CS 5683: Big Data Analytics

Machine Learning for Graphs: Basics of Neural Networks

Arunkumar Bagavathi
Department of Computer Science
Oklahoma State university

Topics Overview

High. Dim. Data

- Data Features
- Dimensionality Reduction
- *Application* Rec. Systems

Text Data

- Clustering
- Non-linear Dim. Reduction
- *Application* IR

Graph Data

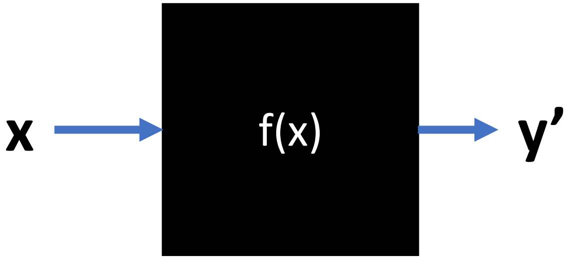
- PageRank
- ML for Graphs
- Community Detection

Others

- Data Streams Mining
- Intro. to Apache Spark

Let’s Start Easy

- **A supervised machine learning task:** Given input ***x***, the goal is to predict label ***y***



- ***Types of x:***
  - Vectors of real numbers
  - Sequences (text)
  - Matrices (images)
  - Graphs (potentially with node and edge features)

- **Let’s formulate this ML task as an optimization problem**

Supervised Task as Optimization

- Formulate the supervised task as an optimization problem

$$\min_{\theta} \boxed{\mathcal{L}(y, f(x))} \longleftarrow \textbf{Objective function}$$

- ***θ:*** a set of parameters to optimize – could be one or more scalars, vectors, matrices,…

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

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

  - **Other common loss functions:** L1 loss, Cross Entropy, KL Divergence,…
  - **Check out:** https://pytorch.org/docs/stable/nn.html#loss-functions

Loss Function Example

- Common loss function for classification tasks with neural networks: Cross **Entropy (CE)**
- Label **y** is a categorical vector (one-hot encoding)
  - Example:

0	0	0	1	0

y is class '4' in the 5 class classification problem

- *f(x) = Softmax(g(x))* ➔ $f(x)_i = \frac{e^{g(x)_i}}{\sum_{j=1}^{C} e^{g(x)_j}}$

g(x)**$_i$* denotes ith coordinate of the vector output of function: ***g(x)

C is the number of classes

0.1	0.2	0.2	0.4	0.1

- $CE\big(y, f(x)\big) = -\sum_{i=1}^{C}(y_i \log f(x)_i)$
  - y_i, $f(x)_i$ are the **actual** and **predicted** value of the ith class
  - **Intuition:** the lower the loss, the closer the prediction is to one-hot
- **Total loss over all training instances:**
  - $\mathcal{L} = \sum_{(x,y)\ \varepsilon\ \tau} CE(y, f(x))$
  - τ: training data containing pairs of data and labels ***(x,y)***

Optimizing the Objective Function

- **Gradient vector:** Direction and magnitude of the fastest increase

$$\nabla_\theta \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}, \ldots\right)$$

Partial derivative

  - θ_1, θ_2 are multiple parameters of the supervised task or model
- **Gradient Descent:** Iterative algorithm to update parameters in the opposite direction of gradients until convergence – ***training stage***

$$\theta = \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}$$

Learning rate – a hyperparameter to control the size of gradient step

- **Ideal algorithm termination condition: 0** gradient
  - In reality, we stop training if it no longer improve the performance of the underlying task on validation dataset (small chunk of training data)

Stochastic Gradient Descent

- **Problems with gradient descent:** Extracting gradient requires computing $\nabla_\theta \mathcal{L}(y, f(x))$, where ***x*** is the entire dataset!
  - This means summing gradient contributions over all points in the dataset
  - Modern dataset often contains billions of data instances
  - Extremely expensive for every gradient step
- ***Solution:* Stochastic Gradient Descent (SGD)**
  - Pick only one sample to make a step
  - **Problems:** The loss keeps fluctuating a lot for each sample and does not decrease after some point. It requires many iterations

