg	47.7
DeepWalk	25.8
SemiEmb	48.6
Planetoid	50.1
DIEL	40.5
GraphSGAN	**51.8**
Upper bound	61.7

Conclusions: Semi-supervised adversarial learning

- GANs clearly have a lot of potential for improving performance of semi-supervised learning on graphs!
  - DeepWalk-like features discard a lot of *fine-grained information* about the structure; yet, performance exceeds that of methods that have access to the *entire adjacency matrix*.
- Clearly, future work should focus on *appropriately rewiring the generated nodes* rather than resorting to generating synthetic structural features.
- And, as before, more informed GAN objectives. . .

Roadmap for today

Graph convolutional networks

- GCN (Kipf & Welling, ICLR 2017)
- MPNN (Gilmer *et al.*, ICML 2017)
- GAT (Veličković *et al.*, ICLR 2018)

Generative models of graphs

- MolGAN (De Cao & Kipf, ICML TADGM 2018)
- GCPN (You *et al.*, NIPS 2018)

Semi-supervised adversarial learning on graphs

- GraphSGAN (Ding *et al.*, CIKM 2018)

Graph-based adversarial defence

- **PeerNets (Svoboda *et al.*, 2018)**

Graph-based adversarial defence

- Lastly, I will focus on how graph methods can be used to aid *defence against adversarial attacks*.
- We will consider the usual *image classification* task:
  - Deployed on standard image datasets (MNIST and CIFAR).
  - Using standard convolutional classifiers (LeNet and ResNet).
- Utilise adversarial attacks such as the gradient descent attack, FGSM (Goodfellow *et al.*, 2015) and universal adversarial perturbations (Moosavi-Dezfooli *et al.*, 2016) to attempt to fool the classifier.

Towards a “peer-knowledge” layer

- Adversarial attacks tend to fool the classifier by exposing it to examples from a statistical distribution it wasn’t trained on.

$\Longrightarrow$ *We can use “trusted” samples from the distribution to strengthen the classification!*

- This leads us to the peer regularisation (PR) layer, which can be trivially plugged into CNNs, and will strengthen the properties of the true data distribution in its feature maps.

The *peer regularisation* layer: Setup

- Assume we have a batch of N "peer" images, fed into a CNN. At an intermediate stage of the CNN, we will have feature maps $(\mathbf{H}_1, \ldots, \mathbf{H}_N)$; $\mathbf{H}_i \in \mathbb{R}^{h \times w \times d}$, for each one of them.
- Now, if we wish to classify a new image, we would feed it into the CNN as usual, and obtain its intermediate feature map, $\mathbf{H}$. For each pixel p of $\mathbf{H}$, let $\vec{h}_p \in \mathbb{R}^d$ denote its features.
- We locate the K nearest neighbours of $\vec{h}_p$, across all pixels of $\mathbf{H}_i$. Denote these as $\vec{h}_{q_1}, \vec{h}_{q_2}, \ldots, \vec{h}_{q_K}$.
  - **N.B.** Due to the structural biases of a CNN, these pixels actually encode similar *patches*!

The *peer regularisation* layer: Computation

- Now, compute the output pixel value at p as a **weighted average** over the nearest neighbours:

