ELEC 5680: Advanced Deep Learning Architecture

Hong, Lanxuan

Spring 2025

This is a lecture note for COMP 5214: Advanced Deep Learning Architecture.

- COMP 5214: website, my-project. **all topics**
- Dive into Deep Learning: website, Chinese version. **all topics covered**
- MIT 6.S191 Deep Learning: website, video. **topic 2-4: intro, Seq, CNN**
- UCB STAT157: website
- Next step of ML: video
- cs231n: notes **topic 5-6: CNNs, classification, object detection**
- cs224n: website **topic 7-8: NLP, transformer**
- cs224w: website **topic 11-12: GNN**

Last updated on Sunday 14th September, 2025.

Contents

1 From Perceptron to Neural Network 4

2 Training a Neural Network 5

2.1 Initialization . . . 5
2.2 Optimization . . . 6
2.3 Regularization . . . 6
2.3.1 L1 Regularization . . . 7
2.3.2 L2 Regularization . . . 7
2.3.3 Batch Normalization . . . 7
2.3.4 Dropout . . . 8
2.4 Loss Function . . . 9
2.4.1 Cross-Entropy Loss . . . 9
2.4.2 MSE Loss . . . 9
2.5 Hyperparameter Tuning . . . 9

3 Deep Sequence Modeling 11

3.1 Recurrent Neural Network(RNN) . . . 11
3.2 Long Short Term Memory . . . 13
3.3 Attention Mechanism . . . 14

4 Convolutional Neural Network 15

4.1 Basic structure and components . . . 15
4.2 Training a CNN . . . 15
4.2.1 Data Augmentation . . . 15
4.2.2 Pooling . . . 15
4.2.3 Convolution . . . 16
4.2.4 Residual Connection . . . 17
4.2.5 Advanced Techniques . . . 17
4.3 Image Classification . . . 18
4.4 Object Detection . . . 19

5 NLP with Deep Learning 21

5.1 Word Embeddings . . . 21
5.2 Seq2Seq Model . . . 22
5.3 Transformer . . . 23

5.3.1 Self-Attention and Encoder-Decoder Structure 23
5.3.2 Transformer Variants 25

6 Generative Model **26**
6.1 Variational Autoencoders(VAE) 26
6.2 Generative Adversarial Network(GAN) 26
6.3 Flow-based Deep Generative Models 27

7 Graph Neural Network(GNN) **27**
7.1 Graph Embeddings 27
7.1.1 RandomWalk Embedding 28
7.1.2 Node2vec Embedding 28
7.1.3 Graph Convolutional Network 29
7.2 A General GNN Framework 30
7.2.1 A single GNN layer 31
7.2.2 Stacking GNN Layers 32
7.2.3 Graph Manipulation 32
7.3 GNN Training Pipeline 32
7.3.1 Prediction 32
7.3.2 Loss Function and Evaluation 33
7.3.3 Split Training 33
7.4 Scale Up GNN 33

8 Foundation Models: LLM **33**

1 From Perceptron to Neural Network

Perceptron is the structural building block of deep learning. The mechanism of a perceptron is to take the linear combination of weighted inputs and get the predicted output $\hat{y}$ from a non-linear activation function.

$$\hat{y} = g\left(\sum_{i=1}^{m} x_i w_i + b\right) = g \cdot \left(b + XW^T\right)$$

Activation functions such as **sigmoid**, **hyperbolic tangent(tanh)**, and **Rectified Linear Unit(ReLU)** introduce non-linearity to the prediction result. Different activation functions can be applied on different layers and even different neurons on the same layer.

We can predict more than 1 output by **multi-output perceptron** where all inputs are densely connected **(dense layers)** to all outputs.

$$\hat{y}_i = g\left(\sum_{j=1}^{m} x_j w_{j,i} + b_i\right) = g \cdot \left(b_i + XW_i^T\right)$$

We can also add **hidden layer** between the input layer and output layer to observe a more complicated relationship. This builds the fundamental **neural network**. This process of calculating predicted output by weights of each dense layer is **forward propagation**.

$$\text{Hidden Layer: } Z = b^{(1)} + XW^{(1)T}$$
$$\text{Output Layer: } \hat{Y} = g\left(b^{(2)} + g(Z)W^{(2)T}\right)$$

A neural network with hidden layers cannot be directly "trained" because the weights are not explicitly adjusted based on the output. Instead, they are updated indirectly through **backward propagation**, which adjusts weights based on the **loss (the cost incurred from incorrect prediction)** of the network.

$$\text{Binary Cross Entropy Loss: } \mathcal{L}(\hat{y}_i, y_i) = -(y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i))$$
$$\text{Mean Squared Error Loss: } \mathcal{L}(\hat{y}_i, y_i) = (y_i - \hat{y}_i)^2$$

The training process of a neural network is to find and update network weights

to optimize the loss. One common way to update the weights to minimize loss is through **gradient descent**.

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

$$W^* = \arg\min_W j(W) \qquad W = W^0, W^1, \ldots W^n$$

$$\text{gradient descent: } W \leftarrow W - \eta\frac{\partial J(W)}{\partial W} \qquad \textbf{update until converges}$$

The optimization in neural network training is more challenging. The loss landscape of neural networks is very complicated, which makes it difficult to optimize loss functions. One smart idea is to create an adaptive learning rate η (e.g. SGD, Adam) to try for a better loss. Another smart idea is to pick a mini-batch of data points for gradient calculation. Mini-batches allow smoother convergence and are faster and more accurate.

2 Training a Neural Network

2.1 Initialization

Proper initialization is crucial for avoiding issues like vanishing or exploding gradients and ensuring efficient training.

