Geometry Processing

8 Data-driven Approach

(Part 2)

Ludwig-Maximilians-Universität München

Session 8: Data-driven Approach (Part 2)

- Automatic Differentiation and PyTorch
- From Graphs to Manifolds
  - Permutation Invariant and Equivariant
  - General Tasks and Layers
  - Laplacian (revisited)
- Differentiable Rendering and PyTorch3D
- Summary and Outlook

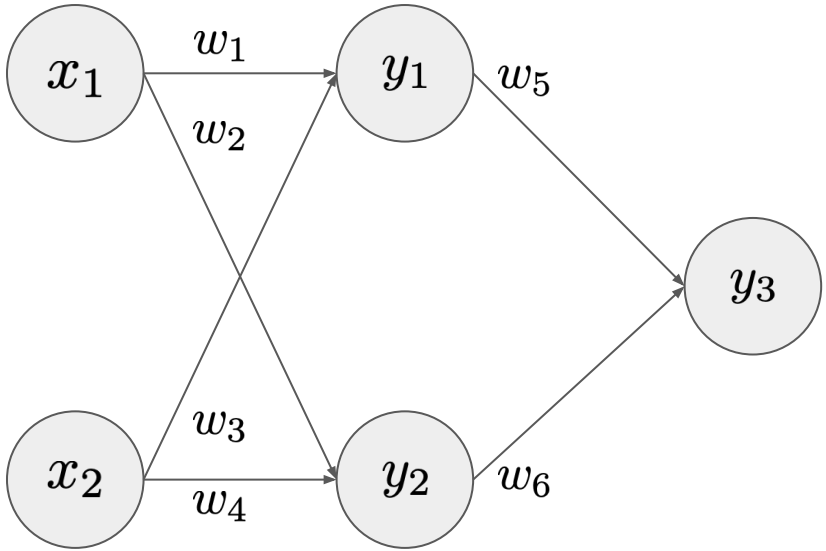
Computational Graph

Computational graph presents a general abstraction of a transformation function. Given an input, the output is determined by the graph.

Neural network represents a set of non-linear transformation functions that maps a given input to an output, and can be represented by a computational graph.

When the weights are determined, the network represents a unique function.

$$y_3 = w_5 y_1 + w_6 y_2 = w_5(w_1 x_1 + w_3 x_2) + w_6(w_2 x_1 + w_4 x_2)$$



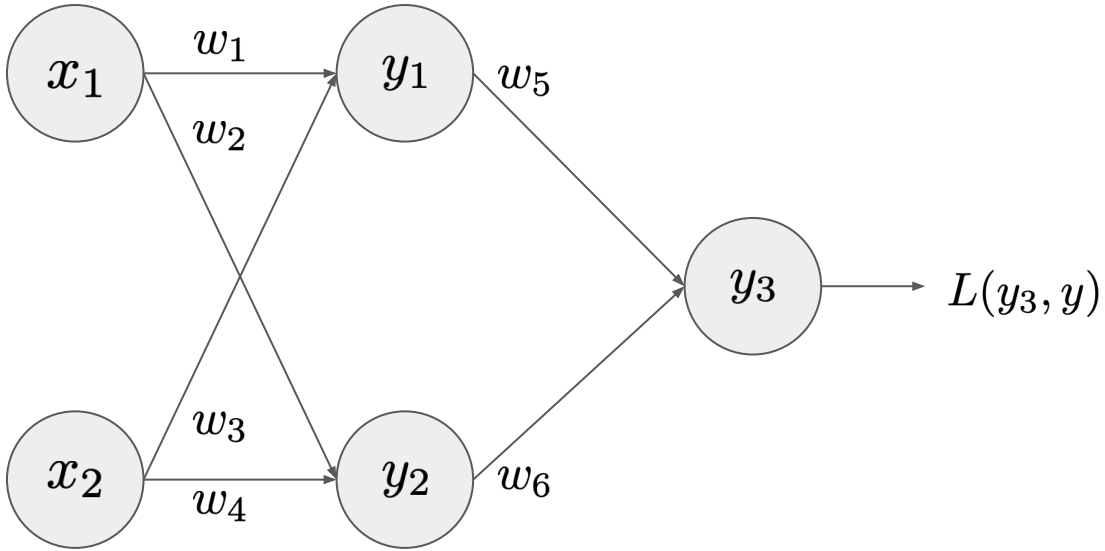
This network is a linear transformation because it does not include any non linear activation function

Automatic Differentiation, Forward Pass, and Loss

Automatic differentiation can determining the weights of a computational graph.

To employee AutoDiff, one has to define a criteria, the *loss* function, to compare the graph output in a *forward pass* to an expected target.

