6.7900 Fall 2024: Lecture Notes 12

1 Motivation

We have all kinds of data in this world that we might wish to model and learn from, spanning images, text, audio, social networks, sensor networks, traffic patterns, and more. Choosing a suitable neural network architecture for a given data modality involves considerations that can dramatically affect both performance and tractability. Key considerations include:

1. **Fitting inductive biases and equivariances to the problem**: We often want architectures that naturally respect the symmetry or structure of the data. For example, images have a 2D grid structure with translational symmetries, while graphs have permutation-invariant node orderings.
2. **Computational tractability in forward and backward passes**: The model needs to scale well with input size and allow efficient backpropagation. Dense, fully-connected architectures on large inputs (like $N \times N$ images) can be prohibitively expensive.
3. **Input/Output compatibility**: The network should be able to ingest data in a form that respects the domain's inherent structure (e.g., grid-like for images, graph structure for networks) and produce outputs of a compatible form.
4. **Favorable optimization dynamics**: Certain architectures and parameterizations lead to smoother optimization landscapes or are known to converge more reliably, which can be crucial in practice.

Consider the task of processing high-resolution images (size $N \times N$). Using a Multilayer Perceptron (MLP) would imply having on the order of $O(N^2)$ input dimensions, which could make the model extremely large. Additionally, MLPs lack intrinsic notions of translation or scale invariance, and they struggle to process variable-size structured inputs.

Limitations like these motivated the development of specialized architectures:

Convolutional Neural Networks (CNNs): Designed to exploit the 2D grid structure of images, enabling parameter sharing and translation equivariance. **Graph**

Neural Networks (GNNs): Designed to operate on graphs, respecting their inherent permutation invariance and making it natural to process data with arbitrary topology and varying size.

In this lecture, we will explore both CNNs and GNNs, understanding how their architectural design choices overcome the drawbacks of naive MLP approaches.

2 Convolutional Neural Networks (CNNs)

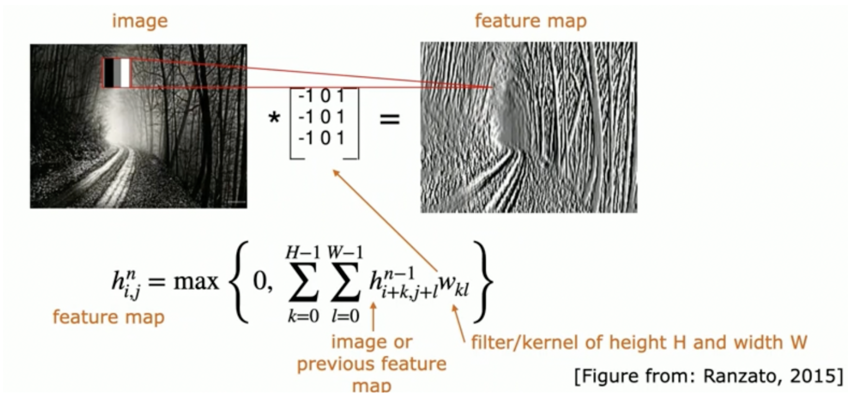


image
feature map
-1 0 1
-1 0 1
-1 0 1
$h^n_{i,j} = \max\left\{0, \sum_{k=0}^{H-1}\sum_{l=0}^{W-1} h^{n-1}_{i+k,j+l} w_{kl}\right\}$
feature map
image or previous feature map
filter/kernel of height H and width W
[Figure from: Ranzato, 2015]

Figure 1: A convolutional kernel scanning across an image.

2.1 Definition

A Convolutional Neural Network leverages the concept of convolutional filters (kernels) that slide over the spatial dimensions of the input. Rather than having fully-connected layers that learn a unique weight for every input pixel, CNNs learn small filters that capture local patterns. These filters are repeatedly applied across the image, allowing the model to be:

- Translation-equivariant: A feature learned at one location can be detected at any other location.
- Parameter-efficient: The same filter is reused across the entire image, drastically reducing the number of parameters relative to a fully-connected layer.

Mathematically, for a given layer, a CNN performs (see Figure 1):

$$\mathbf{h}_{ij}^{(l)} = \sigma\left(\sum_{u,v} \mathbf{W}_{uv}^{(l)} \cdot \mathbf{x}_{i+u,j+v}^{(l-1)} + b^{(l)}\right)$$

where $\mathbf{W}_{uv}^{(l)}$ is the convolutional kernel, $\sigma(\cdot)$ is a nonlinearity, and (i, j) indexes the spatial location in the output feature map.

2.2 Features CNNs Learn

As we move deeper into a CNN, the learned filters tend to progress from capturing simple, low-level features (edges, corners, textures) in early layers to more complex, high-level features (parts of objects, entire objects) in later layers. These layers combine and pool information from earlier ones, building up hierarchical representations of the input. We can visualize the features learned at different layers of a CNN as shown in Figure 2.