1. **Weight Initialization**:

   - **Symmetry Breaking:** Initializing all weights to the same value (e.g., zero) leads to identical gradients for all neurons in a layer, resulting in no diversity in learning. To break symmetry, weights are initialized randomly.
   - **Xavier Initialization:** To maintain a stable variance of activations throughout the network, Xavier initialization sets:

     $$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{\text{fan\_in} + \text{fan\_out}}}, \sqrt{\frac{6}{\text{fan\_in} + \text{fan\_out}}}\right),$$

     where fan_in and fan_out are the number of input and output connections to the neuron. This ensures that the variance of activations remains

constant across layers.

- **He Initialization:** For ReLU-based networks, He initialization is more suitable:
$$W \sim \mathcal{N}(0, \frac{2}{\text{fan\_in}}).$$
This accounts for the fact that half of the ReLU activations are zero.

2.2 Optimization

Optimization aims to minimize the loss function $\mathcal{L}(\theta)$ by updating the model parameters θ:

- **Mini-batch Stochastic Gradient Descent (SGD):**
$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta),$$
where η is the learning rate. Mini-batch SGD uses a small subset of training data to approximate the gradient, reducing computation while introducing stochasticity.
- **Variants of SGD:**
  - **Momentum:** Adds a velocity term to smooth updates:
$$v \leftarrow \mu v - \eta \nabla_\theta \mathcal{L}(\theta), \quad \theta \leftarrow \theta + v,$$
where μ is the momentum coefficient.
  - **Adam:** Combines momentum and adaptive learning rates:
$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta \mathcal{L}(\theta), \quad v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta \mathcal{L}(\theta))^2,$$
$$\theta \leftarrow \theta - \eta \frac{m_t}{\sqrt{v_t} + \epsilon}.$$

2.3 Regularization

Regularization mitigates overfitting by discouraging overly complex models that fit the noise in the training data. It works by introducing additional constraints or penalties to the learning process. Common regularization techniques include:

- Use higher dropout rates (e.g., $p = 0.5$) for fully connected layers and lower rates (e.g., $p = 0.2$) for convolutional layers.
- Dropout is less effective for batch-normalized networks since batch normalization already provides some regularization.
- **Early Stopping:** prevents overfitting by limiting the model's capacity to fit noise in the training data.

2.3.1 L1 Regularization

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \lambda \|W\|_1,$$

where $\|W\|_1 = \sum_{i,j} |W_{i,j}|$. Unlike ℓ_2, ℓ_1 regularization encourages sparsity in the weights, meaning many weights are driven to zero. This can lead to more interpretable models and is especially useful in feature selection tasks.

2.3.2 L2 Regularization

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \lambda \|W\|^2,$$

where λ controls the strength of regularization, $\mathcal{L}_{\text{data}}$ is the data loss, and $\|W\|^2 = \sum_{i,j} W_{i,j}^2$ is the squared Frobenius norm of the weights. This penalizes large weights, encouraging the model to find simpler solutions. Theoretical justification stems from the bias-variance tradeoff: ℓ_2 regularization reduces variance at the cost of increased bias, which can improve generalization.

2.3.3 Batch Normalization

In training process, the distribution of activations in intermediate layers changes during training. This shift slows convergence as layers need to continuously adapt to new distributions. **Batch normalization** normalize the activation input x over a mini-batch so that it can locate in the sensitive region of activation function.

$$\hat{x}_i = \frac{x_i - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}},$$

where $\mathbb{E}[x]$ and $\text{Var}[x]$ are the mini-batch mean and variance, and ϵ is a small constant for numerical stability.

Then scales and shifts the normalized activations:

$$y_i = \gamma \hat{x}_i + \beta,$$

where γ and β are learned parameters that allow the network to recover the original representation. For example:

- Consider a sigmoid activation $g(z) = \frac{1}{1+e^{-z}}$. If z has a large variance, a significant portion of z values will fall into the saturated regions of the sigmoid, where the gradient $g'(z)$ approaches zero. Normalizing z ensures that its distribution remains closer to zero, avoiding saturation and improving gradient flow.
- Empirical results from Ioffe and Szegedy's paper demonstrate that BN allows networks to converge up to 14 times faster on ImageNet classification tasks compared to networks without BN.

2.3.4 Dropout

Dropout randomly sets a fraction p of activations to zero during training, effectively preventing neurons from co-adapting. Mathematically, for a given layer output $\mathbf{h}$, dropout applies:

$$\tilde{\mathbf{h}} = \mathbf{h} \odot \mathbf{r}, \quad r_i \sim \text{Bernoulli}(1-p),$$

where $\odot$ is the element-wise product and $\mathbf{r}$ is a mask vector. Dropout acts as a form of model averaging by training an ensemble of subnetworks, improving robustness and reducing overfitting.

2.4 Loss Function

2.4.1 Cross-Entropy Loss

Cross-entropy loss is widely used for classification tasks. For a dataset with N samples and C classes, the loss is defined as:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log p_{ij},$$

where $y_{ij} \in \{0,1\}$ is the ground truth label (one-hot encoded) and p_{ij} is the predicted probability for class j of sample i. Minimizing cross-entropy ensures that the predicted probabilities p_{ij} approximate the true labels y_{ij}. It is particularly effective when the outputs are soft probabilities (e.g., from a softmax layer).

2.4.2 MSE Loss

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2,$$

where y_i is the true value and $\hat{y}_i$ is the predicted value. MSE penalizes larger errors more heavily, encouraging the model to focus on minimizing significant deviations.

2.5 Hyperparameter Tuning

- **Tuning Techniques:**

    1. **Grid Search:** Exhaustively searches over a predefined set of hyperparameter values. While computationally expensive, it guarantees exploration of all combinations.
    2. **Random Search:** Samples hyperparameter values randomly from a distribution. This is often more efficient than grid search.
    3. **Bayesian Optimization:** Models the objective function (e.g., validation error) as a probabilistic function and selects hyperparameters by maximizing the expected improvement.
    4. **Hyperband:** Combines random search with early stopping to efficiently allocate resources to promising configurations.