$$y_3 = w_5 y_1 + w_6 y_2 = w_5(w_1 x_1 + w_3 x_2) + w_6(w_2 x_1 + w_4 x_2)$$

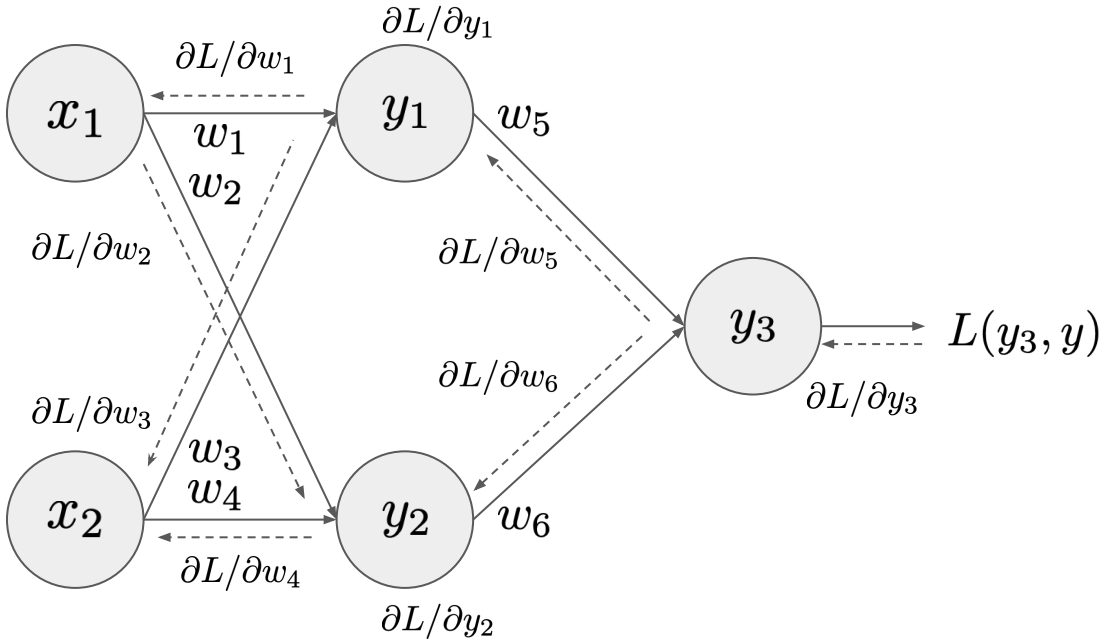


Backward Pass

With a computed loss, one can conduct a differentiation on the graph, and compute the gradient of a weight in a *backward pass*, or back propagation.

The *gradient* conveys information for future weight adjustments, which requires an *optimizer*.

$$y_3 = w_5 y_1 + w_6 y_2 = w_5(w_1 x_1 + w_3 x_2) + w_6(w_2 x_1 + w_4 x_2)$$



Optimizer

An optimizer minimizes the loss of a computational graph. The most frequent optimizer is the *stochastic gradient descent (SGD)*. SGD is both efficient and numerically stable.

For a given weight w_i, and computed gradient ΔL, SGD consists of a hyperparameter learning rate η, and update rule:

$$\Delta w_i = -\eta \nabla L = -\eta \partial L / \partial w_i$$

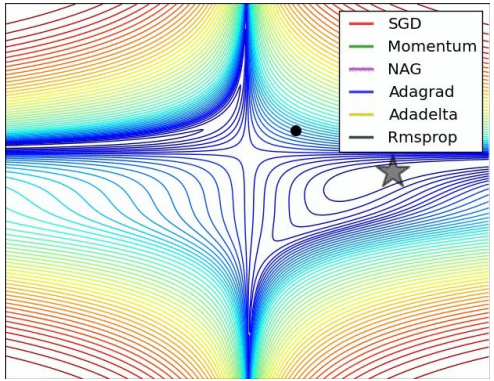
$$w_i = w_i + \Delta w_i$$

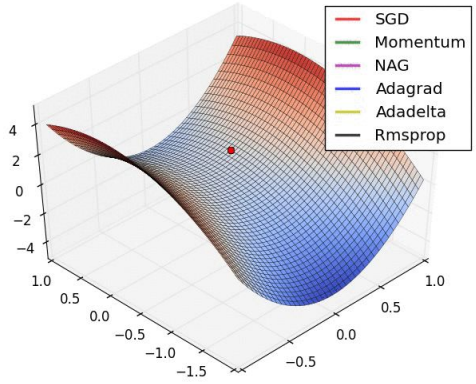
Other frequently used optimizers:

Adam, Rmsprop, …

The selection of optimizers is domain specific,

but SGD is the one that theoretically sound.

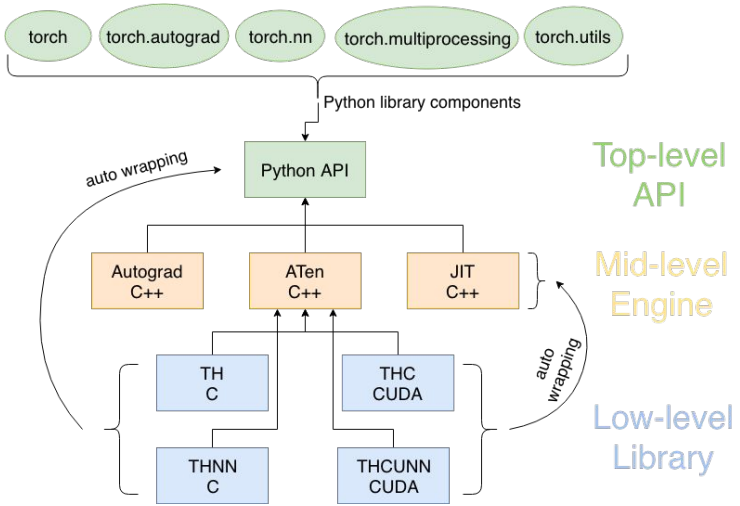




PyTorch

PyTorch is one of the frameworks in machine learning that offers automatic differentiation.

It is popular due to the friendly designed APIs and stability (compare to TensorFlow, which often breaks the APIs).



PyTorch: Dataset

`Dataset` is an abstraction that defines how to access inputs and expected outputs of a computational graph.

PyTorch offers predefined dataset, but mostly we have to define our own dataset hence require to implement the `__init__`, `__getitem__`, and `__len__` methods. The `__getitem__` returns both an input instance tensor, and the expected output.

Reference: https://pytorch.org/docs/stable/data.html?highlight=dataset#torch.utils.data.Dataset

```
from torch.utils.data import Dataset
class ThisIsDataset(Dataset):
    def __init__(self):
        return
    def __getitem__(self, idx):
        return X, y
    def __len__(self):
        return 0
```

PyTorch: DataLoader

A `DataLoader` offers efficient data batching, i.e. a list of data samples.

