1. Basics of deep learning
2. Deep learning for graphs
3. Graph Convolutional Networks
4. GNNs subsume CNNs and Transformers

Machine Learning as Optimization

- **Supervised learning:** we are given input $\boldsymbol{x}$, and the goal is to predict label $\boldsymbol{y}$.
- **Input x can be:**
  - Vectors of real numbers
  - Sequences (natural language)
  - Matrices (images)
  - Graphs (potentially with node and edge features)
- **We formulate the task as an optimization problem.**

Machine Learning as Optimization

- **Formulate the task as an optimization problem:**

$$\min_{\Theta} \boxed{\mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))} \longleftarrow \text{Objective function}$$

- Θ: a set of **parameters** we optimize
  - Could contain one or more scalars, vectors, matrices …
  - E.g. $\Theta = \{Z\}$ in the shallow encoder (the embedding lookup)

- $\mathcal{L}$: **loss function**. Example: L2 loss

$$\mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x})) = \|y - f(x)\|_2$$

  - Other common loss functions:
    - L1 loss, huber loss, max margin (hinge loss), cross entropy …
    - See https://pytorch.org/docs/stable/nn.html#loss-functions

Loss Function Example

- One common loss for classification: **cross entropy (CE)**
- Label $\boldsymbol{y}$ is a categorical vector (one-hot encoding)
  - e.g. $\boldsymbol{y} =$

0	0	1	0	0

y is of class “3”

- $f(\boldsymbol{x}) = \mathrm{Softmax}\big(g(\boldsymbol{x})\big)$
  - Recall from lecture 3: $f(\boldsymbol{x})_i = \dfrac{e^{g(x)_i}}{\sum_{j=1}^{C} e^{g(x)_j}}$

$g(x)_i$ denotes i-th coordinate of the vector output of func. $g(x)$

where C is the number of classes.

  - e.g. $f(\boldsymbol{x}) =$

0.1	0.3	0.4	0.1	0.1

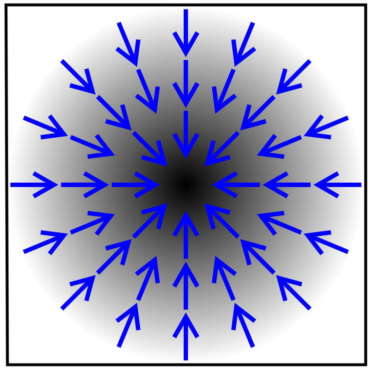
- $\mathrm{CE}\big(\boldsymbol{y}, f(\boldsymbol{x})\big) = -\sum_{i=1}^{C}(y_i \log f(x)_i)$
  - y_i and $f(x)_i$ are the **actual** and **predicted** values of the i-th class.
  - **Intuition:** the lower the loss, the closer the prediction is to one-hot
- **Total loss over all training examples:**
  - $\mathcal{L} = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \mathrm{CE}\big(\boldsymbol{y}, f(\boldsymbol{x})\big)$
    - $\mathcal{T}$: training set containing all pairs of data and labels $(\boldsymbol{x}, \boldsymbol{y})$

Machine Learning as Optimization

- **How to optimize the objective function?**
- **Gradient vector:** Direction and rate of fastest increase

Partial derivative

$$\nabla_{\Theta}\mathcal{L} = \left(\frac{\partial\mathcal{L}}{\partial\Theta_1}, \frac{\partial\mathcal{L}}{\partial\Theta_2}, \ldots\right)$$



https://en.wikipedia.org/wiki/Gradient

  - $\Theta_1, \Theta_2 \ldots$: components of Θ
- Recall **directional derivative** of a multi-variable function (e.g. $\mathcal{L}$) along a given vector represents the instantaneous rate of change of the function along the vector.
- Gradient is the directional derivative in the **direction of largest increase.**

Gradient Descent

- **Iterative algorithm:** repeatedly update weights in the (opposite) direction of gradients until convergence

$$\Theta \leftarrow \Theta - \eta \nabla_{\Theta} \mathcal{L}$$

- **Training:** Optimize Θ iteratively
  - **Iteration**: 1 step of gradient descent
- **Learning rate (LR) η:**
  - Hyperparameter that controls the size of gradient step
  - Can vary over the course of training (LR scheduling)
- **Ideal termination condition:** gradient = **0**
  - In practice, we stop training if it no longer improves performance on **validation set** (part of dataset we hold out from training).