- **Cross-Validation:** Use k-fold cross-validation to evaluate hyperparameters. Split the training dataset into k subsets, train on $k-1$ subsets, and validate on the remaining subset. Compute the average validation error across folds to select the best hyperparameter configuration.
- **Empirical Guidelines:**
  - Start with a small learning rate (e.g., 0.001) and increase it if training converges too slowly.
  - Use batch sizes within the range of 32 to 512, depending on the available memory.
  - Regularization strength (λ) typically varies between 10^{-4} and 10^{-1}.

3 Deep Sequence Modeling

To observe the patterns of sequential data, we need to predict the current output not only on current input but also on past inputs. To learn the inherent structure from the history, we cannot treat the input-output network by each time stamp. A neural network storing historical information while can be updated by latest data input is required. Some design criteria of sequence modeling:

1. it should be adaptive to embedded encoding, e.g. one hot encoding
2. it can handle **variable-length** sequences
3. it can track **long-term** dependencies
4. it maintains information about **order**
5. it **share parameters** across the sequence

3.1 Recurrent Neural Network(RNN)

One idea is to introduce a memory variable h which pass the states of neural networks in each time stamp in a structure called **recurrent cells**. The whole structure, known as **Recurrent Neural Network(RNN)**, updates the state h and then apply the recurrence relation at every time step to process a sequence.

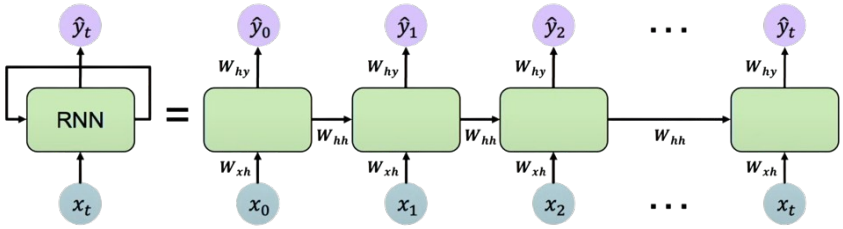


Figure 1: RNN Structure

$$h_t = f_W(x_t, h_{t-1}) \qquad \text{same function } f \text{ and same weights } W$$

$$x_t \qquad \text{input vector}$$

$$h_t = tanh\left(W_{hh}^T h_{t-1} + W_{xh}^T x_t\right) \qquad \text{update hidden state by tanh and } W$$

$$\hat{y}_t = W_{hy}^T h_t \qquad \text{output vector}$$

Note ⚠ The same weight matrix is re-used at every time stamp. W_{xh} loads inputs to recurrent state, and W_{hh} is shows state transition of h_{t-1}. The combination of state transition result and load input is applied to the same activation function to get current state. W_{ht} shows state-to-output transformation.

To train the model and adjust weight function, we need backward propagation process through time(**BPTT**).

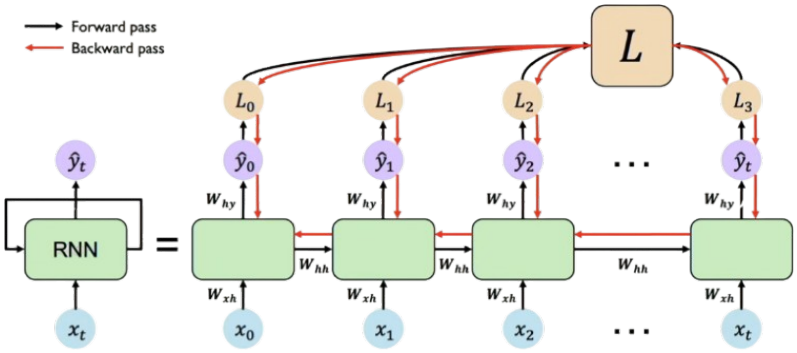


Figure 2: Backward Propagation of RNN

Note ⚠ why gradient vanishing? which weight is effected?

For a fixed h_0 and loss function $\mathcal{L}_t = \frac{1}{2}(\hat{y}_t - y_t)^2$, the backward propagation process is as follows.

$$W_{hy} \leftarrow W_{hy} - \frac{\partial \mathcal{L}}{\partial W_{hy}} = W_{hy} - \sum_t \frac{\partial \mathcal{L}_t}{\partial W_{hy}}$$

$$\frac{\partial \mathcal{L}_t}{\partial W_{hy}} = \frac{\partial \mathcal{L}_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial W_{hy}}$$

not affected by deeper layers

$$W_{xh} \leftarrow W_{xh} - \frac{\partial \mathcal{L}}{\partial W_{xh}} = W_{xh} - \sum_t \frac{\partial \mathcal{L}_t}{\partial W_{xh}}$$

$$\frac{\partial \mathcal{L}_t}{\partial W_{xh}} = \frac{\partial \mathcal{L}_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial h_t} \cdot \partial(f) \cdot \frac{\partial \left(W_{hh}^T h_{t-1} + W_{xh}^T x_t \right)}{\partial W_{xh}}$$

$\frac{\partial h_t}{\partial W_{xh}}$ **is**

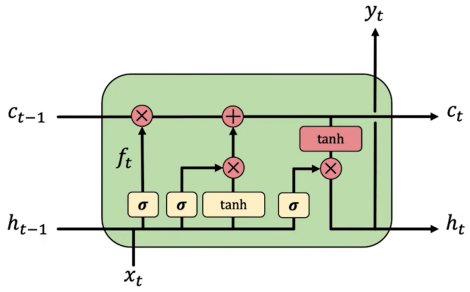
Several solutions include:

1. choose activation function wisely. ReLU is often selected to avoid gradient vanishing since its derivative is 1 when positive.

2. initialize weights wisely, often to 1 to prevent weights from shrinking to 0.
3. Build **gated cells** to selectively add or remove information within each recurrent unit.