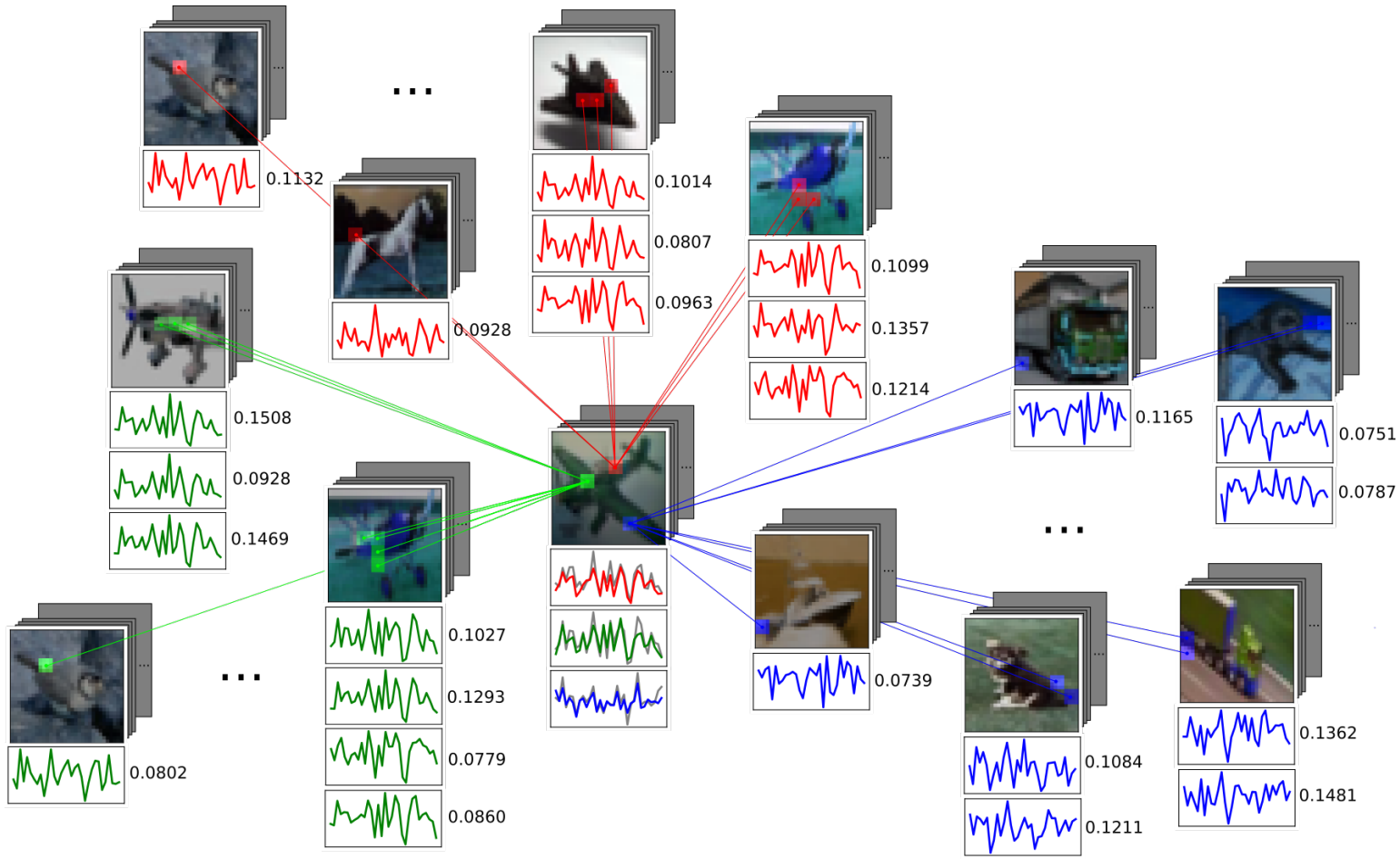
$$\vec{h}'_p = \sum_{k=1}^{K} \alpha_{ik} \vec{h}_{q_k}$$

- Here, the coefficients α_{ik} are computed as a byproduct of an *attention mechanism*—i.e. using the GAT update rule.

$$e_{ik} = a(\vec{h}_p, \vec{h}_{q_k})$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}$$

PeerNets (Svoboda *et al.*, 2018)