Stochastic Gradient Descent (SGD)

- **Problem with gradient descent:**
  - Exact gradient requires computing $\nabla_{\Theta}\mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$, where $\boldsymbol{x}$ is the **entire** dataset!
    - This means summing gradient contributions over all the points in the dataset
    - Modern datasets often contain billions of data points
    - Extremely expensive for every gradient descent step
- **Solution: Stochastic gradient descent (SGD)**
  - At every step, pick a different **minibatch** $\boldsymbol{\mathcal{B}}$ containing a subset of the dataset, use it as input $\boldsymbol{x}$

Minibatch SGD

- **Concepts:**
  - **Batch size**: the number of data points in a minibatch
    - E.g. number of nodes for node classification task
  - **Iteration**: 1 step of SGD on a minibatch
  - **Epoch**: one full pass over the dataset (# iterations is equal to ratio of dataset size and batch size)
- **SGD is unbiased estimator of full gradient:**
  - But there is no guarantee on the rate of convergence
  - In practice often requires tuning of learning rate
- Common optimizer that improves over SGD:
  - Adam, Adagrad, Adadelta, RMSprop …

Neural Network Function

- **Objective:** $\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$
- In deep learning, function f can be very complex
- **Example:**
  - To start simple, consider linear function

$$f(\boldsymbol{x}) = W \cdot \boldsymbol{x}, \qquad \Theta = \{\mathrm{W}\}$$

  - Then, if f returns a scalar, then W is a learnable **vector**

$$\nabla_W f = \left(\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3} \dots\right)$$

  - But, if f returns a vector, then W is the **weight matrix**

$$\nabla_W f = \begin{bmatrix} \frac{\partial f_1}{\partial w_{11}} & \frac{\partial f_2}{\partial w_{12}} \\ \frac{\partial f_1}{\partial w_{21}} & \frac{\partial f_2}{\partial w_{22}} \end{bmatrix}$$

Jacobian matrix of f

Intuition: Back Propagation

- **Goal:** $\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$
  - To minimize $\mathcal{L}$, we need to evaluate the gradient: $\nabla_W \mathcal{L} = \left(\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3} \dots\right)$ which means we need to derive derivative of $\mathcal{L}$.
- **Overview of Back-propagation:**
  - $\mathcal{L}$ is composed from some set of predefined building block functions $g(\cdot)$
  - For each such g we also have its derivative g'
  - Then we can automatically compute$\nabla_{\Theta}\mathcal{L}$ by evaluating appropriate funcs. g' on the minibatch $\boldsymbol{\mathcal{B}}$.

Back-propagation

- How about a more complex function:

$$f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x}), \Theta = \{\mathrm{W}_1, \mathrm{W}_2\}$$

In other words:

$$f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x})$$
$$h(x) = W_1\boldsymbol{x}$$
$$\mathrm{g}(z) = W_2 z$$

- Recall **chain rule**:

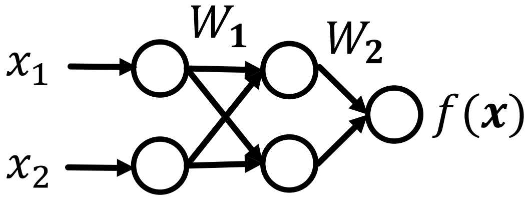
$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}g}{\mathrm{d}h} \cdot \frac{\mathrm{d}h}{\mathrm{d}x}$$ or $$f'(x) = g'\big(h(x)\big)h'(x)$$

- **Example:** $\nabla_{\boldsymbol{x}} f = \dfrac{\partial f}{\partial(W_1\boldsymbol{x})} \cdot \dfrac{\partial(W_1\boldsymbol{x})}{\partial \boldsymbol{x}}$

- **Back-propagation**: Use of **chain rule** to propagate gradients of intermediate steps, and finally obtain gradient of $\mathcal{L}$ w.r.t. Θ.

Back-propagation Example (1)

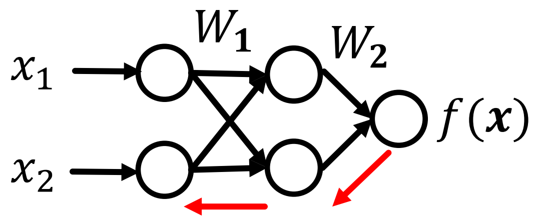
- **Example:** Simple 2-layer linear network
- $f(\boldsymbol{x}) = g\big(h(x)\big) = W_2(W_1\boldsymbol{x})$



- $\mathcal{L} = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\boldsymbol{\mathcal{B}}} \left\|\left(\boldsymbol{y}, -f(\boldsymbol{x})\right)\right\|_{\mathbf{2}}$
  - The loss $\mathcal{L}$ sums the L2 loss in a minibatch $\boldsymbol{\mathcal{B}}$.
- **Hidden layer:**
  - Intermediate representation of input $\boldsymbol{x}$
  - Here we use $h(x) = W_1\boldsymbol{x}$ to denote the hidden layer
  - $f(\boldsymbol{x}) = W_2h(x)$

Back-propagation Example (2)

Remember:
$f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x})$
$h(x) = W_1\boldsymbol{x}$
$g(z) = W_2 z$