3.2 Long Short Term Memory

The **Long short Term Memory(LSTM)** network relies on a gated cell to track information throughout many time stamps. It maintains a cell state and use gates to control the flow of information: information is **added** or **removed**



LSTM Gate Structure

- Forget irrelevant history by f_t
- Store relevant new information into cell state from current input
- Selectively update cell state by c_{t-1}
- output information to next time step

$$\text{Forget step: } f_t = \sigma\left(W_i[h_{t-1}, x_t] + b_f\right)$$
$$\text{Store step: } i_t = \sigma\left(W_i[h_{t-1}, x_t] + b_i\right)$$
$$\tilde{C} = \tanh\left(W_c[h_{t-1}, x_t] + b_c\right)$$
$$\text{Update step: } c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c_t}$$
$$\text{Output step: } o_t = \sigma\left(W_o[h_{t-1}, x_t] + b_o\right)$$
$$h_t = o_t \cdot \tanh c_t$$

Note ⚠ Why is LSTM a more robust model in terms of gradient?

not finished!
not finished!
not finished!
not finished!
A good note about RNN and LSTM, inlcuding gradient descent. here

3.3 Attention Mechanism

The intuition behind self-attention is to attend the most important parts of an input and extract the features with high attention. The following steps learn self-attention with neural networks. **self-attention can be applied in parallel, which is faster than RNN model while capturing long-time information**

1. fed in data at once and encode **position** information.

$$a^i = Wx^i$$

2. extract **query**(to match), **key**(to be matched), and **value**(info to be extracted).

$$q^i = W^q a^i$$
$$k^i = W^k a^i$$
$$v^i = W^v a^i$$

3. compute scaled dot-product attention and normalize through softmax layer.

$$\alpha_{1,i} = \frac{q^1 \cdot k^i}{\sqrt{d}} \Rightarrow \hat{\alpha_{1,i}} = \text{softmax}\alpha_{1,i}$$

4. output

$$b_1 = \sum_i \hat{\alpha_{1,i}} v^i$$

The attention mechanism is the foundational building block of the **transformer architecture**.

4 Convolutional Neural Network

4.1 Basic structure and components

To learn features in a 2-D image by neural network while maintain its spatial invariance, we can connect patches of input to neurons in hidden layer. Neuron connected to region of input only "sees" the values in that region. This spatial structure can be applied to the whole image by sliding windows.

This patchy operation is **convolution**. Different **convolutional kernels** are applied to as filters to extract different features. The neural network based on convolution is **convolutional neural network(CNN)**, which is widely applied on vision learning tasks. The general structure of a CNN is as follows. cs231-note.

- convolution layer: a set of kernels for a set of feature maps for vision features in certain receptive field. ⚠**learn weights in the kernel set**
- Non-linear layer: increase non-linearity and reduce parameter
- pooling layer: down sampling on each feature map for efficiency.
- fully connected layer ⚠**learn weights in neural network**
- output layer for downstream tasks

4.2 Training a CNN

4.2.1 Data Augmentation

Data augmentation simulates real-world variations (e.g., object positions, orientations, and lighting) and inflating the size of dataset. It avoids overfitting and ensures that a CNN generalizes well to unseen data by making the model invariant to transformations that are irrelevant to the task. AlexNet (2012) used random cropping and flipping to prevent overfitting on ImageNet.

4.2.2 Pooling[1]

Pooling layers downsample feature maps, reducing spatial dimensions while retaining essential information.

[1] Pooling can sometimes be replaced by strided convolutions, which simultaneously reduce spatial dimensions and learn features.

- **Max Pooling** selects the maximum value in each pooling window. This emphasizes prominent features, such as edges and textures, while discarding less important details.
- **Global Average Pooling (GAP)** replaces fully connected layers by averaging each feature map. For a feature map of size $H \times W$, GAP computes:

$$y_k = \frac{1}{H \cdot W} \sum_{i=1}^{H} \sum_{j=1}^{W} z_{i,j,k}.$$

  reducing the number of parameters, mitigating overfitting while maintaining spatial information in a compact form.

4.2.3 Convolution

A 1×1 convolution applies a linear transformation to each pixel across the channel dimension. This means it acts like a fully connected layer applied to each spatial location independently.

- **Dimensionality Reduction:** 1×1 convolutions reduce the number of feature maps while retaining spatial information. For example, in GoogLeNet's Inception modules, 1×1 convolutions are used to reduce the number of channels before computationally expensive 3×3 and 5×5 convolutions.
- **Feature Fusion:** They combine information across channels, effectively learning cross-channel interactions. This fusion helps the network capture more complex patterns.
- **Connection to Fully Connected Layers:** Mathematically, a 1×1 convolution at a spatial location (i, j) operates as:

$$y_{i,j,k} = \sum_{c=1}^{C_{\text{in}}} x_{i,j,c} w_{c,k} + b_k.$$

  This is equivalent to computing a weighted sum of the input channels, akin to what a fully connected layer does. However, 1×1 convolutions preserve spatial structure, unlike FC layers.

4.2.4 Residual Connection

Residual connections were introduced in ResNet (2015) to address the vanishing gradient problem in deep networks. Instead of directly learning a mapping $H(\mathbf{x})$, residual blocks learn a residual function:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x},$$

where $\mathcal{F}$ is the residual function (e.g., a stack of convolutions) and $\mathbf{x}$ is the input.

Why Useful?

- **Eases Optimization:** Residual connections allow gradients to flow directly to earlier layers, making very deep networks trainable.
- **Encourages Identity Mappings:** The network can easily learn identity mappings by setting $\mathcal{F}(\mathbf{x}) = 0$, avoiding degradation in deeper models.
- **Experimental Results:** ResNet-152 (152 layers) outperformed shallower networks while requiring fewer parameters, achieving state-of-the-art results on benchmarks like ImageNet.