The data loader consumes a required input that implements `Dataset`.

Reference: https://pytorch.org/docs/stable/data.html?highlight=dataset#torch.utils.data.DataLoader

```
from torch.utils.data import DataLoader
for batch_idx, sample in enumerate(DataLoader(dataset, batch_size=2, shuffle=True)):
    ...
```

PyTorch: nn.Module

`nn.Module` is the key module to define a neural network.

It is required to implement the `forward` method that defines the computational graph.

Reference: https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module

```
import torch
import torch.nn as nn

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()

    def forward(self, x):
        return torch.pow(x, 2) + 1
```

PyTorch: Loss and BackProp

Loss can be computed using provided loss function or be self defined. The backward pass is computed through loss by calling `backward` method.

Reference: https://pytorch.org/docs/stable/nn.html#loss-functions

```
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn

ds        = Dataset()
dl        = DataLoader(ds, batch_size=100)
f         = Model()
criterion = nn.CrossEntropyLoss()
for epoch in range(10000):
   for i, data in enumerate(dataloader, 0):
       inputs, labels = data
       outputs = f(inputs)               # forward
       loss = criterion(outputs, labels) # loss
       loss.backward()                   # backward
```

PyTorch: Optimizer

An optimizer utilizes the computed gradients from a backward pass, and updates the weights and biases accordingly.