Minibatch Stochastic Gradient Descent

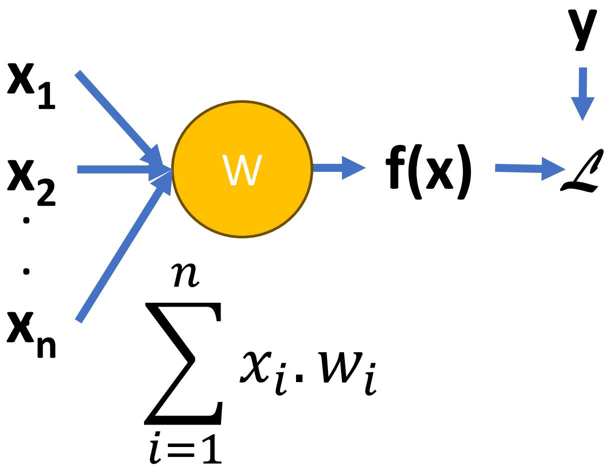
- **Solution to SGD problems:**
  - Pick a different minibatch $\mathcal{B}$ containing a subset of training data for each iteration of the algorithm
  - Use $\mathcal{B}$ as input $\boldsymbol{x}$ for optimizing θ
- **Concepts:**
  - **Batch size:** the number of data points in minibatch
  - **Iteration:** 1 step of SGD on a minibatch
  - **Epoch:** one full pass over the entire dataset (# iterations = $\frac{dataset\ size}{batch\ size}$)
- Minibatch SGD is an unbiased estimator of full gradient – however, there is no guarantee on the rate of convergence
- Optimizers that improve over SGD: ***Adam, AdaGrad, RMSProp,…***

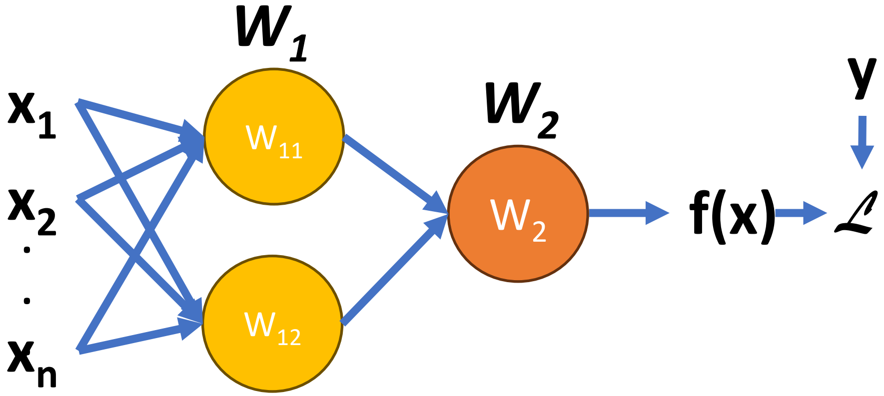
Neural Network Function

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

$$f(x) = W.x \longrightarrow \Theta = \{W\}$$

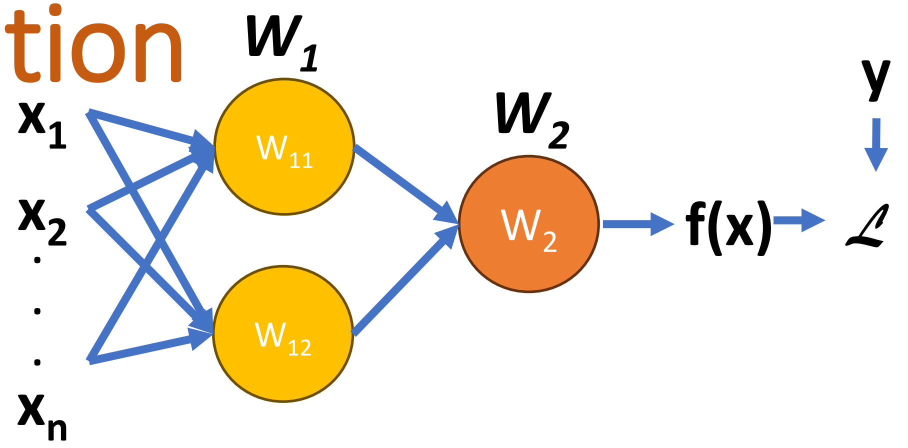
- If ***f*** returns a scalar, then ***W*** is a learnable vector
- If ***f*** returns a vector, then ***W*** is a learnable matrix