4.2.5 Advanced Techniques

- **Dilated Convolutions:** Expand the receptive field without increasing the number of parameters or losing resolution. Useful in tasks like semantic segmentation (e.g., DeepLab).
- **Depthwise Separable Convolutions:** Decompose standard convolutions into depthwise (per-channel) and pointwise (1×1) convolutions. This drastically reduces computation and is widely used in lightweight architectures like MobileNet.

4.3 Image Classification

Challenges in image classification problem include **viewpoint variation**, **scale variation**, **deformation**, **occlusion**, **illumination conditions**, **background clutter**, and **intra-class variation**. A good image classification model should be invariant to the cross product of all these variations, while simultaneously retaining sensitivity to the inter-class variations.

CNN for image classification is one of the most advanced **data-driven** algorithm for image classification task. The downstream of the neural network is **classifier**. The most popular classifiers are **Support Vector Machine(SVM)** and **softmax classifier**. These approaches maps the original image to classes by score function, and quantify the correctness of classification using loss function, which can be converted to an optimization problem for neural network to learn.

- Multi-class Support Vector Machine(SVM)

$$\mathcal{L}_i = \sum_{j \neq y_i} \max(0, s_j - s_i + \Delta)$$

The loss will be calculated if the score difference between correct and incorrect class is greater than the margin Δ.

Figure 3: Multi-class SVM Loss

Regularization penalty will be applied to consider regularization loss and improve the generalization of a classifier.

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

- **Softmax Classifier** generalize binary logic regression and provide the possibility of classifying an image to certain class.

not finished!

4.4 Object Detection

For single object detection, the task can be view as a combination of classification and localization, where the total loss is collected from class loss and localization loss(L2 loss of the **bounding box** (x, y, w, h)).

First idea: **object detection as classification** problem. By doing multi-object detection where a sliding window is applied to the image and each crop of image is considered as a image classification problem, it can be fed into pre-trained ImageNet by **transfer learning**. **2 problems of the sliding window method: the window size is fixed, only 1 object can be detected once.** This largely increase the parameters for training(a CNN for each size of window and each object).

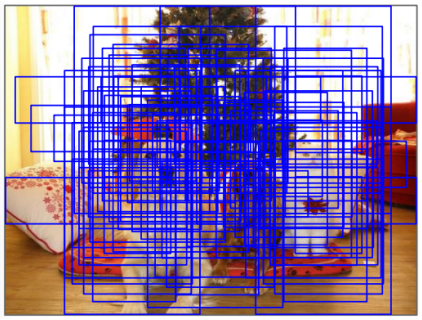


Figure 4: Object Detection by Sliding Window

Is there ways to apply adaptive box size based on the size of an object in the image, while detect multi-object class at the same time? One smart idea is **region proposal** or selective search, where "lobby" image regions that are likely to contain objects are proposed and run. **R-CNN** applies regional proposal on object detection by selecting **RoI**(region of interest) with selective search, using pre-trained networks to extract features, and applying a output model for classification and bounding box prediction.

- **IoU** measures the similarity between 2 boxes.

$$J(A, B) = \frac{A \cup B}{A \cap B}$$

- **bbox regression**

blog here

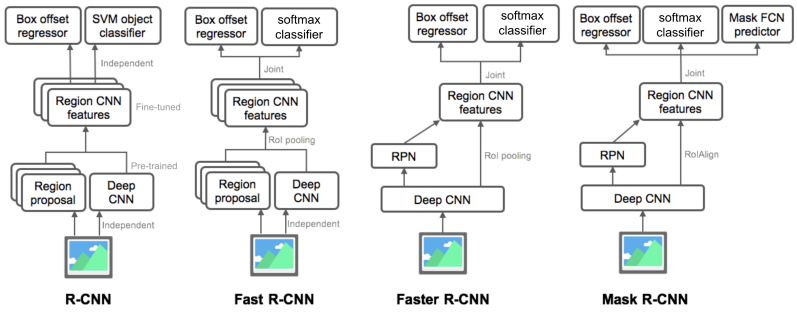


Figure 5: R-CNN Series

not finished!
not finished!
not finished!
not finished!
not finished!
not finished!

5 NLP with Deep Learning

5.1 Word Embeddings

In traditional NLP, each word type, presented as **tokens** in sentences, is encoded by one-hot vectors. It requires a large vocabulary while losing similarity information between similar words. In modern NLP, a good **word embedding** should be rather low-level(allowing for more data), and being able to learn similarities. **Distributional semantics**, one of the most influential and successful ideas in modern NLP assumes the similarity by context.

Intuitively, we can directly count the co-occurance of 2 words in context to represent their similarity. This approach, however, is high-level and sparse, which is not applicable. The **word2vec** provides a lower-level model for word vector learning. It can either predict context based on center word(**skip-gram**), or predict center word based on context(**CBOW**).

Take skip-gram as an example.

"I like drinking tea.", window size = 3.

sliding window	target word	context
[I like]	I	like
[I like drinking]	like	I, drinking
[like drinking tea]	drinking	like, tea
[drinking tea]	tea	drinking

$$L(\theta) = \Pi_{t=1}^{T} \Pi_{-m \le j \le m} P(w_{t+j}|w_t;\theta)$$

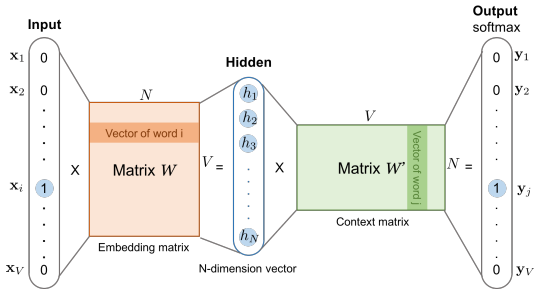


Figure 6: Skip-gram Model

The training objective of the model is to maximize the likelihood of correct center word. For the above model, the output layer is softmax of the probability of w_{t+j} given center word w_t.

$$P(w_c \mid w_t) = \frac{\exp(u_{w_c}^\top v_{w_t})}{\sum_{w \in V} \exp(u_w^\top v_{w_t})}$$

This softmax classifier, though efficient in images, is less efficient in NLP for a larger category of vocabulary. The **negative log-likelihood** is applied on this problem.