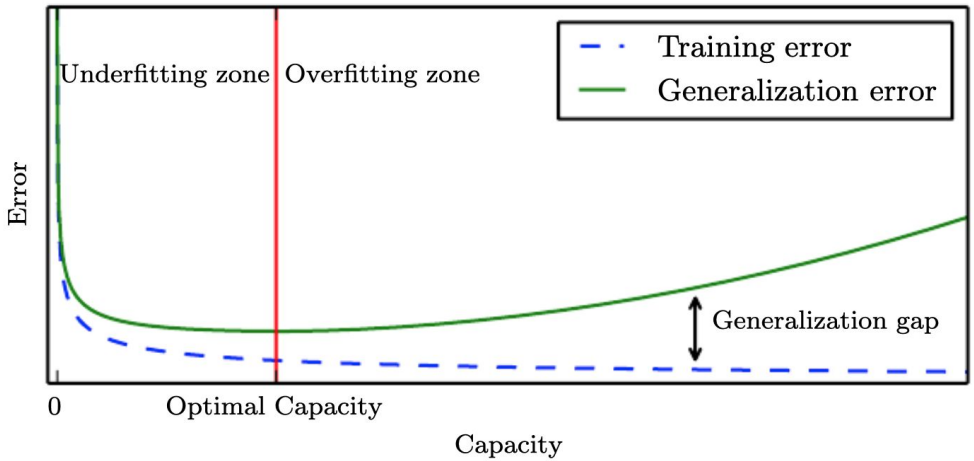
Reference: https://pytorch.org/docs/stable/optim.html

```
import torch.optim as optim

optimizer = optim.SGD(f.parameters(), lr=0.001, momentum=0.9)
for epoch in range(10000):
    for i, data in enumerate(dataloader, 0):
        inputs, labels = data

        optimizer.zero_grad()
        outputs = f(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

PyTorch: Training, validation, and testing

During training, one can use a training set and a validation set to prevent overfitting.

After observing the optimal capacity, one can stop the training and apply testing for inference tasks.



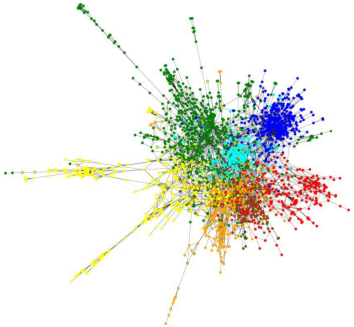
Session 8: Data-driven Approach (Part 2)

- Automatic Differentiation and PyTorch
- From Graphs to Meshes
  - Permutation Invariant and Equivariant
  - General GNN Tasks and Layers
  - Laplacian (revisited)
- Differentiable Rendering and PyTorch3D
- Summary and Outlook

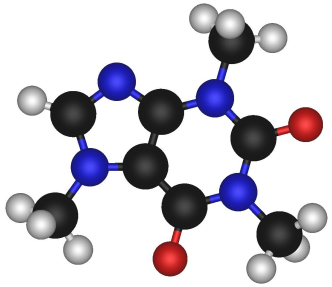
Graphs and Meshes (revisited)

Roughly speaking, mesh is a subset of graph (irregular non-Euclidean structure).

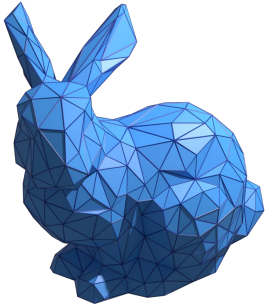
Social Network



Publication Network



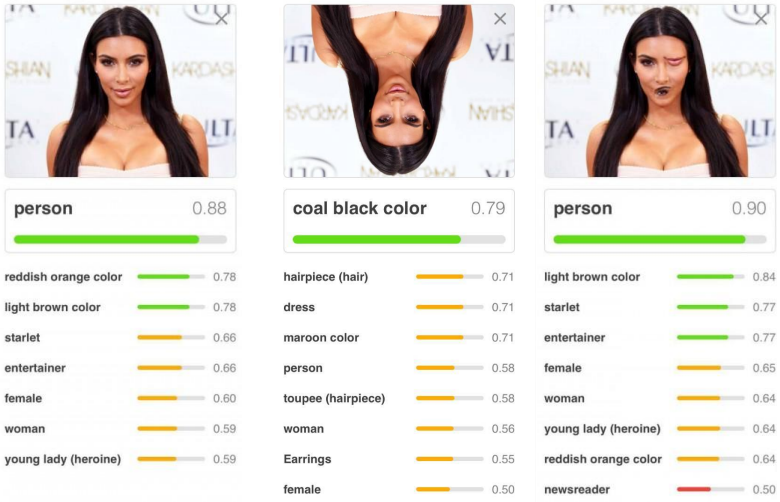
Molecule

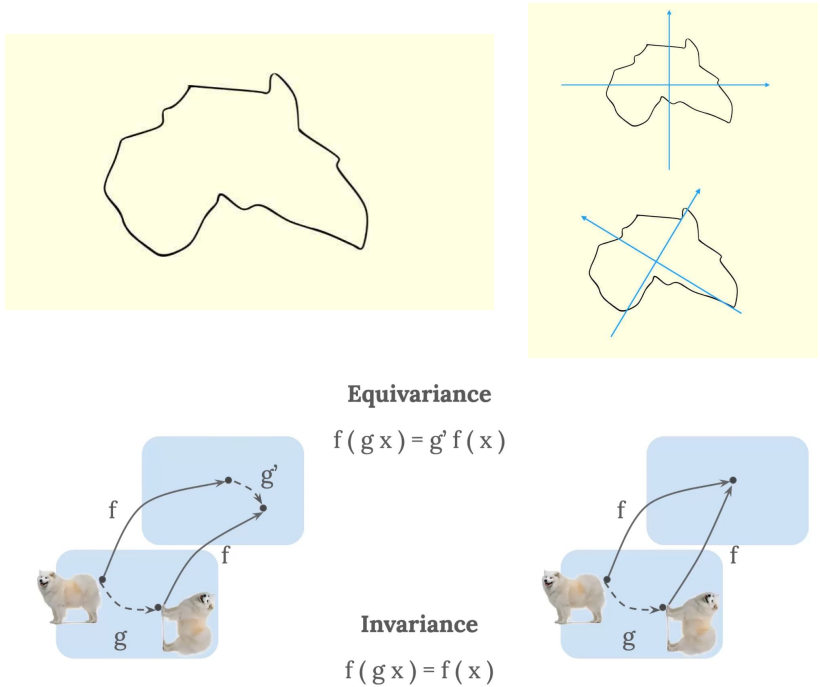


Mesh

Geometric Priors

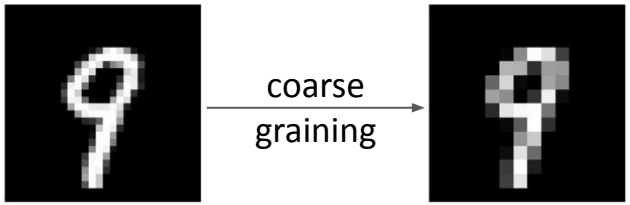
- Symmetric prior
  - Invariance and Equivariance => Learning a class of instance approximately equals to the goal of learning symmetric structures

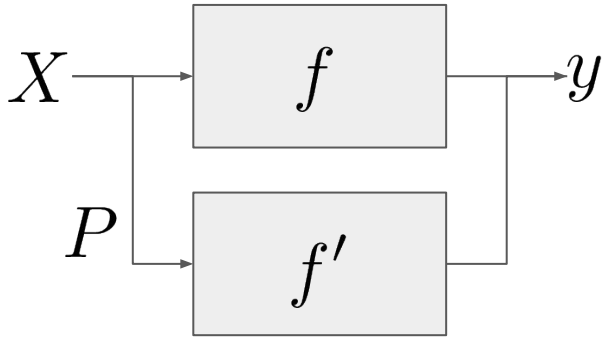




Geometric Priors

- Symmetric prior
  - Invariance and Equivariance => Learning a class of instance approximately equals to the goal of learning symmetric structures
- Scale separation prior
  - Coarsening => Local scales dominate

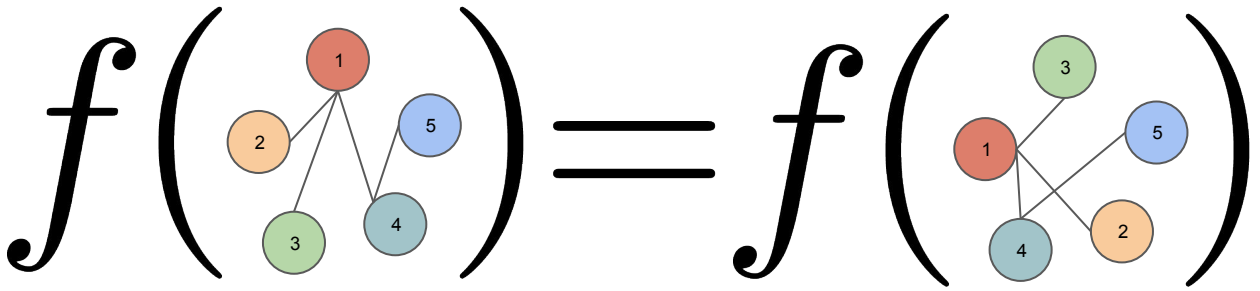




$$f \approx P f'$$

Permutation Invariant and Equivariant

The major issue of learning graphs: A graph can be mapped as different matrices.



