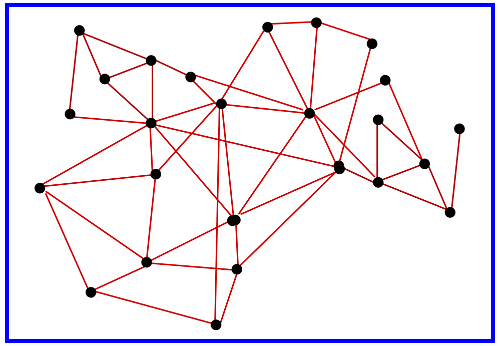
APPLICATIONS OF GRAPH LAPLACEANS: GRAPH EMBEDDINGS

Graph embeddings

- We have seen how to build a graph to represent data
- *Graph embedding* does the opposite: maps a graph to data

Given: a graph that models some data (e.g., a kNN graph)



$\longrightarrow$

Data: $\boldsymbol{Y} = [\boldsymbol{y}_1, \boldsymbol{y}_2, \cdots, \boldsymbol{y}_n]$ in $\mathbb{R}^{d\times n}$
Note: In practice $\boldsymbol{Y}$ is transposed [$\boldsymbol{Y} \in \mathbb{R}^{n\times d}$]

- Trivial use: visualize a graph ($d = 2$)
- Wish: mapping should preserve *similarities* in graph.

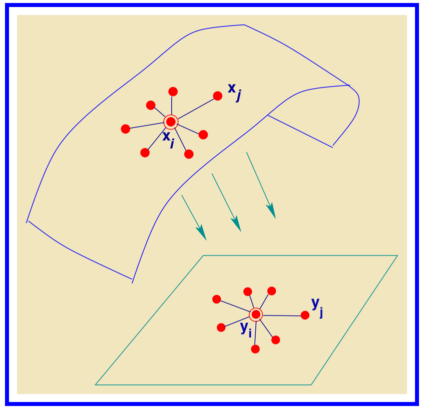
Vertex embedding: map every vertex x_i to a vector $y_i \in \mathbb{R}^d$

- Many applications [clustering, finding missing link, semi-supervised learning, community detection, ...]

Graph embedding: Embed a whole graph to a vector $y \in \mathbb{R}^d$ [e.g., graph classification]

- Graph captures similarities, closeness, ..., in data
- Many methods do this

Objective: Build a mapping of each vertex i to a data point $y_i \in \mathbb{R}^d$



- Next we focus on vertex embedding.

- Eigenmaps and LLE are two of the best known classical methods

- Eigenmaps uses the *graph Laplacean*
- Recall: Graph Laplacean is a matrix defined by :

$$\boxed{\boldsymbol{L = D - W}}$$

$$\begin{cases} w_{ij} \ge 0 & \text{if } j \in Adj(i) \\ w_{ij} = 0 & \text{else} \end{cases} \qquad D = \text{diag}\left[d_{ii} = \sum_{j \ne i} w_{ij} \right]$$

with $Adj(i)$ = neighborhood of i (excludes i)

- Remember that vertex i represents data item x_i. We will use i or x_i to refer to the vertex.
- We will find the y_i's by solving an optimization problem.

The Laplacean eigenmaps approach

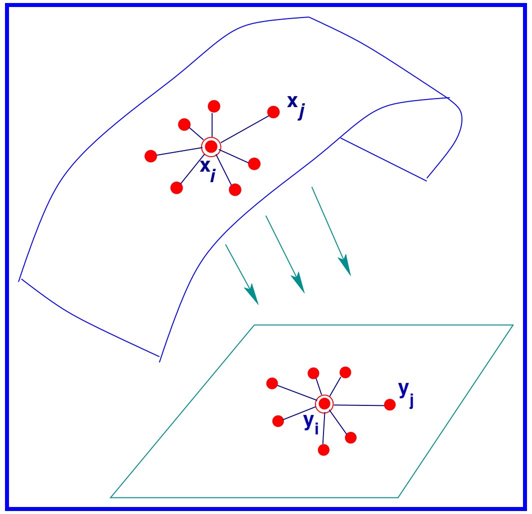
Laplacean Eigenmaps [Belkin-Niyogi '01] *minimizes*

$$\mathcal{F}(Y) = \sum_{i,j=1}^{n} w_{ij} \|y_i - y_j\|^2 \quad \text{subject to} \quad Y D Y^\top = I$$

Motivation: if $\|x_i - x_j\|$ is small (orig. data), we want $\|y_i - y_j\|$ to be also small (low-Dim. data)

➤ Original data used indirectly through its graph

➤ Objective function can be translated to a trace (see Property 3 in Lecture notes 9) and will yield a sparse eigenvalue problem



- Problem translates to:

$$\min_{\begin{cases} Y \in \mathbb{R}^{d\times n} \\ Y D Y^\top = I \end{cases}} \text{Tr}\left[Y(D-W)Y^\top\right] .$$

- Solution (sort eigenvalues increasingly):

$$(D-W)u_i = \lambda_i D u_i \; ; \quad y_i = u_i^\top ; \quad i = 1, \cdots, d$$

- An $n \times n$ sparse eigenvalue problem [In 'sample' space]

- Note: can assume $D = I$. Amounts to rescaling data. Problem becomes

$$\boxed{(I-W)u_i = \lambda_i u_i \; ; \quad y_i = u_i^\top ; \quad i = 1, \cdots, d}$$

Locally Linear Embedding (Roweis-Saul-00)

➤ LLE is very similar to Eigenmaps. Main differences:

1) Graph Laplacean matrix is replaced by an 'affinity' graph

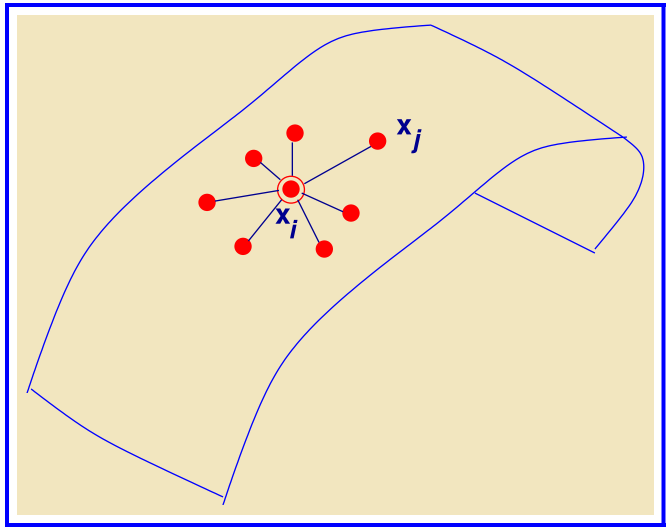
2) Objective function is changed: want to preserve graph

1. Graph: Each x_i is written as a convex combination of its k nearest neighbors:

$x_i \approx \Sigma w_{ij} x_j, \quad \sum_{j \in N_i} w_{ij} = 1$

➤ Optimal weights computed ('local calculation') by minimizing

$$\|x_i - \Sigma w_{ij} x_j\| \quad \text{for} \quad i = 1, \cdots, n$$



2. Mapping:

The y_i's should obey the same 'affinity' as x_i's $\rightsquigarrow$

Minimize:

$$\sum_i \left\| y_i - \sum_j w_{ij} y_j \right\|^2 \quad \text{subject to:} \quad Y \mathbb{1} = 0, \quad YY^\top = I$$

Solution: $\boxed{(I - W^\top)(I - W)u_i = \lambda_i u_i; \qquad y_i = u_i^\top \ .}$

- $(I - W^\top)(I - W)$ replaces the graph Laplacean of eigenmaps

Implicit vs explicit mappings

- In Eigenmaps and LLE we only determine a set of $y_i's$ in $\mathbb{R}^d$ from the data points $\{x_i\}$.