Apply softmax function usually to the output of the last/output layer

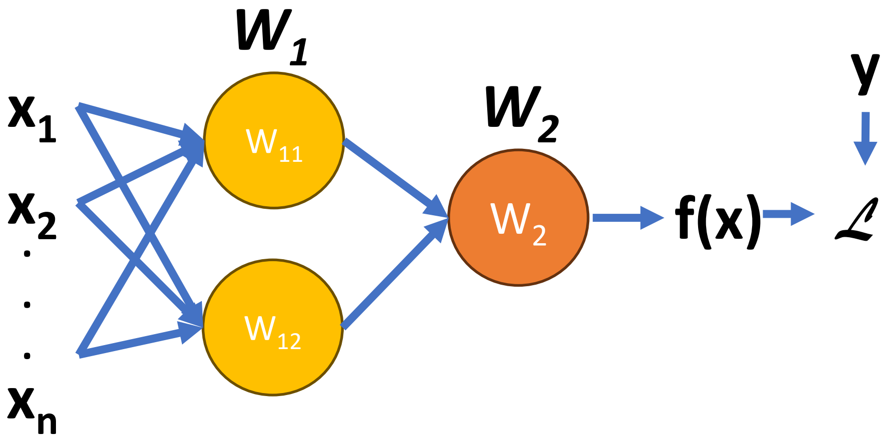
NN – Back-propagation



- ***f(x) = $W_2(W_1(x))$*** ⟶ **$\Theta = \{W_1, W_2\}$**
- **Chain Rule:** $\frac{dz}{dx} = \frac{dz}{dy} . \frac{dy}{dx}$
- Example: $\nabla_x f = \frac{\partial f}{\partial W_1(x)} . \frac{\partial W_1(x)}{\partial x}$
- We use chain rule to propagate gradients of intermediate steps to finally obtain gradient of $\mathcal{L}$ w.r.t θ

Back-propagation Example (1)

- Consider two-layer linear network
- ***$f(x) = W_2(W_1(x)) = g(h(x))$***

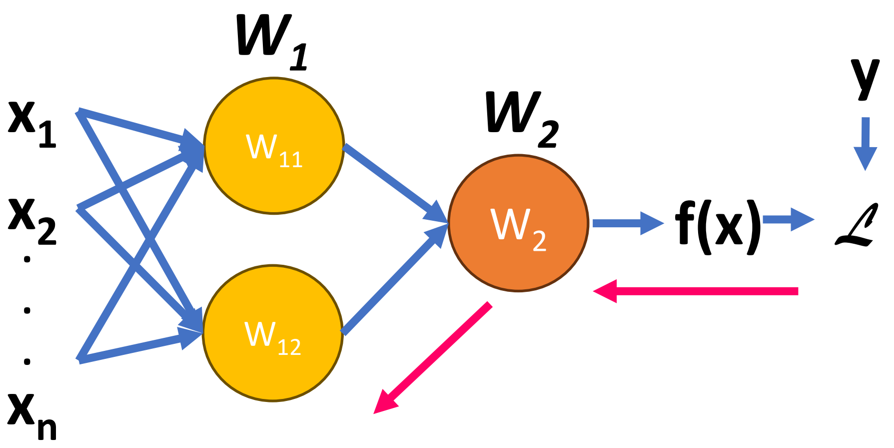


- $\mathcal{L} = \sum_{(x,y)\in\mathcal{B}} ||y - f(x)||_2$ sums L2 loss in a minibatch $\mathcal{B}$
- ***Forward propagation:*** Compute loss starting from input
  - $x \xrightarrow[\text{Multiply } W_1]{} h \xrightarrow[\text{Multiply } W_2]{} g \xrightarrow[\text{Loss}]{} \mathcal{L}$

Back-propagation Example (2)

- **Back-propagation to compute gradient of**

  $\Theta = \{W_1, W_2\}$



Remember:

$f(x) = W_2(W_1(x))$

Start from loss and compute the gradient

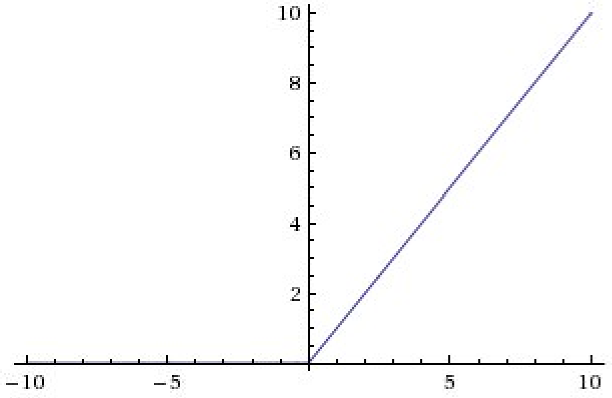
- $$\frac{\partial \mathcal{L}}{\partial W_2} = \frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial W_2}$$
- $$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial W_2} \cdot \frac{\partial W_2}{\partial W_1}$$

Compute backwards

Non-linearity

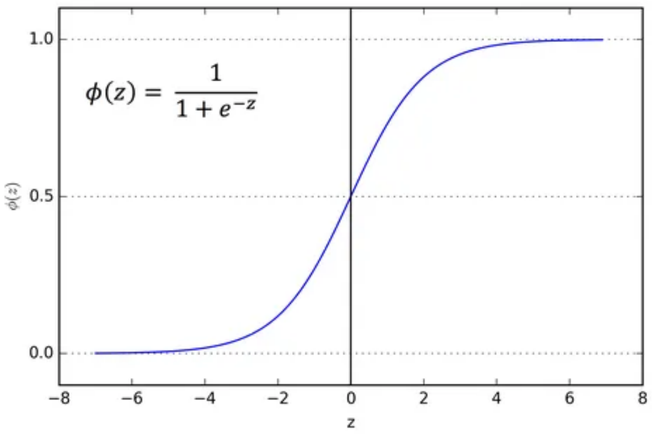
- In our simple example of **$f(x) = W_2(W_1(x))$**, **$f(x)$** is still linear w.r.t **x** no matter how many weight matrices we compose in intermediate layers
- **Introducing non-linearity:**
  - **Rectified Linear Unit (ReLU)**

$$ReLU(x) = max(x, 0)$$



  - **Sigmoid function**

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



Purpose: Converts linear signals to non-linear signals to learn complex and higher order polynomials

Multi-Layer Perceptron (MLP)

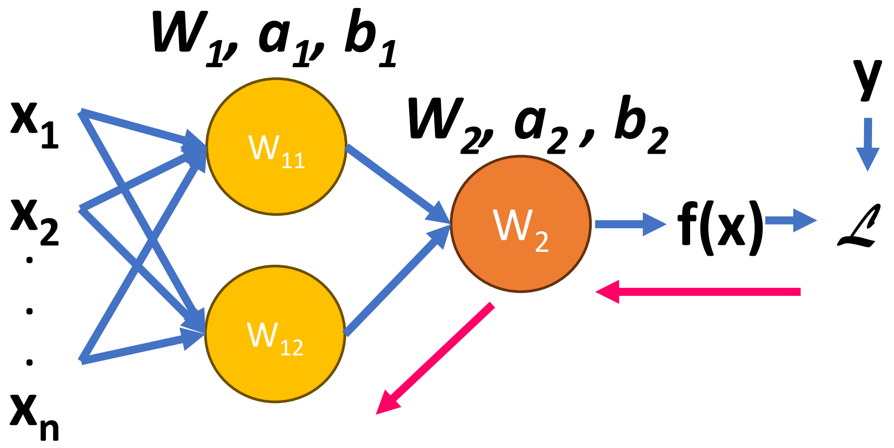
- Each layer of MLP combines linear transformation and non-linearity

$$a^l = x^{l+1} = \sigma(W^l . x^l + b^l)$$

  - $W^{(l)}$: weight matrix to transform hidden representations at layer '*l*' to layer '*l+1*'
  - b^l: bias of layer *l* and added to the linear transformation of x
  - σ – some non-linear function (sigmoid function, for example)