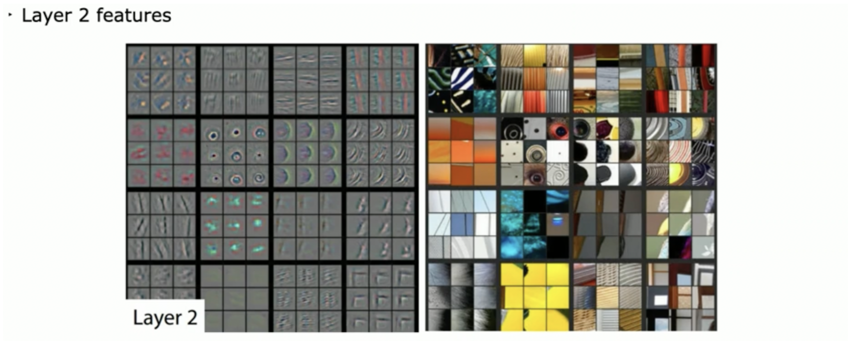


Figure 2: Convolutional kernels visualized from an intermediate layer and the dataset images that strongly activate them.

Clarification on Feature Visualization: There was some confusion in lecture about whether these visualized features perfectly capture what causes certain activations. In fact, these visualizations are created through techniques like deconvolution networks or gradient-based optimization to find inputs that strongly activate certain neurons. As explained in the method section of the paper [Zeiler & Fergus, arXiv:1311.2901], these visualizations are not guaranteed to be unique or exact. The non-bijective nature of deconvolution and other interpretability methods means these images are suggestive "prototypes," not definitive explanations.

2.3 CNN History and Advances

CNNs were first introduced in the late 20th century, with a famous early application by Yann LeCun on the MNIST digit classification problem (LeNet-5 architecture). Despite their promise, CNNs were not widely used until the resurgence of deep learning methods in the 2010s. The breakthrough came with AlexNet (Krizhevsky

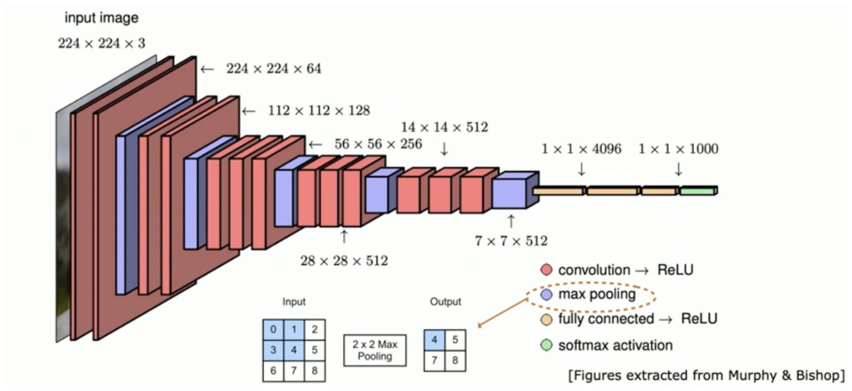
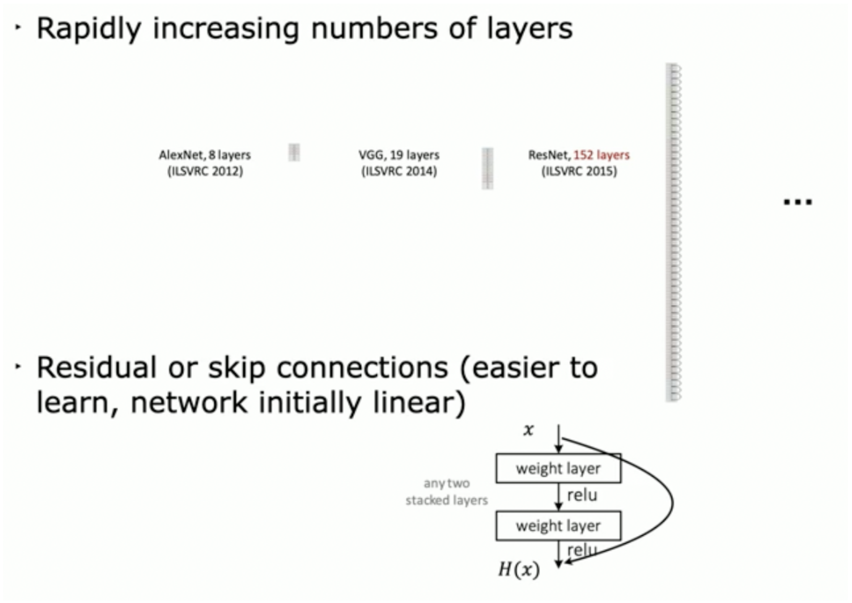


Figure 3: AlexNet, an early CNN architecture that sparked the deep learning revolution in computer vision.