- The mapping $y_i = \phi(x_i), i = 1, \cdots, n$ is implicit

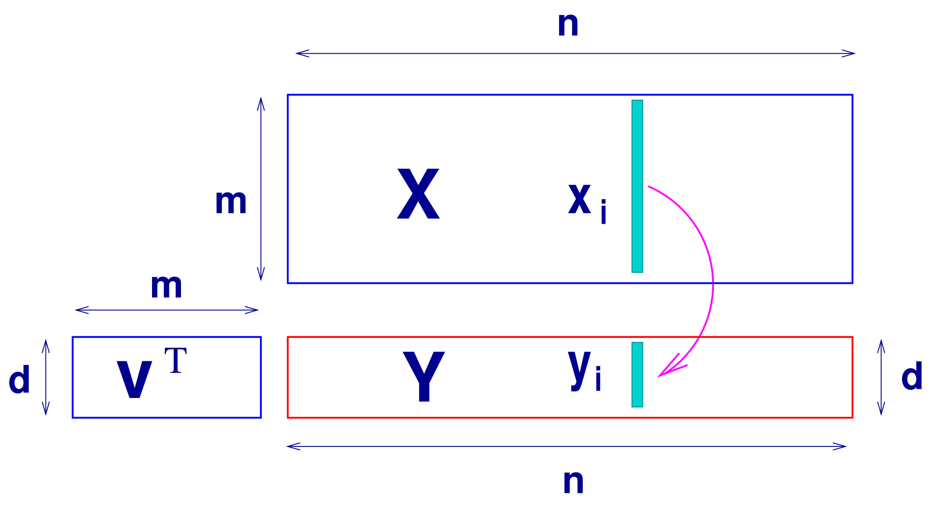
- Difficult to compute a y for an x that is not one of the x_i's

- Inconvenient for classification. Thus is known as the "The out-of-sample extension" problem

- In Explicit (also known as linear) methods: mapping ϕ is known explicitly (and it is linear.)

Locally Preserving Projections (He-Niyogi-03)

- LPP is a linear dimensionality reduction technique



- Recall the setting:

Want $V \in \mathbb{R}^{m \times d}$; $Y = V^\top X$

- Starts with the same neighborhood graph as Eigenmaps: $L \equiv D - W =$ graph 'Laplacean'; with $D \equiv diag(\{\Sigma_i w_{ij}\})$.

- Optimization problem is to solve

$$\min_{Y \in \mathbb{R}^{d\times n},\ Y D Y^\top = I} \quad \Sigma_{i,j} w_{ij} \left\| y_i - y_j \right\|^2, \quad Y = V^\top X.$$

- Difference with eigenmaps: Y is an explicit projection of X
- Solution (sort eigenvalues increasingly)

$$X L X^\top v_i = \lambda_i X D X^\top v_i \quad y_{i,:} = v_i^\top X$$

- Note: essentially same method in [Koren-Carmel'04] called 'weighted PCA' [viewed from the angle of improving PCA]

ONPP (Kokiopoulou and YS '05)

➤ Orthogonal Neighborhood Preserving Projections

➤ A linear (orthogonoal) version of LLE obtained by writing $\boldsymbol{Y}$ in the form $\boldsymbol{Y} = \boldsymbol{V}^\top \boldsymbol{X}$

➤ Same graph as LLE. Objective: preserve the affinity graph (as in LLE) *but* with the constraint $\boldsymbol{Y} = \boldsymbol{V}^\top \boldsymbol{X}$

➤ Problem solved to obtain mapping:

$$\min_{\boldsymbol{V}} \text{Tr}\left[\boldsymbol{V}^\top \boldsymbol{X}(\boldsymbol{I} - \boldsymbol{W}^\top)(\boldsymbol{I} - \boldsymbol{W})\boldsymbol{X}^\top \boldsymbol{V}\right]$$

s.t. $\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{I}$

➤ In LLE replace $\boldsymbol{V}^\top \boldsymbol{X}$ by $\boldsymbol{Y}$

More recent methods

➤ Quite a bit of recent work - e.g., methods: node2vec, DeepWalk, GraRep, ... See the following papers ... among many others :

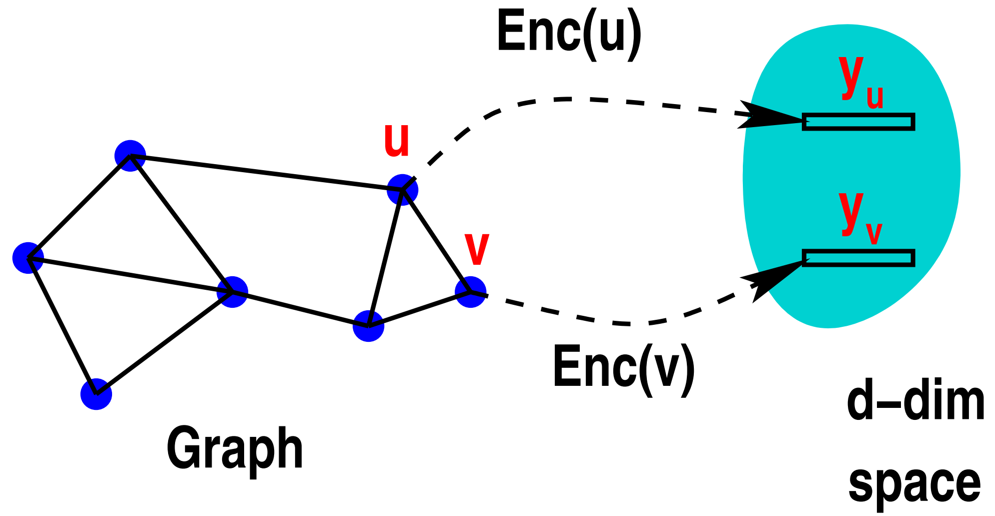
[1] *William L. Hamilton, Rex Ying, and Jure Leskovec* *Representation Learning on Graphs: Methods and Applications* arXiv:1709.05584v3

[2] *Shaosheng Cao, Wei Lu, and Qiongkai Xu* *GraRep: Learning Graph Representations with Global Structural Information*, CIKM, ACM Conference on Information and Knowledge Management, 24

[3] *Amr Ahmed, Nino Shervashidze, and Shravan Narayanamurthy*, *Distributed Large-scale Natural Graph Factorization* [Proc. WWW 2013, May 13-17, 2013, Rio de Janeiro, Brazil]

Terminology: Encoding

- The mapping from node to vector is often called an encoding



- Goal: *encode* should reflect similarity (if u and v are 'similar', their encodings should be 'close')

- Example: measure similarity by $y_v^T y_u$

Example: Graph factorization

- Line of work in Papers [1] and [3] above + others
- Instead of minimizing $\sum w_{ij}\|y_i - y_j\|_2^2$ as before

... try to minimize $$\boxed{\sum_{ij} |w_{ij} - y_i^T y_j|^2}$$

- In other words solve: $\min_Y \|W - Y^T Y\|_F^2$
- Referred to as *Graph factorization*
- Common in knowledge graphs
- Method seen so far are termed 'shallow encoders'

DEEP NEURAL NETWORKS (DNNS)

A (very) brief history of AI and DNNs

1950: 'Turing test' – can a machine think?