Note ⚠ Negative Loss Likelihood.

More about word embedding see this blog.

5.2 Seq2Seq Model

In modern NLP, a single end-to-end neural network with **seq2seq** architecture is extremely powerful in machine translation, summarization, dialogue, parsing, and code generation. The core idea of seq2seq is to convert a sequence(at **encoder**) to another sequence(at **decoder**) with **various length**.

Consider machine translation task by stacked RNN(an RNN as encoder and another RNN as decoder). The encoder and decoder is connected by certain middle state at the end of the sentence.

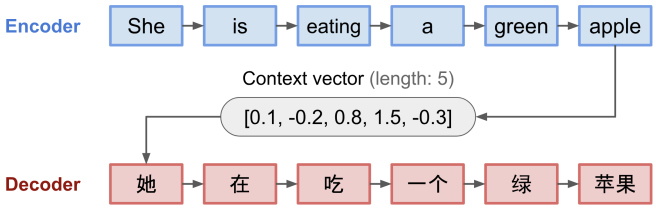


Figure 7: Seq2Seq Machine Translation Workflow

To find the most appropriate output, we should consider the meaning of the whole sentence instead of a single word.

$$P(y|x) = \Pi_{t=1}^{T} P(y_t|y_1, y_2, ..., x)$$

Though this approach compute a good solution, it is too computational expensive. A trade-off consideration is to use **beam search decoding**, where the beam size k keep track of the k most probable partial translations.

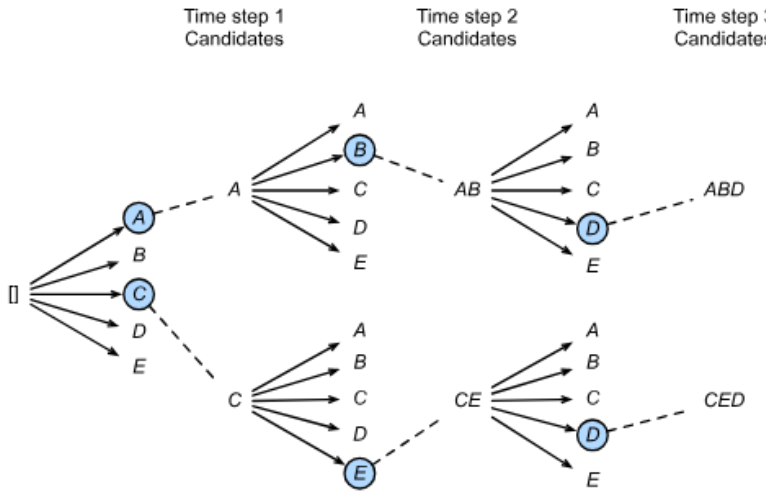


Figure 8: Beam Search Decoding

For machine translation tasks, **BLEU** compares the machine-written translation result and human-written translations for evaluation.

5.3 Transformer

There are several problems of the above seq2seq structure using RNN:

- lack of long-term dependency as RNN forget longer history
- a compressed middle state vector cannot represent all information well
- different input token have same contribution to the result

⚠ One of the attempts in solving long-term memory is the memory network.

5.3.1 Self-Attention and Encoder-Decoder Structure

A more amazing attempt, the attention mechanism directly connect each step of decoder to the encoder to focus on a particular part of the source sequence. The

self-attention building block matches the query(encoder) and values(decoder) to obtain a self-attention score and compute the output accordingly.

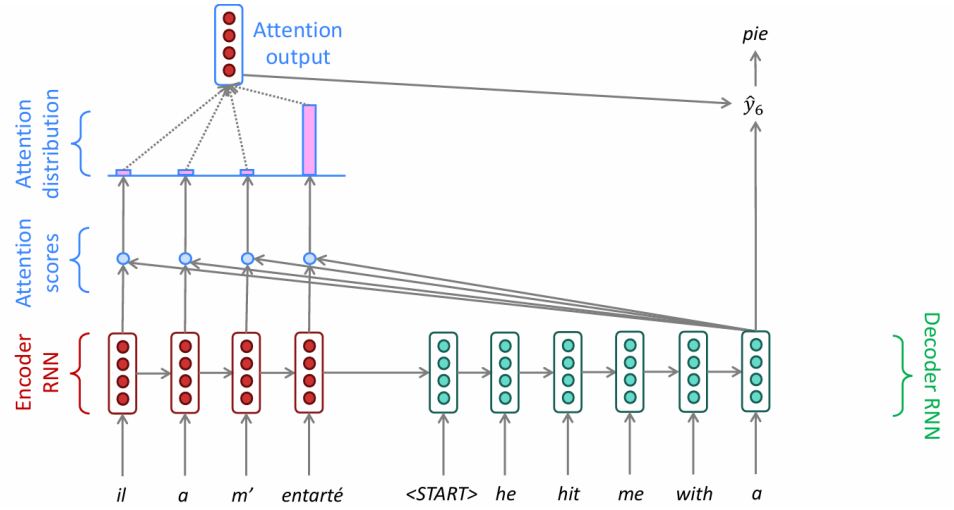


Figure 9: Transformer Structure for Machine Translation

The above structure can be computed efficiently by the following matrix computation. To enable parallelization, we mask out attention to future words by setting attention scores to $-\infty$. Each token can have different attention blocks, known as **multi-head** self-attention(similar to the idea of different channels in convolutional neural network, to obtain different features).