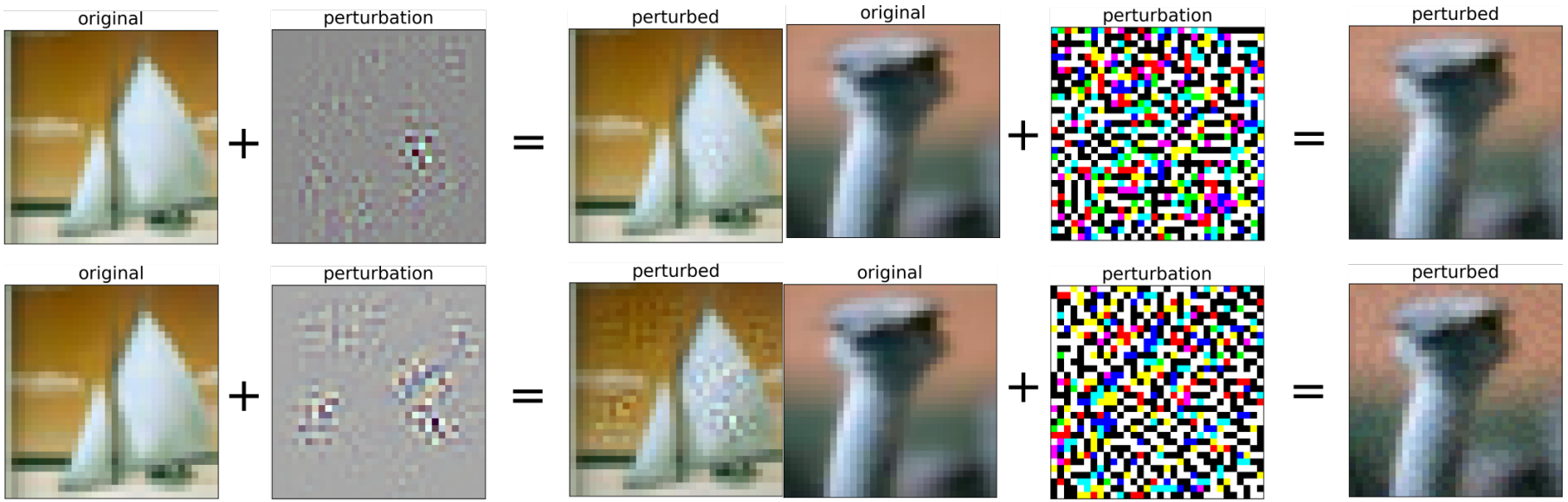


Results: Gradient descent attacks

- For the gradient descent attack method at $\epsilon = 0.1$:
  - **Targeted** (trying to perturb arbitrary CIFAR-10 test images into the “cat” class). Reduces fooling rate of ResNet-32 (from 16% to 3%), and increases the magnitude of the required attack (from 125.59 to 176.83).
  - **Non-targeted** (trying to perturb into any other class). Similarly, reduces ResNet-32 fooling rate from 59% to 22%, and magnitude of attack from 28.55 to 102.11.

- Analogous results observed for FGSM.