1956: Dartmouth College Artificial Intelligence Conference. Invention of the term 'Artificial Intelligence' [J. McCarthy]

1958: Rosenblatt invented the 'Perceptron' - idea of imitating neurons

1958+: Emphasis on symbolic processing/reasoning. invention of LISP

1964: Eliza (MIT) - a natural language processing program

1974-1980: 1st AI winter (lack of progress). Pb: NLP going nowhere

1980s: Multilayer Perceptron. More numerical/optimization approaches. Departure away from Natural Language Processing.

1982: Convolutionan Neural Networks (CNNs)

Mid-1980s: Back-propagation enters in force

1987-1993: 2nd AI winter (lack of progress). Pb: Lack of compute power

1997: Deep Blue (IBM) beets G. Gasparov - world Chess Champion

Mid-1990s: Research in 'Data Mining' gaining ground

2012: Huge breakthrough in CNNs (Alex-net) - boost from GPUs

2016: AlphaGo (DeepMind) beets Go Champion

2017: 'Transformers' ["Attention is all you need"]

2018: GPT-1 (OpenAI) ... [Large Language Models]

2019: GPT-2 — The rest is history.

Deep Neural Networks (DNNs) - general remarks

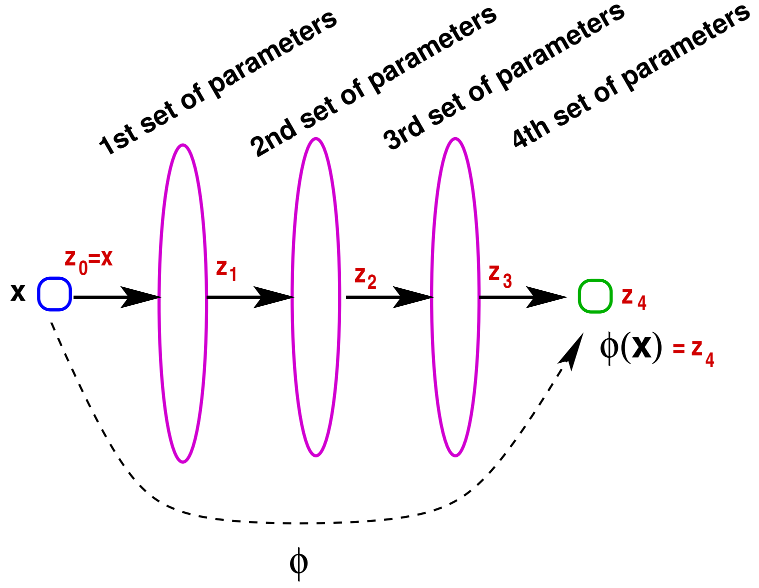
➤ Ideas of neural networks goes back to the 1960s - were popularized in early 1990s – then laid dormant until recently.

➤ Two reasons for the come-back:

- DNN are remarkably effective in some applications
- big progress made in hardware [$\rightarrow$ affordable ‘training cost’]

Multilayer Perceptron (MLP)

- Training a neural network can be viewed as a problem of approximating a function ϕ which is defined via sets of parameters:



Problem: find sets of parameters such that $\phi(x_i) \approx y_i$, for $i = 1, \cdots, n$

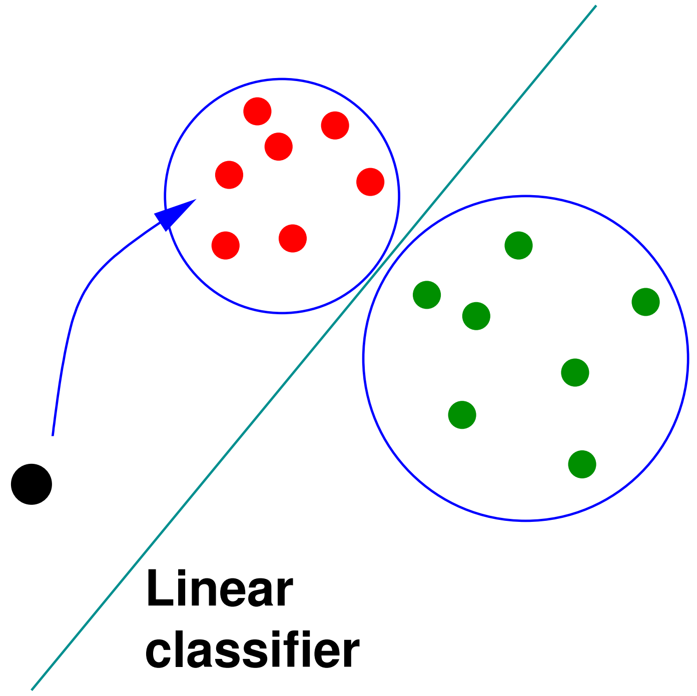
- The set $\{x_i, y_i\}$ is the training set
- Notation: Often $\hat{y}_i \equiv \phi(x_i)$ so we want $y_i \approx \hat{y}_i$ for $i = 1, \cdots, n$

Start with one layer: Perceptron

- Objective: To separate two given sets (A) and (B) of input data
- Example of application: Distinguish SPAM and non-SPAM e-mails

Linear classifiers: Find a hyperplane which best separates the data in classes A and B.

- Use hyperplane defined by: $\boxed{\phi(x) = w^T x + \beta}$



- Sets (A) , (B) defined by: $\boldsymbol{\phi(x) = \sigma(w^T x + \beta)}$ ($\boldsymbol{\sigma}$ $==$ sign function)

- $\boldsymbol{\phi(x) \geq 0 \rightarrow x \ \in (A)}$ and $\boldsymbol{\phi(x) < 0 \rightarrow x \ \in (B)}$

- Given: training data set $\boldsymbol{(x_i, y_i)}$ with labels (e.g., 'spam'–'non-spam', 'malignant' –'non-malignant',...) where $\boldsymbol{y_i = \pm 1}$

- Determine an optimal $\boldsymbol{w}$ for which $\boldsymbol{\phi(x_i) \approx y_i}$ for $\boldsymbol{i = 1, \cdots, n}$

- 'Inference': Determine class of a new 'test' item x by evaluating $\phi(x)$

Multi-Layer Perceptrons (MLPs)

➤ Neural Networks (NNs) generalize what was just described

➤ First: Instead of a single vector $\boldsymbol{w}$ we will use a $\boldsymbol{d} \times \boldsymbol{k}$ matrix $\boldsymbol{W}$ and $\boldsymbol{\sigma}$ is replaced by a continuous function known as an 'activation function'

➤ $\phi(\boldsymbol{x})$ is a vector.