$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \neq \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \neq \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Graph NN

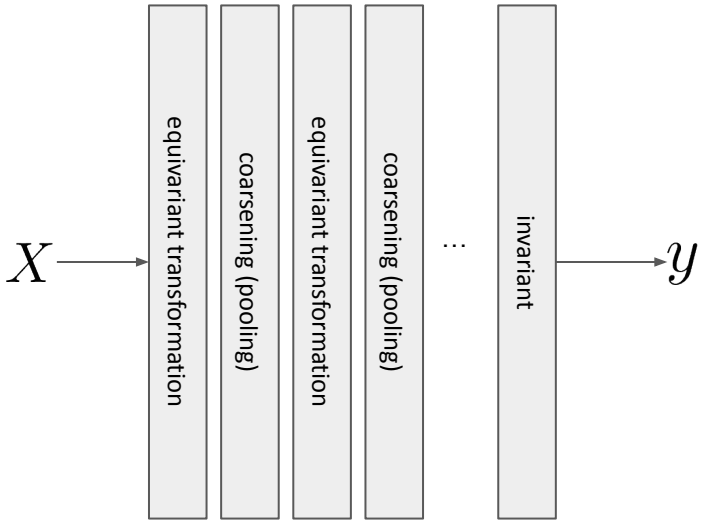
Permutations act on the edges of a graph, hence we need a proper permute both rows and columns of graph matrix A.

Invariance

$$f(PX, PAP^\top) = f(X)$$

Equivariance

$$F(PX, PAP^\top) = PF(X, A)$$



Theorem: A local function applied over all neighbourhoods ensures equivariance if the function does not depends on the order of the graph nodes (permutation invariant).

Graph NN Tasks and Layers

There are three major tasks in graph NN:

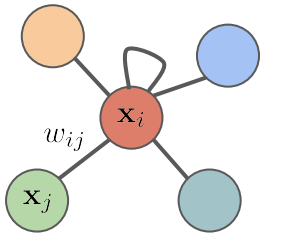
- Node classification (e.g. mesh segmentation)
- Graph classification (e.g. mesh object recognition)
- Edge prediction (e.g. infer a mesh from a point cloud)

There are also three major categories of GNN layers (increasing order of generality):

- Convolutional
- Attentional
- Message-passing

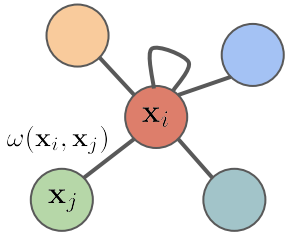
Pooling?

- Edge collapse
- Q-sim
- …?



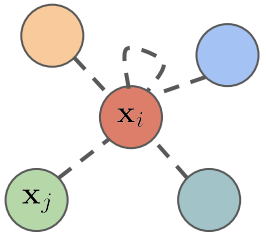
$$\mathbf{k}_i = f(\mathbf{x}_i, \bigoplus_j w_{ij} g(\mathbf{x}_j))$$

Convolutional



$$\mathbf{k}_i = f(\mathbf{x}_i, \bigoplus_j \omega(\mathbf{x}_i, \mathbf{x}_j) g(\mathbf{x}_j))$$

Attentional



$$\mathbf{k}_i = f(\mathbf{x}_i, \bigoplus_j g(\mathbf{x}_i, \mathbf{x}_j))$$

Message-passing

Laplacian (revisited)

Discrete Laplacian: uniform weights

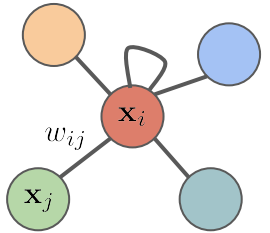
$$(\Delta \mathbf{x})_i = \sum_j (\mathbf{x}_i - \mathbf{x}_j)$$

Mesh Laplacian: could be cotan weights

$$(\Delta \mathbf{x})_i = \sum_j w_{ij} (\mathbf{x}_i - \mathbf{x}_j)$$

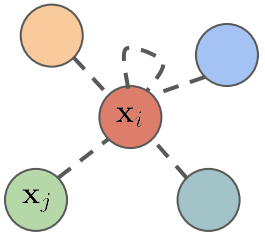
Graph Laplacian: g is local permutation-invariant

$$(\Delta \mathbf{x})_i = g(\mathbf{x}_i, \mathbf{x}_j)$$



$$\mathbf{k}_i = f(\mathbf{x}_i, \bigoplus_j w_{ij} g(\mathbf{x}_j))$$