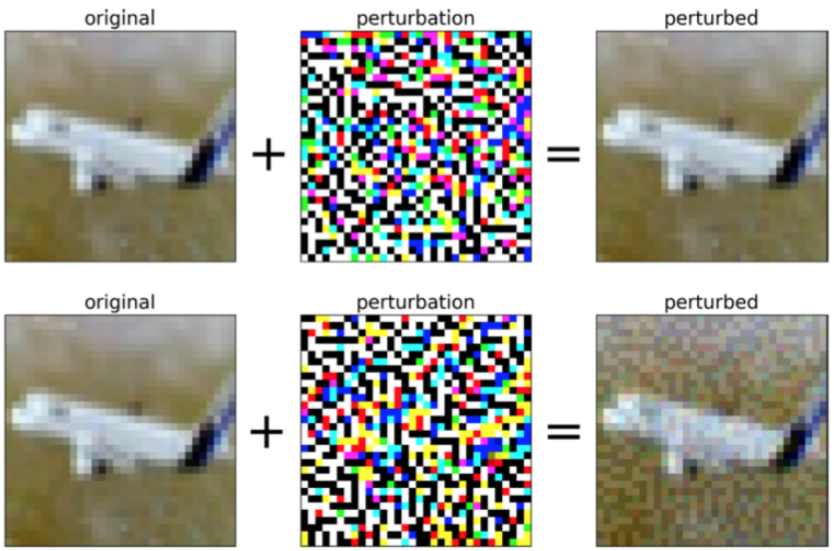
Gradient descent attacks: Samples



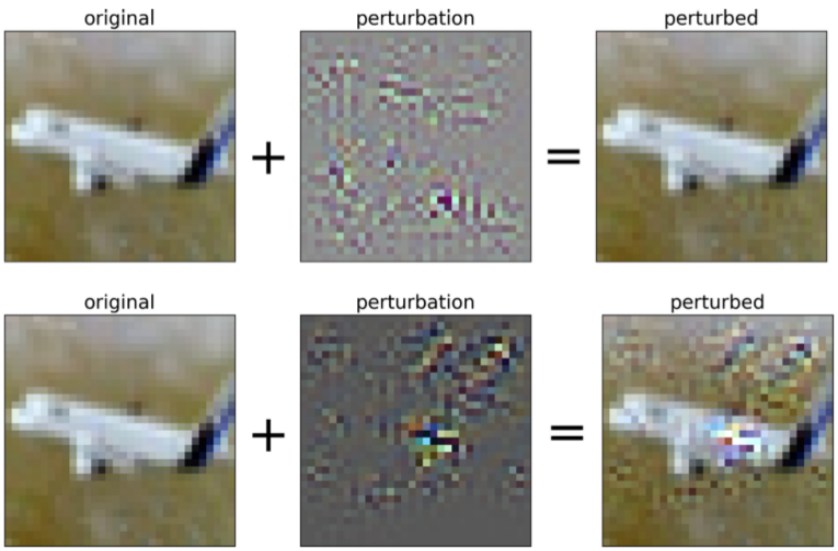
Gradient descent (targeted)

FGSM (non-targeted)

Gradient descent attacks: More samples



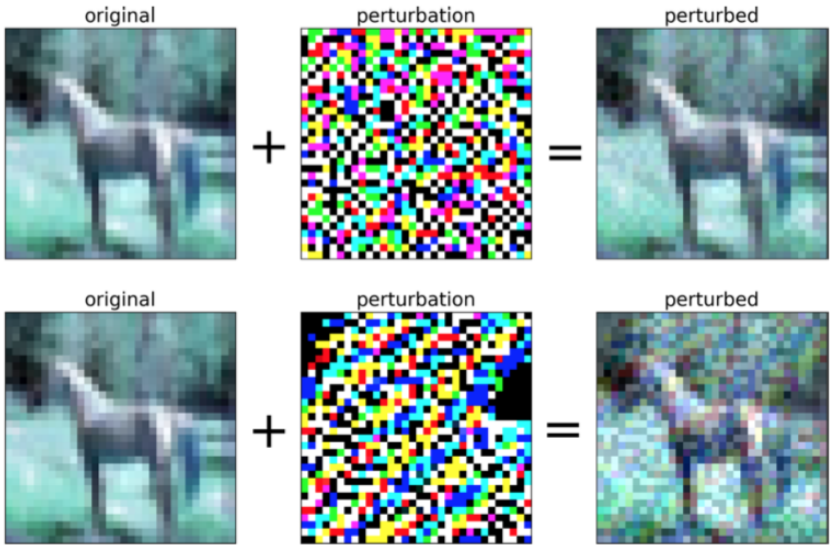
(a) Non-targeted FGSM



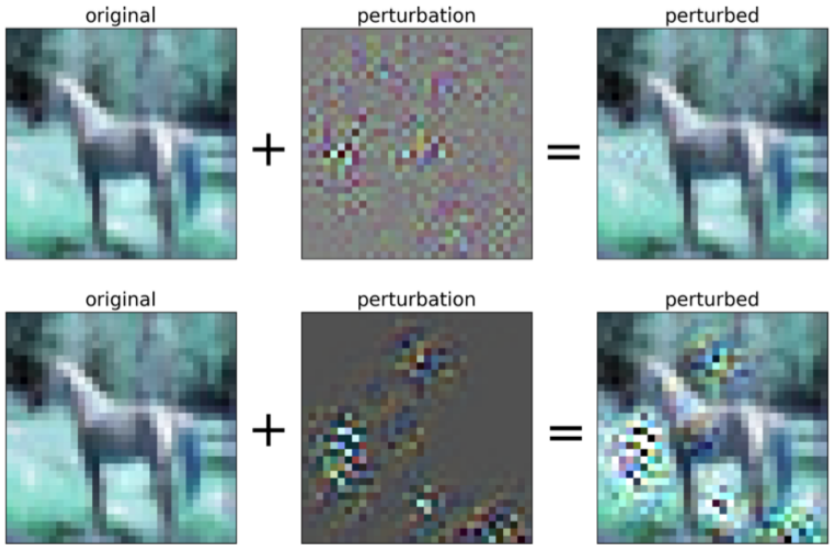
(b) Non-targeted gradient descent

Note how the PR-regularised networks cause adversarial examples to be perceptibly tampered to the human eye!