➤ Second big change: use several layers of perceptrons instead of one.

➤ First Layer: transform $\boldsymbol{x}$ to $\boldsymbol{z_1 = \sigma(W_1^T x + b_1)}$ where $\boldsymbol{W_1} \in \mathbb{R}^{d \times d_1}$ and $\boldsymbol{\sigma}$ = activation

The activation functions

➤ Several choices for the activation function σ used

➤ Best known Rectified Linear Unit, or `ReLU`: $\sigma(t) = \max\{0, t\}.$

➤ The Sigmoid: $\sigma(t) = (1 + e^{-t})^{-1}$

➤ ... and the hyperbolic tangent $\sigma(t) = \tanh(t)$

➤ Note: `ReLU` ≥ 0; sigmoid and tanh lie in $(0, 1)$ and $(-1, 1)$ respectively.

➤ The sigmoid is related to logistic regression and its derivative satisfies $\sigma' = \sigma(1 - \sigma)$.

✍41 Prove the above relation.

✍42 If $\theta(t) = \tanh(t)$ and σ is the sigmoid, show that $\theta(t) = 1 - 2\sigma(-2t)$

MLP - continued

- 2nd layer transforms output z_1 from 1st layer $z_2 = \sigma(W_2^T z_1 + b_2)$
- Generally, going from layer $l-1$ to layer l:

$$z_l = \sigma(W_l^T z_{l-1} + b_l)$$

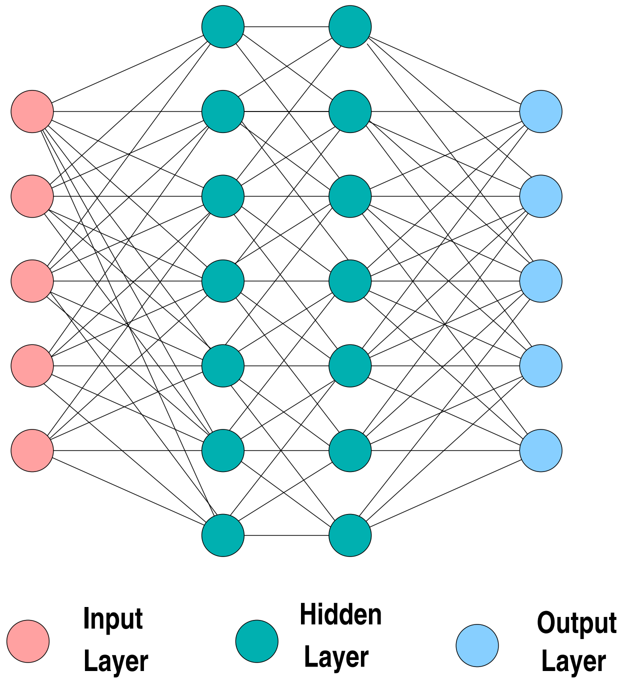
- where $W_l \in \mathbb{R}^{d_{l-1}\times d_l}$, $b_l \in \mathbb{R}^{d_l}$, and σ
- Do this for $l = 1, 2, \cdots, L+1$ - where $L =$ number of 'hidden' layers
- $z_{L+1} =$ output $= \phi(x)$. For example, when $L = 3$:

$$\phi(x) = \sigma(W_4^T \sigma(W_3^T \sigma(W_2^T \sigma(W_1^T x + b_1) + b_2) + b_3) + b_4).$$

MLP

```
Input: $x$, Output: $y$
Set: $z_0 = x$
For $l = 1 : \mathtt{L+1}$ Do:
    $z_l = \sigma(W_l^T z_{l-1} + b_l)$
End
Set: $\phi(x) := z_{L+1}$
```

- layer # 0 = input layer
- layer # $(L + 1)$ = output layer



➤ A matrix W_l is associated with layers $1, 2, \cdots, L + 1$ (for L hidden layers)

MLP

- Problem: $\boxed{\text{Find } \phi \text{ (i.e., params. } \boldsymbol{W_l}, \boldsymbol{b_l}\text{) s.t. } \phi(\boldsymbol{x_i}) \approx \boldsymbol{y_i} \text{ for } \boldsymbol{i = 1 : n}}$

Example: digit recognition

- We have a set $\boldsymbol{x_1, \cdots, x_n}$ of labeled images of digits.
- Each $\boldsymbol{x_i}$ = vectorized picture.
- $\boldsymbol{y_i}$ = a digit between 0 and 9
- Often $\boldsymbol{y_i}$ expressed as a *one-hot* vector of length 10.
- For example digit 2 will be [0,0,1,0,0,0,0,0,0,0] = $\boldsymbol{e_3}$

- If the images are 10×20 ...
- ... and we have $L = 2$ hidden layers with $d_1 = d2 = 100$
- Then input data has size $n \times d_0$ where $d_0 = 400$ and the output will be of size $n \times 10$.

Loss function and training

➤ To train the model we need a set of data points $x_i, y_i, i = 1 : n$.

➤ Input == a matrix X of size $n \times d_0$, – each row == a sample

➤ Output == matrix Y of length $n \times C$ whose rows are 'one-hot' vectors [C = # classes]

➤ Each of the internal variables z_l becomes a matrix $Z_l \in \mathbb{R}^{n\times d_l}$ Now:

$$\boxed{Z_l = \sigma(Z_{l-1} \times W_l + b_l)}$$

where W_l $\mathbb{R}^{d_{l-1}\times d_l}$, $b_l \in \mathbb{R}^{1\times d_l}$, and σ are the same as before.

- Note change of notation: samples $\boldsymbol{x}_i$ and internal variables $\boldsymbol{z}_i$ are now row vectors
- They occupy the rows of the matrix $\boldsymbol{X}$ and $\boldsymbol{Z}_l$ respectively.
- Above equation explots 'broadcasting' [feature of Pyhon]
- Define $\boldsymbol{W} = \{\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2, \cdots, \boldsymbol{W}_L, \boldsymbol{b}_L\}$ the set of parameters
- $\phi(\boldsymbol{x})$ is written as $\phi_{\boldsymbol{W}}(\boldsymbol{x})$
- Problem: Find function $\phi_{\boldsymbol{W}}$ s.t. $\phi_{\boldsymbol{W}}(\boldsymbol{x}_i) \approx \boldsymbol{y}_i$ for $i = 1 : n$
- In matrix form $\phi_{\boldsymbol{W}}(\boldsymbol{X}) \approx \boldsymbol{Y}$.
- Possible formulation:

$$\min_{\boldsymbol{W}} \mathcal{L}(\boldsymbol{W}) \equiv \|\boldsymbol{Y} - \phi_{\boldsymbol{W}}(\boldsymbol{X})\|_F^2 = \sum_{i=1}^{n} \|\boldsymbol{y}_i - \phi_{\boldsymbol{W}}(\boldsymbol{x}_i)\|_2^2$$