Convolutional



$$\mathbf{k}_i = f(\mathbf{x}_i, \bigoplus_j g(\mathbf{x}_i, \mathbf{x}_j))$$

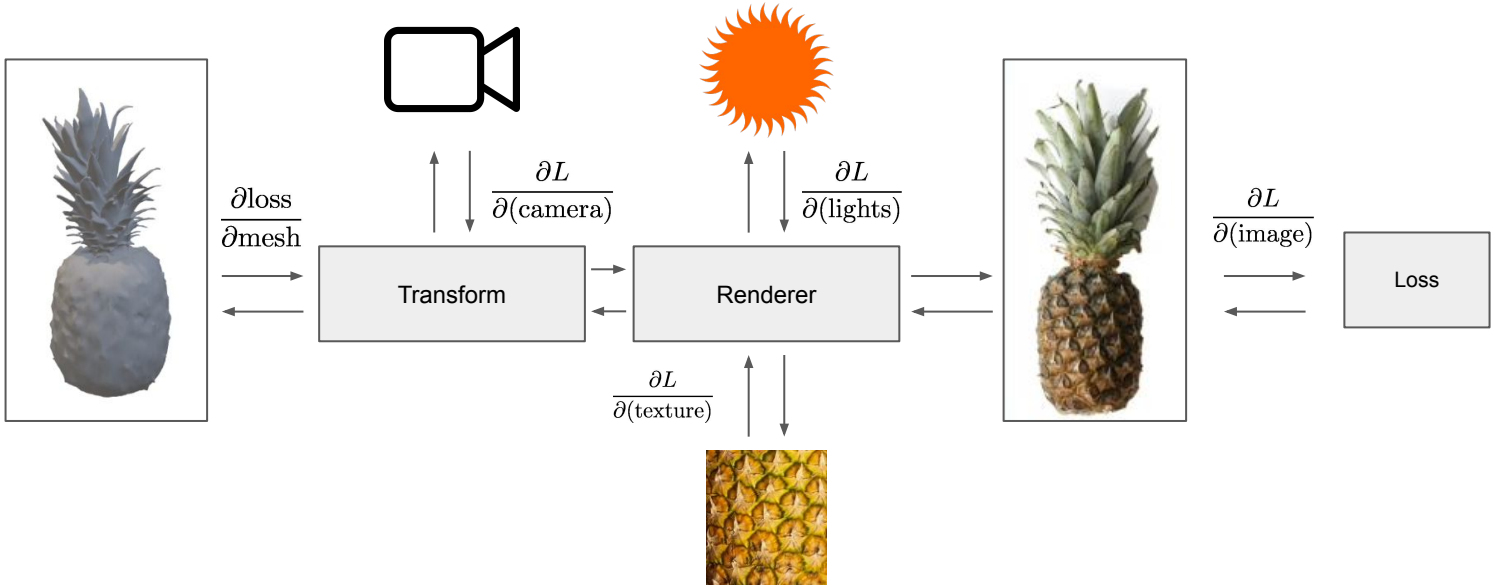
Message-passing

Session 8: Data-driven Approach (Part 2)

- Automatic Differentiation and PyTorch
- From Graphs to Manifolds
  - Permutation Invariant and Equivariant
  - General Tasks and Layers
  - Laplacian (revisited)
- Differentiable Rendering and PyTorch3D
- Summary and Outlook

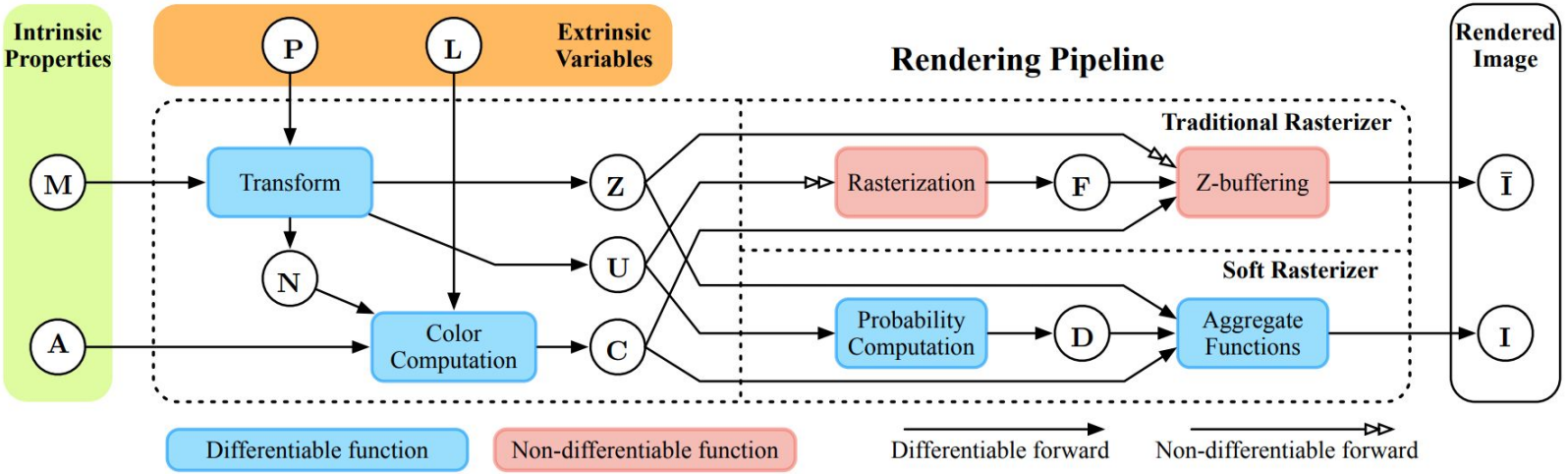
Differentiable Rendering

GNN provides the foundation of processing mesh data, depending on the goal, the network may be designed differently. In a rendering context, the output does not only depend on the mesh data, but also other rendering assets and parameters, such as camera, lights, textures. The loss is usually the *difference* of a target rendering goal, and a forward output.



Soft Rasterizer [Liu et al. 2019]

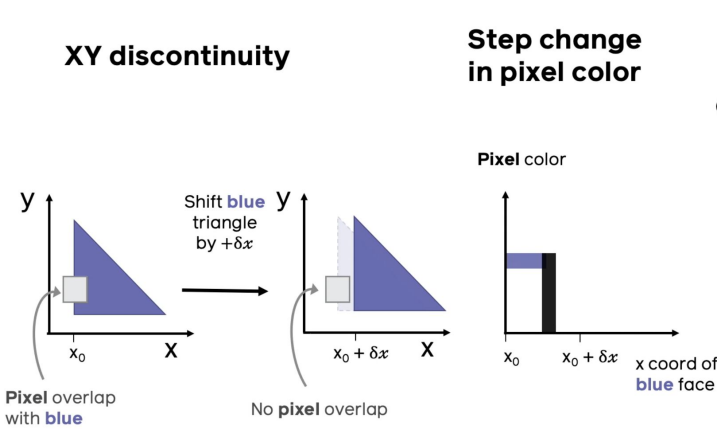
Soft Rasterizer encodes a tensor representation of a mesh **M**, per-vertex attributes **A**, camera **P**, and lights **L** to compute a fragment color **C**. The image-space coordinates **U** and depths **Z** are utilized in *aggregate function* that enables backward propagation of a rendering target **I**.

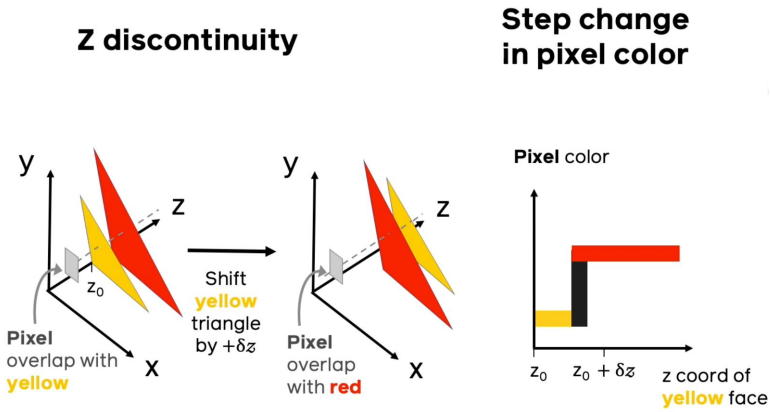


XYZ Discontinuity

Gradient can be designed to consider the changes along x, y, and z axis. But it introduces a fundamental challenge to tackle: Discontinuity, which are essentially the following two cases:

- The image-space x and y direction may not be differentiable.
- The depth buffer in z direction is also not differentiable.

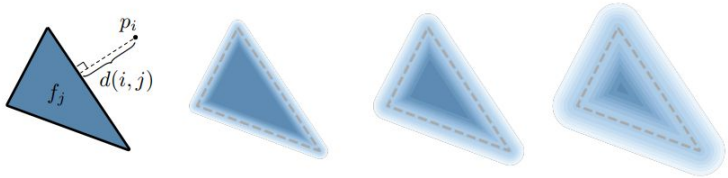


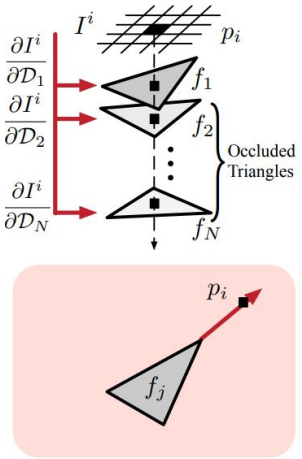


Soft Aggregation [Liu et al. 2019]

Soft aggregation does the following:

- XY discontinuity: Consider faces which fall within a blur radius
- Z discontinuity: Blend closest K faces in the depth direction