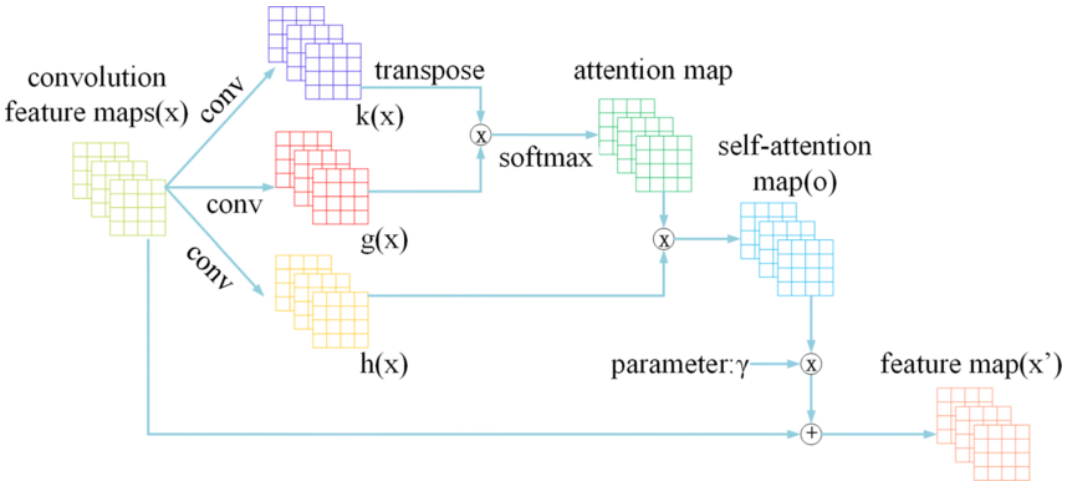
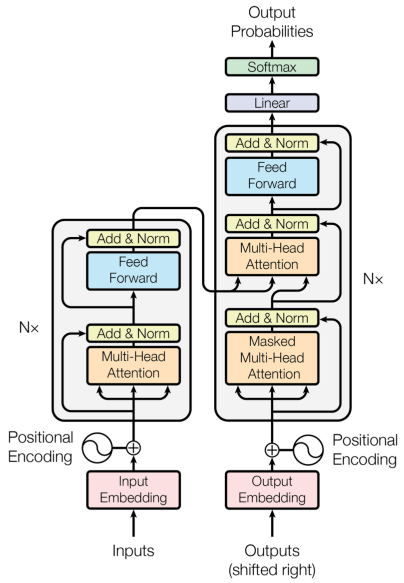


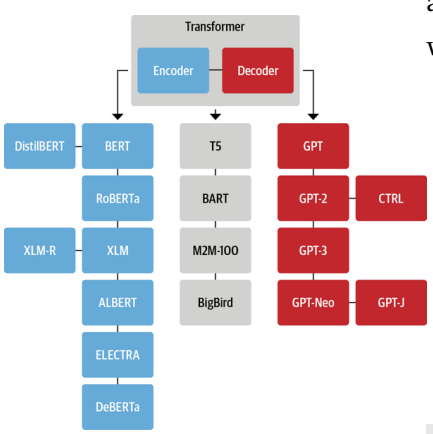
Figure 10: General Transformer Structure in terms of Matrix Computation



- The **encoder block** obtains k, q, v vectors from word embeddings and build multi-headed attention.
- The **decoder block** draws keys and values from encoder block as memory and queries from the decoder to compute **cross attention**.

Note ⚠ Things to note about transformer structure: quadratic compute in self-attention, possible position representation improvements, and pre-training process(will be mentioned later).

5.3.2 Transformer Variants



Pre-training is a powerful builds strong representation of language, parameter initialization and probability distribution over languages that we can sample from. We can pre-train on:

- **encoder only**: hiding contexts and get bi-directional context information, can not apply on text generation. **BERT** is a powerful pre-train model based on encoder.
- **decoder only**: for generative tasks
- **encoder decoder**

⚠ pre-training, post-training and continuous training process

6 Generative Model

The goal of generative modeling is to take input training samples from some distribution and learn a model that represents that distribution, i.e. learn $P_{model}(x)$ similar to $P_{data}(x)$. As one of the most powerful generative models, **latent variable** models tries to learn true explanatory factors(latent variables) from observed data only.

6.1 Variational Autoencoders(VAE)

Autoencoders raise an unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data. Then the decoder should train a model to use these features in latent space to reconstruct the original data.

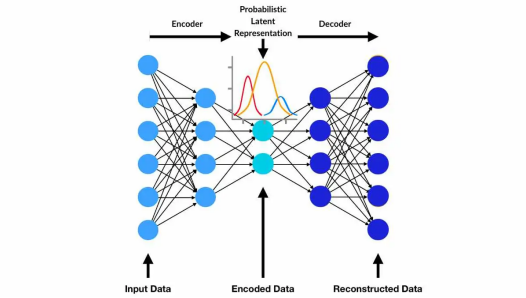


Figure 11: VAE

6.2 Generative Adversarial Network(GAN)[2]

Instead of explicitly model density, the idea of GAN is more straightforward: just sample to generate new instances. Main challenges lies in the evaluation of the generation result(**since we cannot directly compare it to the input**). The brilliant idea to solve this to combine the **generator** and **discriminator**, where, through the process, both generator and discriminator becomes stronger.

For the final goal, the discriminator tries to make $D(G(z))$ close to 0 while $D(x)$ close to 1; while the generator tries to make $D(G(z))$ close to 1. Thus the loss function to optimize is:

[2]Referred to this blog.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))],$$

where:

- $D(x)$ represents the discriminator's output for a real instance x, aiming to maximize $\log D(x)$.
- $D(G(z))$ represents the discriminator's output for a generated instance $G(z)$, aiming to minimize $\log(1-D(G(z)))$.
- $G(z)$ is the output of the generator when given input noise z sampled from a prior $p_z(z)$.
- $p_{\text{data}}(x)$ represents the real data distribution.