- Recall: $Y, \phi_W(X) \in \mathbb{R}^{n \times d_{L+1}}$ where $d_{L+1} \equiv C$ == number of classes.
- Above formulation is seldom employed. Preferred approach: exploit cross-entropy distance - a notion based on information theory.
- if $y_i, \hat{y}_i$ are scalars minimize cross-entropy loss: $\mathcal{L}(W) = -\frac{1}{n}\sum_i y_i \log(\hat{y}_i)$
- Otherwise - apply softwax operation to each row y_i: $\hat{y}_i = \mathtt{softmax}(y_i)$
- Softmax of a row/col. vector z is: Exp. Operation done componentwise

$$\mathtt{softmax}(z) = \frac{\exp(z)}{\text{sum}[\exp(z)]}$$

- Product $y_i \log(\hat{y}_i)$ in scalar case, replaced by inner product.
- Thus, the cross-entropy loss function which we want to minimize is

$$\mathcal{L}(W) = -\frac{1}{n}\sum_{i=1}^{n}(y_i, \log \hat{y}_i) \ .$$

Training a DNN

- Basic idea: use Gradient Descent $w_{j+1} = w_j - \eta_j \nabla \phi(w_j),$
scalar $\eta_j =$ termed the *step-size* or *learning rate* in ML

- Well understood algorithm when ϕ is convex - not too useful as is in ML

- In deep learning, $\phi(w)$ is often the mean of other cost functions:

$$\phi(w) = \frac{1}{n}\sum_{i=1}^{n} \phi_i(w) \qquad \rightarrow \qquad \nabla \phi(w) = \frac{1}{n}\sum_{i=1}^{n} \nabla \phi_i(w)$$

- Recall 'Mean Squared Error' (MSE) caseL $\phi(w) = \frac{1}{n}\sum_{i=1}^{n} \|y_i - \phi_w(x_i)\|_2^2$

- Similarly for the cross-entropy cost.

➤ Expensive to compute 'full' gradient $\nabla\phi$ but not $\nabla\phi_i(w)$, for some i

➤ Idea of Stochastic Gradient Descent (SGD): replace $\nabla\phi(w_j)$ by $\nabla\phi_k(w_j)$ where k is an index between 1 and n drawn at random.

➤ Result is an iteration of the type: $w_{j+1} = w_j - \eta_j \nabla\phi_k(w_j)$ - - where ϕ_k drawn at random among $\{\phi_1, \phi_2, \cdots, \phi_n\}$

Mini-batching Using a single function ϕ_k at a time not efficient.

➤ Compromise: replace function ϕ_k, by average of m functions drawn randomly from full set. Let $\mathcal{B}_j$ the sample at step j - define:

$$\boxed{\phi_{\mathcal{B}_j}(w) \equiv \frac{1}{|\mathcal{B}_j|}\sum_{k\in\mathcal{B}_j} \phi_k(w).}$$

Batch-SGD:

$$w_{j+1} = w_j - \eta_j \nabla\phi_{\mathcal{B}_j}(w_j) \quad j = 1, 2, \cdots, n_B.$$

- 'epoch' == a cycle through all mini-batches $\mathcal{B}_j$
- Simplest among a few 'optimizers'
- Best known technique to train a neural network is know as the Adaptive Moment Estimation (Adam) algorithm.
- Adam exploit two ideas: variance reduction and momentum.
- Variance reduction is a form of diagonal preconditioning - scales variables adaptively, adjusting the learning rate for each parameter individually
- Momentum == add a multiple of the previous increment $w_j - w_{j-1}$:
- GD +Momemtum: $\boxed{w_{j+1} = w_j - \eta_j \nabla\phi(w_j) + \nu(w_j - w_{j-1})}$
- Adam has two momentum terms: for gradient and for variance.

Adam:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \ , \qquad \hat{m}_t = m_t/(1-\beta_1^t)$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(g_t)^2 \ , \qquad \hat{v}_t = v_t/(1-\beta_2^t)$$
$$w_t = w_{t-1} - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}.$$

- g_t is the gradient at step t,and β_1 and β_2, are decay rates.
- Divisions, squaring, square roots, of vectors done componentwise.
- Recommended parameters: $\beta_1 = 0.9, \ \beta_2 = 0.999, \ \epsilon = 10^{-8}$.

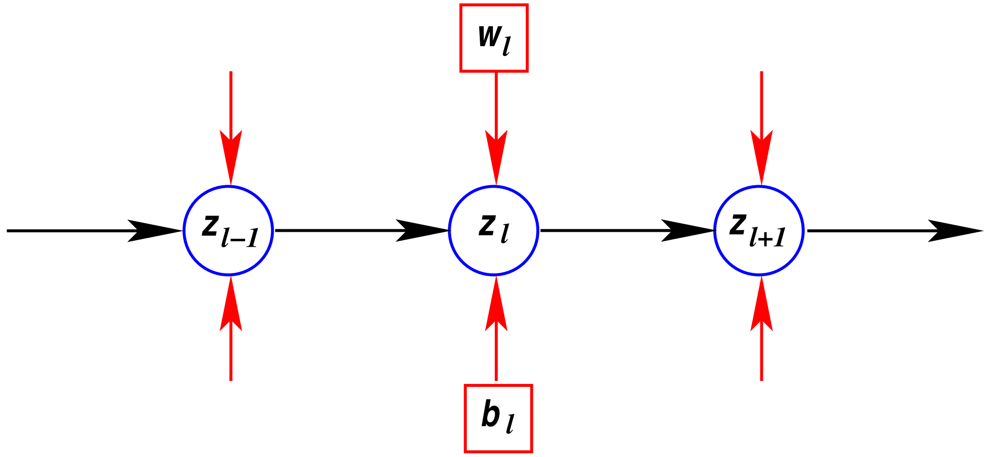
Issues with 'optimization'

- Problem is not convex, highly parameterized, ...,
- We may have a huge number of local minima.
- Hard to analyze mathematically why it all works.
- Over-parameterization plays a central role
- Notion of generalization: How does the model perform on unseen data (not in training set)? Defines accuracy of model
- Important: Lower cost function does not mean better accuracy

Back to Back-propagation

Graph of **forward** phase for calculation $z_l = \sigma(W_l^T z_{l-1} + b_l)$

➤ Nodes of comput. graph: circles (the z_k's) and squares (the parameters, W_k, b_k).



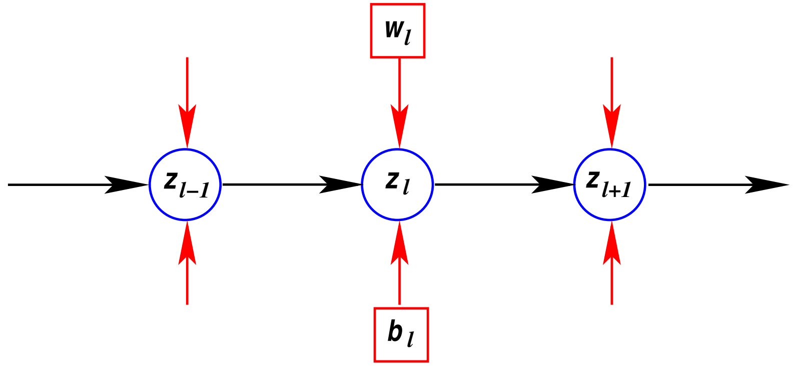
➤ Call f the original objective function - $\mathcal{L}(W)$

➤ Want: the gradient of f with respect to all parameters w_l, b_l

➤ Assume a forward propagation step was done. All nodes evaluated

➤ In back-propagation arrows in Figure are reversed.