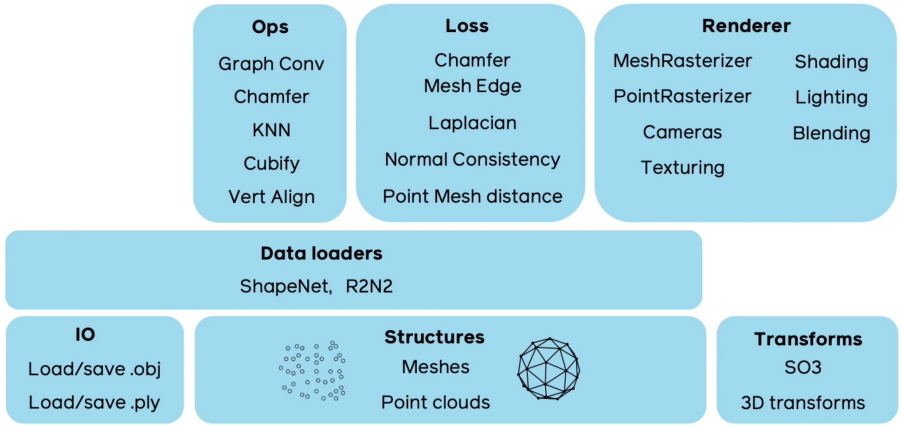


Case study: PyTorch3D

PyTorch3D is one of the solutions in processing meshes, and utilizes the SoftRasterizer approach.

Other choices, such as PyTorch Geometric, Tensorflow Graphics, Nvdiffrast, … utilizes different approaches to conduct differentiable rendering. The theory of which approach is fundamentally better is an open question as of 2021.

The overview of provided modules in PyTorch3D:

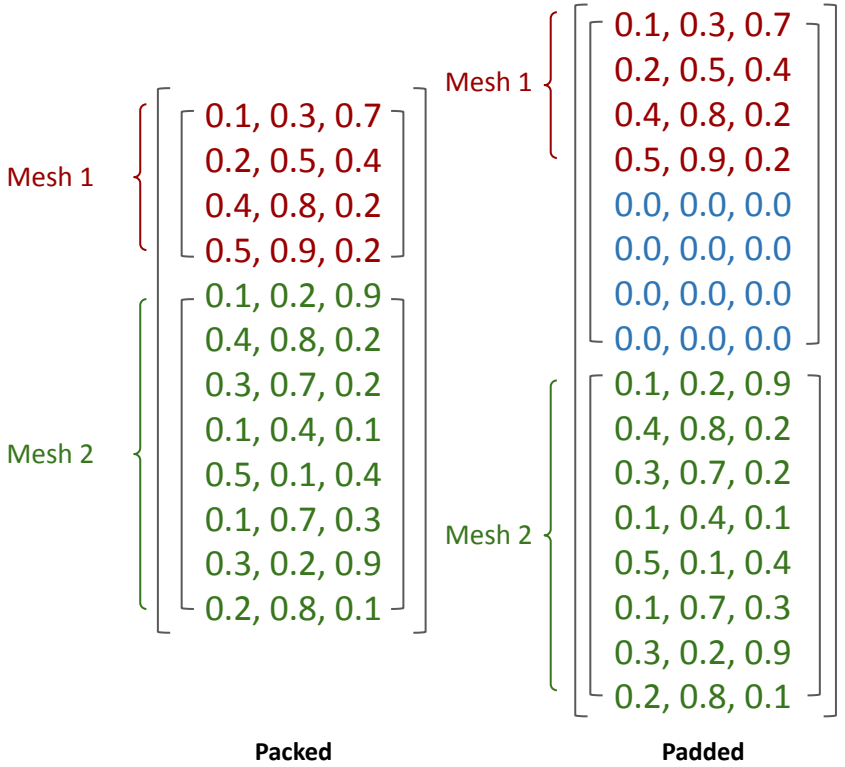


PyTorch3D: Meshes

Meshes provides *packed* or *padded* representations for vertex positions, and edge indices. Meshes includes multiple meshes as a batch.

Meshes may have different numbers of faces, vertices, and etc.

PyTorch3D internally uses padded tensors for processing.



PyTorch3D: 3D Operators and Losses

PyTorch3D only provides a GraphConv as a convolution layer. The current implementation is a uniform summation of all neighbors.

Reference: https://pytorch3d.readthedocs.io/en/latest/modules/ops.html

The current provided losses in PyTorch3D concerns these properties: point-wise distances, edge-wise distances, face-wise distances, length of edges, Laplacian smoothness, and normal consistency.

Reference: https://pytorch3d.readthedocs.io/en/latest/modules/loss.html

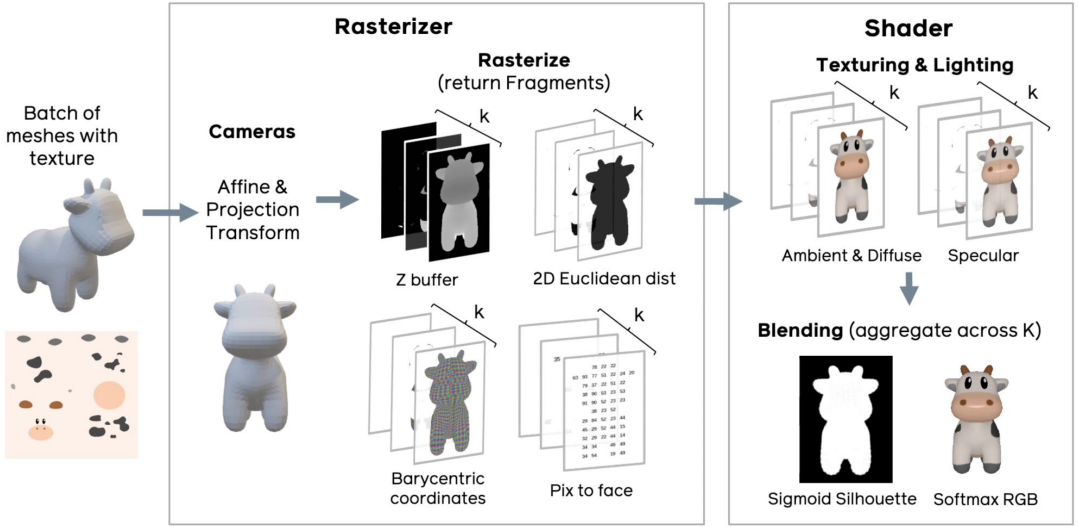
PyTorch3D: Renderer

The renderer (`MeshRasterizer`) itself has no surprises and essentially a classic hardware rasterizer. The benefit is that it can process torch.Tensor, and embeds aggregation for differentiation. The rendering engine involves:

- Separate rasterizer & shader modules
- 2 step rasterization
- Return Top K faces in Fragments
- Heterogeneous batching
- Shading in PyTorch

The other relevant classes are:

Cameras, Texturing, Shading, Lighting, Blending



Session 8: Data-driven Approach (Part 2)

- Automatic Differentiation and PyTorch
- From Graphs to Manifolds
  - Permutation Invariant and Equivariant