Gradient descent attacks: Even more samples



(c) Non-targeted FGSM



(d) Non-targeted gradient descent

Note how the PR-regularised networks cause adversarial examples to be perceptibly tampered to the human eye!

Results: Universal adversarial perturbations

- Universal adversarial perturbations expose a perturbation parameter, ρ. Results are measured for all datasets, at various perturbations levels.

- PeerNets consistently provide a **stronger defence** from adversarial examples, while only **marginally sacrificing accuracy**!

Results: MNIST and CIFAR-100

Table 2: Performance and fooling rates on the MNIST dataset for different levels ρ of universal adversarial noise.

Method	Original Accuracy	Accuracy / Fooling Rate				
		$\rho = 0.2$	$\rho = 0.4$	$\rho = 0.6$	$\rho = 0.8$	$\rho = 1.0$
LeNet-5	98.6%	92.7% / 7.1%	33.9% / 66.0%	14.1% / 85.9%	7.9% / 92.2%	8.2% / 91.7%
PR-LeNet-5	98.2%	94.8% / 4.6%	93.3% / 6.0%	87.7% / 11.7%	53.2% / 46.4%	50.1% / 50.1%

Table 4: Performance and fooling rates on the CIFAR-100 dataset. PR-ResNet-110 v2 has double the amount of feature maps after the last two convolutional blocks, meaning instead of (16, 16, 32, 64), it has (16, 16, 64, 128).

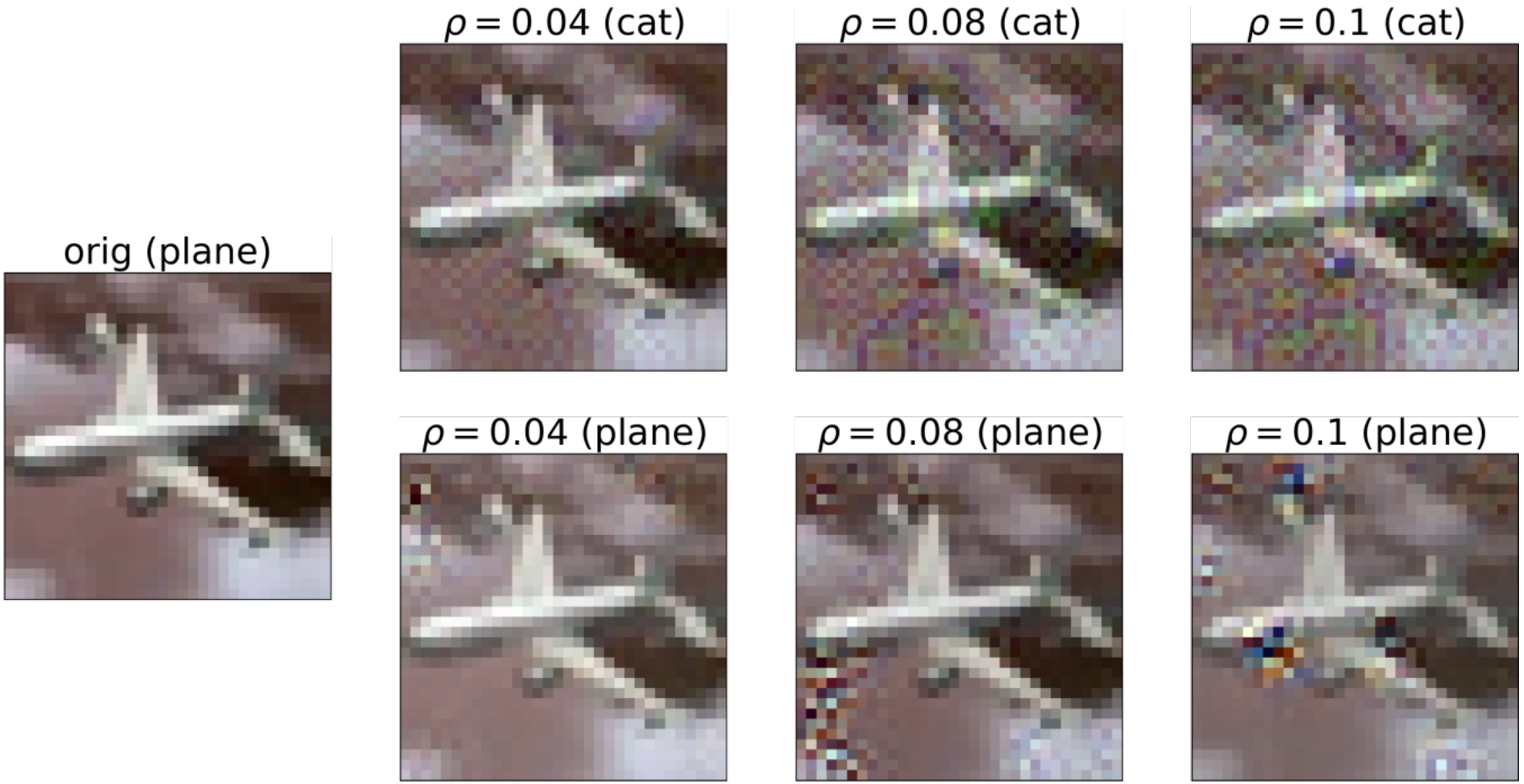
Method	Graph size	MC runs	Acc. orig [%]	Acc pert. [%] / Fool rate [%]		
				$\rho = 0.02$	$\rho = 0.04$	$\rho = 0.06$
ResNet-110	N/A	N/A	71.63	45.49 / 49.78	20.99 / 77.64	12.74 / 86.56
PR-ResNet-110	500	5	66.40	61.47 / 23.65	52.61 / 38.59	44.64 / 49.54
PR-ResNet-110 v2	500	5	70.66	63.71 / 22.84	56.40 / 35.01	36.74 / 59.76