- Evaluate: $\frac{\partial f}{\partial z_l} = \frac{\partial f}{\partial z_{l+1}} \times \frac{\partial z_{l+1}}{\partial z_l}$
- Note: $\frac{\partial f}{\partial z_{l+1}}$ was evaluated at a prior traversal step in graph



- Also: $\frac{\partial z_{l+1}}{\partial z_l}$ is readily computable from $z_{l+1} = \sigma(W_{l+1}^T z_l + b_{l+1})$
- Next follow the (reversed) arrows, and compute

$$\frac{\partial f}{\partial W_l} = \frac{\partial f}{\partial z_l} \times \frac{\partial z_l}{\partial W_l} \quad \text{and} \quad \frac{\partial f}{\partial b_l} = \frac{\partial f}{\partial z_l} \times \frac{\partial z_l}{\partial b_l}.$$

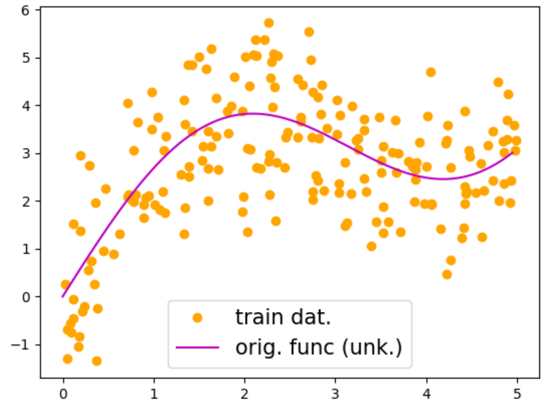
- Above calculations take place in the 'leaves' of back-propagation graph. They yield desired partial derivaties wrt W_l, b_l
- Similar situation when the z_i's are matrices [general case]
- Back-propagation ampounts to a sequence of matrix products.

AI thinking vs. numerical analysis thinking: "Attention"

Trivial example: Given very noisy 'training points' x_i, y_i to an unknown function f, 'recover' f

NA : Interpolate in Least-Squares sense

➤ Need to select interpolant type, e.g., cubic

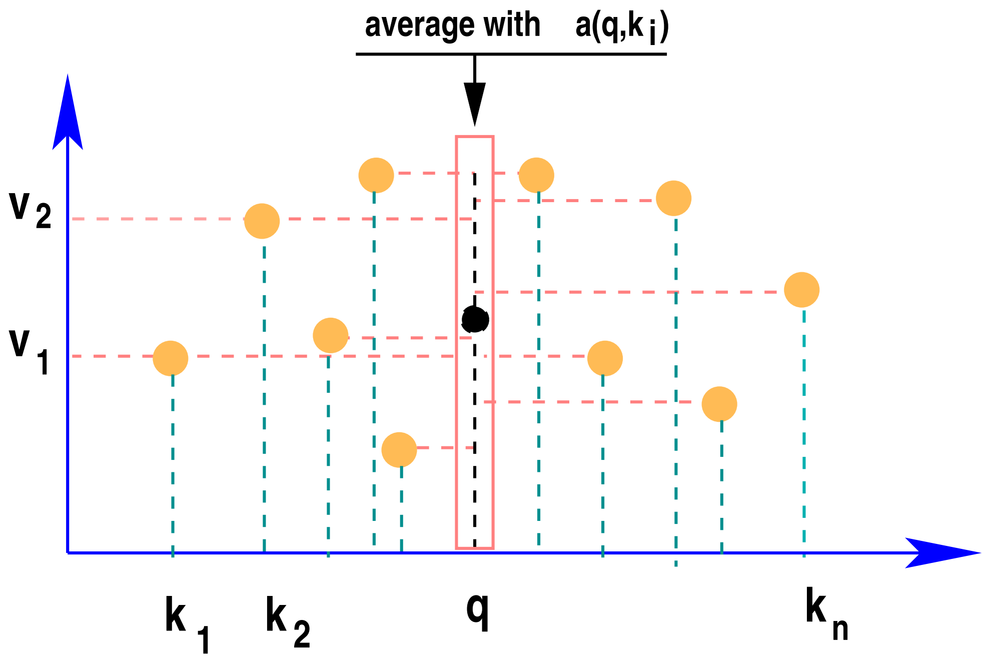


ML : use data points + some form of averaging with 'attention'.

- Given $\{k_i, v_i\}$ keys, values (NA: $x_i^{train}, y_i^{train}\}$)
- ... a query q (NA: The x where we want $f(x)$)
- ... and a Kernel $a(q, k)$. Approximation at q:

$$A(q) = \sum_i a(q, k_i) v_i$$

- "Attention" mechanism averages by giving more importance to points near q
- Nadaraya-Watson attention [Kernel Regression]



Transformers

- Up-to $\approx$ 2015: MLP, CNN, RNN+LSTM, + Focus on images. Then:
- *"Attention is all you need"* paper [Vaswani al, '17] – a major breakthrough
  - Before: LLMs needed to account for sequentiality.. order in words. Difficulties: stability, ...,
  - Now: use (1) attention + (2) adding 'positional encoding' scheme to embedding.

Tokenization + Embedding

➤ Very first step of LLMs: transform sequences of strings (words, chars) into tokens ....

➤ ... and then into vectors via embedding

➤ Result: matrix $\boldsymbol{X}_0$ of size $\boldsymbol{n} \times \boldsymbol{d}$;

➤ $\boldsymbol{n} =$ number of tokens, $\boldsymbol{d}$ = embedding dimension

➤ $\boldsymbol{X}_0$ transformed through $\boldsymbol{L}$ passes

$$\boldsymbol{X}_L = \mathcal{T}(\boldsymbol{X}_0) = (\mathcal{T}_L \circ \mathcal{T}_{L-1} \circ \cdots \circ \mathcal{T}_1)(\boldsymbol{X}_0).$$

➤ $\mathcal{T}_\ell$ termed ℓ-th ‘transformer block’

- ℓ-th block == parameterized function $\mathcal{T}_\ell(\cdot\,;\Theta_\ell) : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$.

$$\boxed{\begin{array}{ll} (a) & \boldsymbol{A}_\ell(\boldsymbol{X}_{\ell-1}) = \texttt{MHA}(\ \texttt{LN}(\boldsymbol{X}_{\ell-1})\ ), \\ (b) & \boldsymbol{M}_\ell = \texttt{MLP}(\ \texttt{LN}(\ \boldsymbol{X}_{\ell-1} + \boldsymbol{A}_\ell(\boldsymbol{X}_{\ell-1})\ )\ ), \\ (c) & \boldsymbol{X}_\ell = \boldsymbol{X}_{\ell-1} + \boldsymbol{A}_\ell(\boldsymbol{X}_{\ell-1}) + \boldsymbol{M}_\ell \equiv \mathcal{T}_\ell(\boldsymbol{X}_{\ell-1}) \end{array}}$$

  - MHA = Multi-headed attention block
  - LN = Layer-Normalization
  - MLP = Multilayer Perceptron block
  - 'Residual Attention' $\boldsymbol{X}_{\ell-1} + \boldsymbol{A}_\ell(\boldsymbol{X}_{\ell-1})$ in (b) and (c) helps capture incremental changes