  - General Tasks and Layers
  - Laplacian (revisited)
- Differentiable Rendering and PyTorch3D
- Summary and Outlook

Summary and Outlook

- Computational graph and automatic differentiation are the force of neural networks
- Training a powerful network to process meshes requires permutation invariance local operators
- Differentiable rendering as one possible target application domain in utilizing mesh-based surfaces
- PyTorch as a practice of AutoDiff and PyTorch3D as an implementation of SoftRasterizer

Both *geometric deep learning* and *differentiable rendering* are under active research!

Further Reading Suggestions

[Bronstein et al. 2017] Bronstein, Michael M., et al. "Geometric deep learning: going beyond euclidean data." IEEE Signal Processing Magazine 34.4 (2017): 18-42.

[Bronstein et al. 2017] Bronstein, Michael M., et al. "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges." arXiv preprint arXiv:2104.13478 (2021).

[Loper et al. 2014] Loper, Matthew et al. "OpenDR: An approximate differentiable renderer." *European Conference on Computer Vision*. Springer, Cham, 2014.

[Kato et al. 2018] Kato, Hiroharu et al. "Neural 3d mesh renderer." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018.

[Li et al. 2018] Li, Tzu-Mao et al. "Differentiable monte carlo ray tracing through edge sampling." *ACM Transactions on Graphics (TOG)* 37.6 (2018): 1-11.

[Liu et al. 2019] Liu, Shichen, et al. "Soft rasterizer: A differentiable renderer for image-based 3d reasoning." *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019.

[Chen et al 2019] Chen, Wenzheng, et al. "Learning to predict 3d objects with an interpolation-based differentiable renderer." *Advances in Neural Information Processing Systems* 32 (2019): 9609-9619.

Open Positions

- Work as a tutor in Computer Graphics 1
  - Teaching is a further step of learning

- An *Einzelpraktikum* or a *Thesis* in this area
  - Feel free to contact me :)