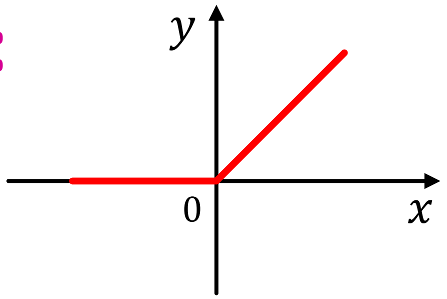


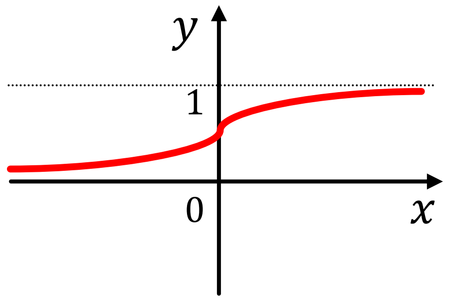
- **Forward propagation:**
  Compute loss starting from input
  - $$\boldsymbol{x} \xrightarrow[\text{Multiply } W_1]{} h \xrightarrow[\text{Multiply } W_2]{} g \xrightarrow[\text{Loss}]{} \mathcal{L}$$
- **Back-propagation to compute gradient of**
  $$\Theta = \{W_1, W_2\}$$
  Start from loss, compute the gradient
  $$\frac{\partial \mathcal{L}}{\partial W_2} = \underbrace{\frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial W_2}}_{\xrightarrow{\text{Compute backwards}}}, \qquad \frac{\partial \mathcal{L}}{\partial W_1} = \underbrace{\frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial W_2} \cdot \frac{\partial W_2}{\partial W_1}}_{\xrightarrow{\text{Compute backwards}}}$$

Non-linearity

- Note that in $f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x})$, W_2W_1 is another matrix (vector, if we do binary classification)
  - Hence $f(\boldsymbol{x})$ is still linear w.r.t. $\boldsymbol{x}$ no matter how many weight matrices we compose

- **We introduce non-linearity:**
  - **Rectified linear unit (ReLU)**
    $ReLU(x) = \max(x, 0)$
  - **Sigmoid**
    $$\sigma(x) = \frac{1}{1 + e^{-x}}$$



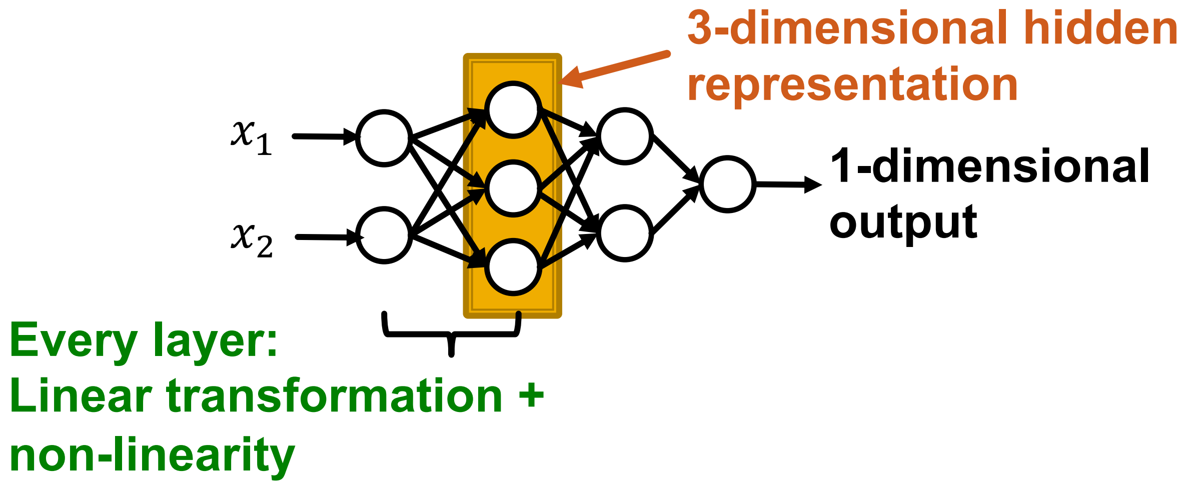


Multi-layer Perceptron (MLP)

- **Each layer of MLP combines linear transformation and non-linearity:**

$$\boldsymbol{x}^{(l+1)} = \sigma(W_l \boldsymbol{x}^{(l)} + b^l)$$

  - where W_l is weight matrix that transforms hidden representation at layer l to layer $l+1$
  - b^l is bias at layer l, and is added to the linear transformation of $\boldsymbol{x}$
  - σ is non-linearity function (e.g., sigmod)
- Suppose $\boldsymbol{x}$ is 2-dimensional, with entries x_1 and x_2


3-dimensional hidden representation
x_1
x_2
1-dimensional output
Every layer:
Linear transformation + non-linearity

Summary

- **Objective function:**

$$\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$$

- f can be a simple linear layer, an MLP, or other neural networks (e.g., a GNN later)
- Sample a minibatch of input $\boldsymbol{x}$
- **Forward propagation:** Compute $\mathcal{L}$ given $\boldsymbol{x}$
- **Back-propagation:** Obtain gradient $\nabla_{\mathrm{w}}\mathcal{L}$ using a chain rule.
- Use **stochastic gradient descent (SGD)** to optimize for Θ over many iterations.