- Each layer of the neural network – perform linear + non-linear transformation

Putting them all together…



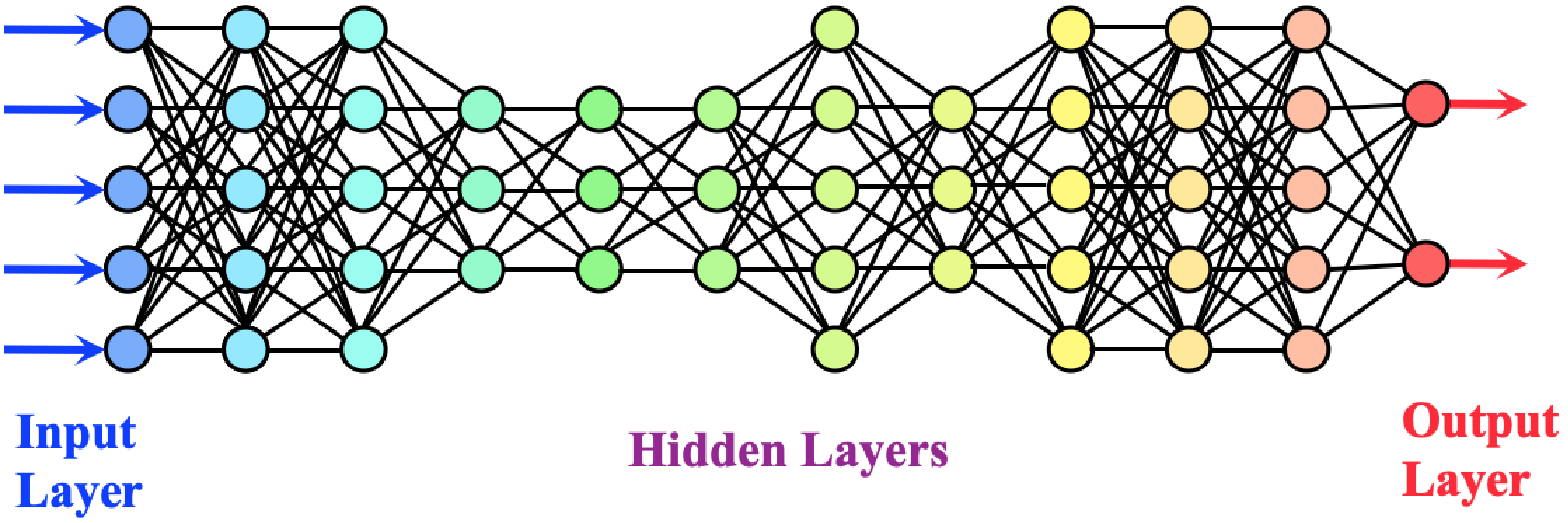
- We update W_l and b_l using
  - $W_l = W_l - \eta \frac{\partial \mathcal{L}}{\partial W_l}$
  - $b_l = b_l - \eta \frac{\partial \mathcal{L}}{\partial b_l}$
- Chain rule for gradient:

$$\frac{\partial \mathcal{L}}{\partial W_2} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} ; \frac{\partial \mathcal{L}}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_2}$$

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial W_1} ; \frac{\partial \mathcal{L}}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial b_1}$$

Assume: $\boldsymbol{z_l} = \boldsymbol{W_l} . \boldsymbol{a_{l-1}} + \boldsymbol{b_l}$ with $\boldsymbol{a_0} = \boldsymbol{x}$

Real-World Deep Neural Nets



*** https://medium.com/binaryandmore/beginners-guide-to-deriving-and-implementing-backpropagation-e3c1a5a1e536

Summary

- **Objective function** $\min_{\theta} \mathcal{L}(y, f(x))$
  - $\boldsymbol{f}$ can be a simple linear layer, MLP, or any other neural networks (say, GNN)
  - Sample a minibatch $\mathcal{B}$ of input $\boldsymbol{x}$
  - **Forward Propagation:** Compute $\mathcal{L}$ given $\boldsymbol{x}$
  - **Backpropagation:** obtain gradient $\nabla_{\theta} \mathcal{L}$ using a chain rule
  - Use **Stochastic Gradient Descent (SGD)** to optimize θ over many iterations of updating the matrix $\boldsymbol{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$

Questions???