Results: CIFAR-10

Table 3: Performance and fooling rates on the CIFAR-10 dataset. ResNet-32 v2 and PR-ResNet-32 v2 have double the amount of feature maps after the last two convolutional blocks, meaning instead of (16, 16, 32, 64), it has (16, 16, 64, 128).

Method	Graph size	MC runs	Acc. orig [%]	Acc pert. [%] / Fool rate [%]		
				$\rho = 0.04$	$\rho = 0.08$	$\rho = 0.10$
ResNet-32	N/A	N/A	92.73	55.27 / 44.42	26.84 / 73.14	22.74 / 77.34
ResNet-32 v2	N/A	N/A	94.17	44.51 / 55.32	16.65 / 83.40	12.58 / 87.58
PR-ResNet-32	50	1	88.18	87.27 / 7.98	82.43 / 14.08	69.33 / 28.80
PR-ResNet-32	50	10	89.30	87.27 / 7.13	83.32 / 12.99	70.01 / 28.31
PR-ResNet-32	100	5	89.19	87.33 / 7.43	83.37 / 13.20	70.11 / 28.19
PR-ResNet-32 v2	50	10	90.72	85.26 / 11.05	75.46 / 22.20	60.75 / 38.14
PR-ResNet-32 v2	100	5	90.65	85.35 / 11.25	75.94 / 21.82	61.10 / 37.77

Results: CIFAR-10 samples



Conclusions: Graph-based adversarial defence

- Clearly, leveraging knowledge already existing in the training set can strengthen a network in the presence of adversarial attacks.
- Further work is certainly called upon—for scaling to larger datasets, dealing with a wider variety of attacks, and exploiting the neighbourhood graph in a better way.
- Once again, input from adversarial learning experts would be very desirable!

Thank you!

Questions?

petar.velickovic@cst.cam.ac.uk

http://www.cst.cam.ac.uk/∼pv273/

http://petar-v.com/GAT

https://github.com/PetarV-/GAT