- Rapidly increasing numbers of layers
- Residual or skip connections (easier to learn, network initially linear)

Figure 4: Over time, CNNs have become deeper and more accurate, as shown by models like VGG, Inception, and ResNet on benchmarks like ImageNet.

et al., 2012), which won the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) by a large margin(see Figure 3) Since then, CNNs have evolved rapidly (see Figure 4):

- Deeper Architectures: VGG, Inception, ResNet introduced increasingly deep and sophisticated architectures.

- Better Regularization: Dropout, batch normalization, and improved optimization methods.
- Architectural Innovations: Residual connections, attention mechanisms, and more to ease training and improve representational power.

3 Graph Neural Networks (GNNs)

While CNNs excel at processing grid-structured data, many real-world datasets are represented as graphs: social networks, traffic networks, molecular structures, knowledge graphs, and more. Such data often does not have a regular spatial arrangement. Instead, we have nodes (entities) and edges (relationships), and we want architectures that:

- Are permutation-invariant: The ordering of nodes should not affect the output.
- Are scalable to variable-sized inputs: Graphs can grow or shrink, and we need to handle different shapes.
- Leverage topological information: Patterns in how nodes are connected are crucial.

GNNs are designed to handle these requirements naturally.

3.1 General GNN Idea

A GNN updates node representations by aggregating features from their neighbors. Consider a graph $G = (V, E)$ with node features $\mathbf{x}_v$ for $v \in V$ and potentially edge features $\mathbf{e}_{uv}$ for $(u, v) \in E$. A single GNN layer can be summarized as (see also Figures 5, 6):

- Aggregate: Each node v collects information from its neighbors $\mathcal{N}(v)$

$$\mathbf{m}_v^{(l)} = \text{AGGREGATE}(\{\mathbf{h}_u^{(l-1)}, \mathbf{e}_{uv} : u \in \mathcal{N}(v)\})$$

- Combine: Each node updates its own representation based on the aggregated messages

$$\mathbf{h}_v^{(l)} = \text{COMBINE}(\mathbf{h}_v^{(l-1)}, \mathbf{m}_v^{(l)})$$

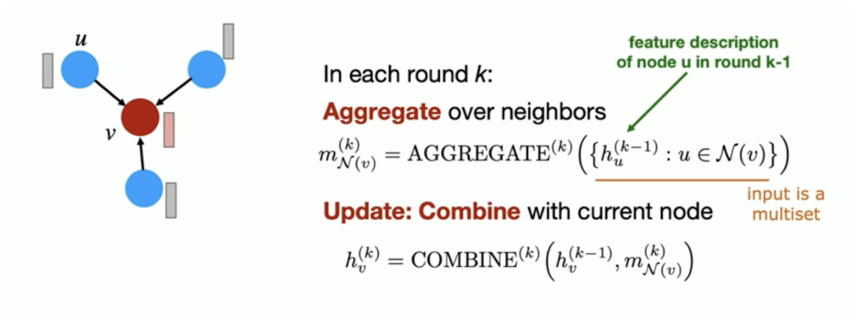

u
v
In each round k:
feature description of node u in round k-1
Aggregate over neighbors
$m_{\mathcal{N}(v)}^{(k)} = \text{AGGREGATE}^{(k)}\left(\{h_u^{(k-1)} : u \in \mathcal{N}(v)\}\right)$
input is a multiset
Update: Combine with current node
$h_v^{(k)} = \text{COMBINE}^{(k)}\left(h_v^{(k-1)}, m_{\mathcal{N}(v)}^{(k)}\right)$

Figure 5: General scheme of GNN message passing and update.

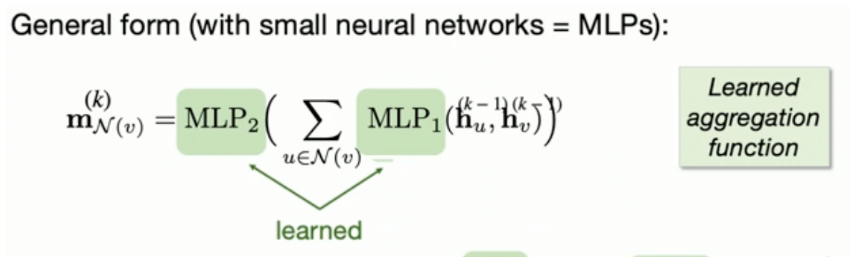

General form (with small neural networks = MLPs):
$\mathbf{m}_{\mathcal{N}(v)}^{(k)} = \text{MLP}_2\left(\sum_{u \in \mathcal{N}(v)} \text{MLP}_1(\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)})\right)$
Learned aggregation function
learned

Figure 6: More detailed breakdown of how node embeddings are updated using neighbor embeddings.