- Additional LN step added at last layer: $\boxed{\boldsymbol{X}_L := \texttt{LN}(\boldsymbol{X}_L)}$

- Final output is passed to a bias-free linear layer to obtain loss function

MLP stage - written in (row) vector form

$$\begin{cases} \mathbf{a}_i^{(\ell)} = \texttt{MHA}(\ \texttt{LN}(\mathbf{x}_1^{(\ell-1)}, \mathbf{x}_2^{(\ell-1)}, \cdots, \mathbf{x}_i^{(\ell-1)}) \\ \boxed{\mathbf{m}_i^{(\ell)}} = \texttt{MLP}(\ \texttt{LN}(\mathbf{x}_i^{(\ell-1)} + \mathbf{a}_i^{(\ell)})\ ) \\ \mathbf{x}_i^{(\ell)} = \mathbf{x}_i^{(\ell-1)} + \mathbf{a}_i^{(\ell)} + \mathbf{m}_i^{(\ell)} \end{cases}$$

$$\boxed{\mathbf{m}_i^{(\ell)}} = \underbrace{\boldsymbol{W}_{\texttt{dwn}}^{(\ell)}}_{\text{Weights (2nd MLP)}} \ \underbrace{\boldsymbol{\sigma}}_{\text{Activation function, (e.g. ReLU, thanh,..)}} \left(\underbrace{\boldsymbol{W}_{\texttt{up}}^{(\ell)}}_{\text{Weights (1st MLP)}} \ \underbrace{\gamma}_{\text{Normalization function}} (\mathbf{x}_i^{(\ell-1)} + \mathbf{a}_i^{(\ell)}) + \mathbf{b}_{\texttt{up}}^{(\ell)} \right) + \mathbf{b}_{\texttt{dwn}}^{(\ell)}$$

Weights

(2nd MLP)

Activation function,

(e.g. ReLU, thanh,..)

Weights

(1st MLP)

Normalization function

GPT3: counting the 175B parameters

- Embedding dimension in GPT3 = $d_{embed} = 12,288$

MLP: Dimension used for $W_{up}, W_{dwn} = 4d_{embed} \times d_{embed} = 4 \times (12,288)^2 \approx 4 \times 1.5 \times 10^8 = 6 \times 10^8$ Each. i.e., $\boxed{\approx 12 \times 10^8}$

- Multiply by the number of blocks (=96 in GPT3) $\rightarrow$ $\boxed{\approx 120B}$

MHA: 4 matrices of size $d_{embed} \times 120$ times 96 heads times 96 blocks, $\rightarrow$ $\boxed{\approx 54B+}$

- Total 174B + add initial params for embeddings $\approx 175B$
- For Llama3: 450B params.

Q: Where is the Linear Algebra?

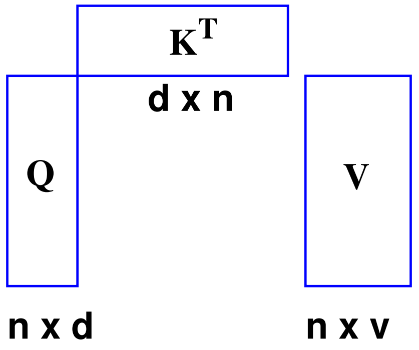
- More precisely: Which Linear Algebra tools/methods can help here?
- Really need to look deep inside the various boxes to find answers
- Some recent advances were deeply rooted in NLA -
- Next: 2 examples

Example: A pure LA idea that is very successful

> *"Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention"*, A. Katharopoulos, A. Vyas, N. Pappas, F. Fleuret ('21)

- Scaled Dot-product Attention :
- Softmax applied row-wise
- $Q: n \times d, \ K: n \times d, \ V: n \times v$

$$\boxed{A_l = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V}$$



- Cost: $O(n^2)$ – But without the softmax term:
- Do $K^T V$ first – then $Q\times$ result: $\rightarrow O(n)$ cost
- Idea: replace softmax(QK^T) by $\phi(Q)\phi(K^T)$

  (Judicious func. ϕ applied rowwise to Q, K)

- Very simple idea. Very impactful paper [Huge gain in training time]

Example: Low-rank structure in DNN

> *"LoRa: Low-Rank Adaptation of Large Language Models"* E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen ('21)

- LoRa able to reduce number of parameters in Chat-GPT3 from 175B to $17M$ - (i.e., / by 10,000)

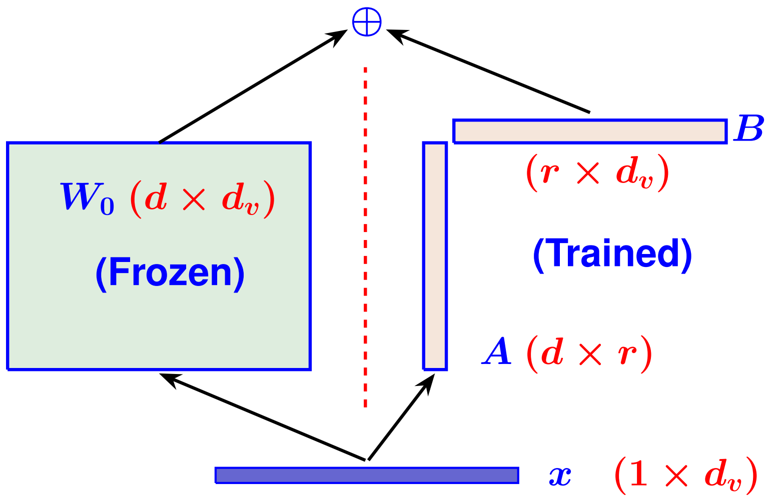
- Observed: Depth of DNN $\rightarrow$ low-dimensional paramater-spaces

Over-parameterization $\rightarrow$ Low-Dim.

- Idea: Low-rank modifs to some W_0

- Many follow-up papers, (e.g., analysis)

$$\boxed{h = x \times W_0 + x \times AB}$$



GRAPH NEURAL NETWORKS

Graph Neural Networks

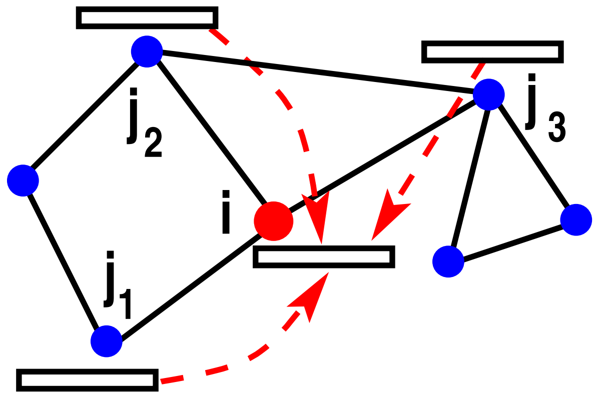
- Idea: exploit graphs in neural networks - Replace CNN with Aggregation
- A GNN is not a specific model but rather a framework
- Goal == to produce an embedding for nodes of a graph

Given: A graph $G = (V, E)$ (n nodes) + feature matrix $X^{(0)} \in \mathbb{R}^{n\times d_0}$

- Row i of $X^{(0)}$ == 'feature' of node i
- At layer l we will create/modify features in $\mathbb{R}^{d_l}$ for each node
- Fundamental operation used for this: the message-passing mechanism

$$\boldsymbol{x}_i^{(\ell+1)} = \mathtt{UPDATE}\left(\boldsymbol{x}_i^{(\ell)},\ \mathtt{AGGREGATE}(\ \{\boldsymbol{x}_j^{(\ell)} : j \in \mathcal{N}(v_i)\}\ )\right),$$

- Node features created/modified from layer to layer - At layer l: $x_i^{(l)}$.
- Message-passing: aggregate features from neighbors $\mathcal{N}(v_i)$:



- In addition, the features are linearly modified by weights to be optimized.
- See the GCN example.