After several such layers, the final node embeddings $\mathbf{h}_v^{(L)}$ can be used for tasks like node classification, or pooled to form a graph-level representation for graph classification.

Exercise: If you wanted to incorporate edge features in the update scheme, how would you do it?

Hint: Incorporating edge features typically means augmenting the aggregation function with edge-specific information. For example, when collecting messages from neighbors $u \in \mathcal{N}(v)$, you could combine both the neighbor's node embedding $\mathbf{h}_u^{(l-1)}$ and the edge embedding $\mathbf{e}_{uv}$ via a learned function:

$$\mathbf{m}_v^{(l)} = \sum_{u \in \mathcal{N}(v)} \phi(\mathbf{h}_u^{(l-1)}, \mathbf{e}_{uv})$$

where ϕ is a neural network that fuses node and edge features.

3.2 Are All GNNs Convolutional?

Classical GNNs, often called Graph Convolutional Networks (GCNs), draw analogy to convolutions by performing a weighted aggregation of neighbor features similar to a convolution operation on irregular domains. However, training very deep GCNs often leads to ”oversmoothing,” where node embeddings converge to similar values and lose discriminative power.

Not all GNN variants use a strictly ”convolutional” approach. Some GNN designs leverage different mechanisms:

- Random Walks or Diffusions: Methods like GraphSAGE, or those that incorporate personalized PageRank, use random walk distributions to gather information from the graph.
- Attention Mechanisms (GAT): Assign different importances to different neighbors rather than a uniform or fixed weighting.
- Message Passing Variants: Many models explore different aggregation functions (max, mean, sum, MLP-based) or learn different types of edge transformations.

The key takeaway: While the “graph convolution” idea is common, the field of GNNs is broader and includes many designs that do not strictly resemble classical CNN-style convolutions.

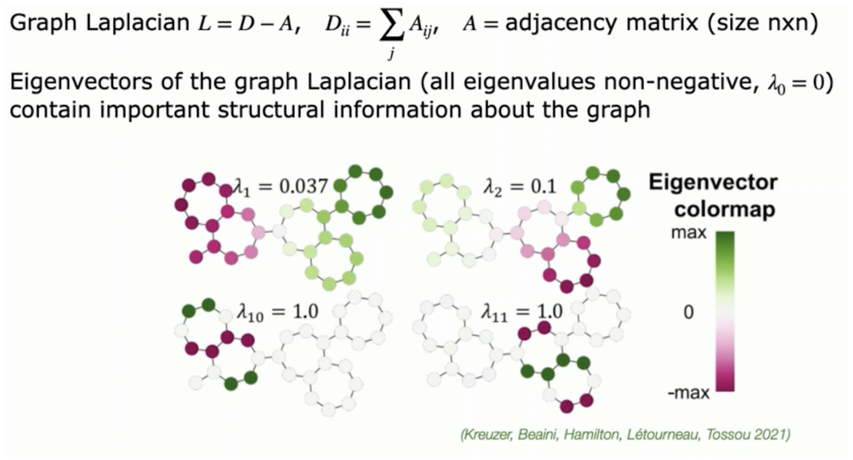

Graph Laplacian $L = D - A$, $D_{ii} = \sum_j A_{ij}$, A = adjacency matrix (size nxn)
Eigenvectors of the graph Laplacian (all eigenvalues non-negative, $\lambda_0 = 0$) contain important structural information about the graph
$\lambda_1 = 0.037$
$\lambda_2 = 0.1$
$\lambda_{10} = 1.0$
$\lambda_{11} = 1.0$
Eigenvector colormap
max
0
-max
(Kreuzer, Beaini, Hamilton, Létourneau, Tossou 2021)

Figure 7: Illustration of how graph eigenvectors can differentiate regions of a graph, giving a positional reference frame.

3.3 Graph Positional Encodings

Graph structure can also be encoded using positional information. Unlike grids where pixel coordinates are explicit, nodes in graphs do not come with a natural positional embedding. One approach is to use the eigenvectors of the graph Laplacian (the matrix Λ derived from the adjacency matrix and degree matrix); see Figure 7. These eigenvectors, called Laplacian eigenvectors, can serve as a "positional basis" that provides a notion of global and local structure. The eigenvalues and eigenvectors capture fundamental properties of the graph's topology, and encoding each node's coordinates along these eigenvectors can provide a powerful topological signature.

Such positional encodings can help GNNs overcome some of their inherent permutation invariance by providing a shared frame of reference, often improving performance on tasks that rely on global structural understanding.