Graph Convolutional Networks

- Aggregate operation simple to describe.

- Let $\boldsymbol{A}$ = adjacency matrix and $\tilde{\boldsymbol{D}} =$ diag of row-sums of $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$. Define:

$$\hat{\boldsymbol{A}} := \tilde{\boldsymbol{D}}^{-1/2} \tilde{\boldsymbol{A}} \tilde{\boldsymbol{D}}^{-1/2}$$

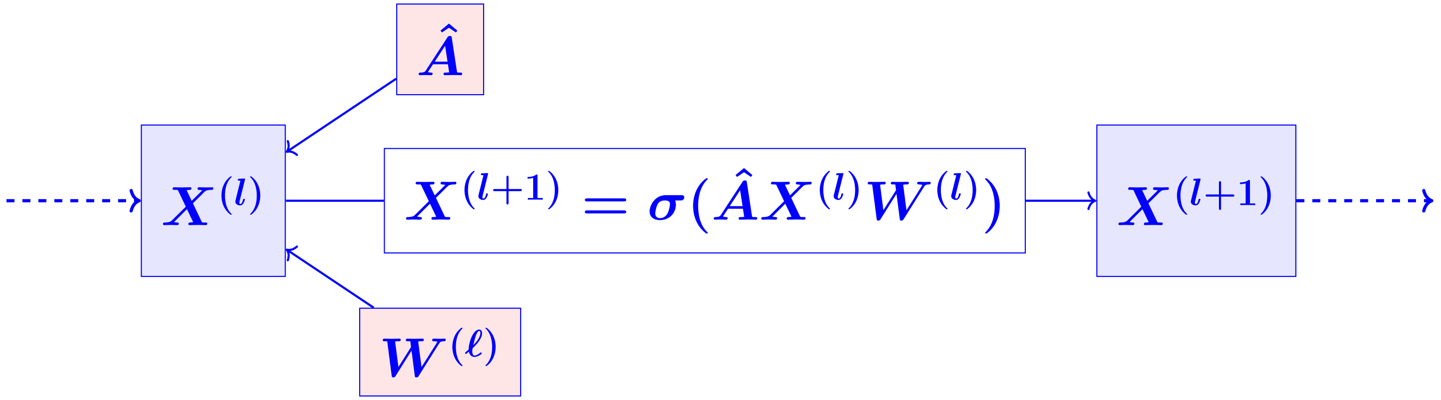
GCN - Layer l to $l + 1$ update

$$\boldsymbol{X}^{(\ell+1)} = \boldsymbol{\sigma}\left(\hat{\boldsymbol{A}} \boldsymbol{X}^{(\ell)} \boldsymbol{W}^{(\ell)}\right)$$

- $\boldsymbol{W}^{(l)}$ is a parameter determined by training

- Each row of $\boldsymbol{X}^{(l)}$ is a feature

- At last layer this becomes the desired embedding [a row for each node.]



Layer l to $l+1$ in GCN

✍43 See Pytorch codes for the ENZYME dataset [graph classification]

✍44 See Pytorch codes for the Cora dataset [Node classification]

✍45 What is the difference between Node and Graph Classification?

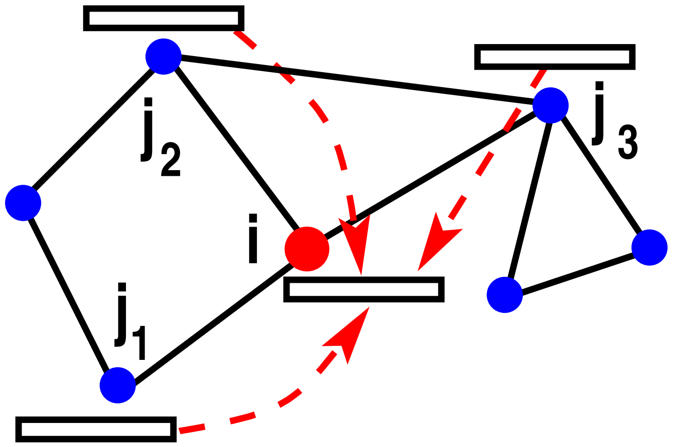
Graph Attention Networks (GATs)

- Idea: exploit ‘Attention’ in GCN
- In very simple terms: we now add weights to adjacency matrix
- Weights based on attention mechanism - and they are learned

Given: A graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ ($\boldsymbol{n}$ nodes) + feature matrix $\boldsymbol{X}^{(0)} \in \mathbb{R}^{n \times d_0}$

- Goal same as before == produce an embedding for nodes of a graph

- Attention-based message-passing: compute weighted average of transformed features in $\mathcal{N}(v_i)$:



$$\boldsymbol{X}^{(\ell+1)} = \boldsymbol{\sigma}\left(\boldsymbol{A}_\alpha \boldsymbol{X}^{(\ell)} \boldsymbol{W}^{(\ell)}\right)$$

- Entry α_{ij} of A_α == attention weight between nodes i and j:

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

- $\equiv$ *softmax* of neighoring e_{ij}'s – What is e_{ij}?

- e_{ij} == Attention score between node i and neighbors $j \in \mathcal{N}(i)$

$$\boxed{e_{ij} = \texttt{LeakyReLU}\left(a^T \cdot \left[W^{(\ell),T} x_i \,\|\, W^{(\ell),T} x_j\right]\right)}$$

$\|$ == concatenation

- Note: Bias often added before applying `LeakyReLu`
- $a \in \mathbb{R}^{2d}$ is a learnable attention vector
- $\mathtt{LeakyReLU}(t) = \max\{t, \alpha t\}$ (where $0 \leq \alpha \ll 1$)
- Aggregation similar to GCN. Main differences:
  - Scaled Adjacency matrix $\hat{A}$ replaced by A_α
  - A_α is now learned
  - Additional parameter: $a \in \mathbb{R}^{2d}$

✍46 See Pytorch codes for GAT

Final words

- *Many* interesting new matrix problems in areas that involve the effective exploitation of data
- Change happens fast in part because we are better connected
- In particular: many many resources available online.
- Huge potential for making a good impact by looking at a topic from new perspective
- To a researcher in computational linear algebra : Tsunami of change on types or problems, algorithms, frameworks, culture,..

- My favorite quote. Alexander Graham Bell (1847-1922) said:

When one door closes, another opens; but we often look so long and so regretfully upon the closed door that we do not see the one which has opened for us.

- Visit my web-site at *www.cs.umn.edu/ ~saad*

- More complete version of this material will available in course csci-8314 – notes (and more) are open to all.

Thank you !