6.3 Flow-based Deep Generative Models[3]

A flow-based generative model is constructed by a sequence of invertible transformations.

7 Graph Neural Network(GNN)

A graph has several components including nodes, edges, and the whole system. A number of different tasks can be executed on graph data.

1. Node level prediction: to characterize the structure and position of a node
2. Edge level prediction: to show property for a pair of nodes
3. Graph level prediction: for entire graph

7.1 Graph Embeddings

In a traditional ML, the model is first trained on features of a graph in those 3 levels and then the model is applied to a new graph for prediction tasks. **Representation learning** avoids the need of doing feature engineering. The goal is

[3] Referred to this blog.

to map the graph into vectors, where the similarity of embeddings should reflect the similarity of graphs and nodes, known as **node embeddings**. To obtain such embeddings, there are 3 design choices:

1. Encoder: a look-up-table from nodes to vectors.
2. Decoder: similarity measurement of 2 embeddings. e.g. dot product
3. Similarity: a target similar function measuring similarity of 2 nodes, the objective to approximate by encoder and decoder. e.g. distance in the node.

Many methods stem from this simple setting, including **DeepWalk** and **node2vec**. We can also generate word embeddings end to end using graph neural network.

7.1.1 RandomWalk Embedding

7.1.2 Node2vec Embedding

7.1.3 Graph Convolutional Network

To learn embeddings from a graph neural network, the biggest challenge is to maintain the **permutation invariant** of a graph[4], i.e. for any ordering of nodes in adjacency matrix, the output should be the same(invariant), or remains the same ordering property(equivariant).

$f(PAP^T, PX) = f(A, X)$ **permutation invariant** or $Pf(A, X)$ **permutation equivariant**

One smart idea to apply the invariant convolutional kernel to a target node in the graph to "convolve" each layer of its neighborhood. Then for each target node, there is a computation graph based on local network neighborhoods. We can aggregate information across layers using **neighborhood aggregation**[5].

$$h_u^{(0)} = x_u \quad \textbf{the 0th layer node embedding is its input feature}$$
$$h_u^{(k+1)} = UPDATE^{(k)}(h_u^{(k)}, AGGREGATE^{(k)}(h_v^{(k),\forall v \in N(u)})) \quad \textbf{update embeddings}$$
$$z_u = h_u^{(K)} \quad \textbf{output embedding}$$

where UPDATE and AGGREGATE are neural network to be trained. A simple aggregation function is to take the average of all its neighbors.

$$h_v^{(k+1)} = \sigma\left(W_k \frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(k)} + B_k h_v^{(k)}\right)$$

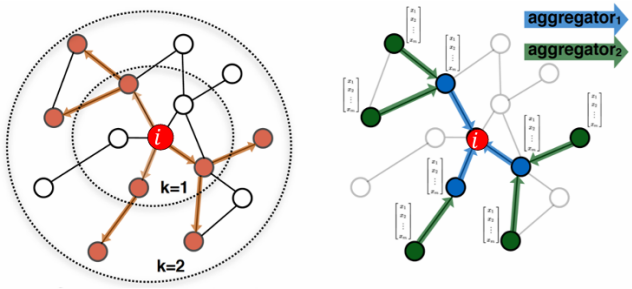


Figure 12: Computation Graph from Neighborhood

[4]Many deep learning network including seq2seq, traditional CNN, does not have such property.

[5]After k iteration, a node embedding should contain information about its k-hop neighborhood, which encodes the graph structure information and features information

A GNN can be trained using SGD by updating the weight matrices W_k, B_k, for aggregation and transformation, respectively. The update function can re-write into a matrix form of **self-transformation** and **neighborhood aggragation**.[6]

$$D_{v,v}^{-1} = \frac{1}{|N(v)|} \text{ a diagonal matrix for aggregation}$$
$$H^{(k+1)} = \sigma\left(W_k^T D^{-1} A H^{(k)} + B_k^T H^{(k)}\right)$$

There are several settings to train a GNN:

1. supervised setting: minimize the loss
2. unsupervised setting: supervised by graph structure to capture similarity.

$$\mathcal{L} = \sum_{z_u, z_v} CE(y_{u,v}, DEC(z_u, z_v))$$

These trained parameters can embed nodes of same type for generalization.

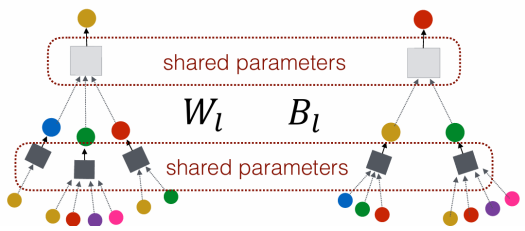


Figure 13: Graph Embedding Generalization

7.2 A General GNN Framework

1. message
2. aggregation function(GCN, GraphSAGE, GAT)
3. layer connectivity
4. graph augmentation
5. learning objective

[6]The $D^{-1}AH^{(k)}$ guarantees the invariant property

A GNN layer contains aggregation and message, which is connected to the whole GNN by stacking layers sequentially. For different tasks it requires a proper way to augment the graph feature and structure for computation, instead of directly input the raw graph data. Learning objective decides how we train the model to be optimized.

7.2.1 A single GNN layer

A GNN layer compresses a set of vectors into a single vector. As shown in the previous GNN for graph embedding, it takes input $h_v^{(k)}$ and $h_{u\in N(v)}^{(k)}$ and output $h_v^{(k+1)}$ through **message** function and **aggregation** function.

$$h_v^{(k)} = concat(Agg(\{m_u^{(k)} : u \in N(v)\}), m_v^{(k)})$$

⚠ it is possible to send different messages to different neighbours, e.g. GAT

Based on this architecture, we can develop several instance of GNN layers.

1. **Graph Convolutional Network(GCN)**

$$h_v^{(k)} = \sigma\left(\sum_{u\in N(v)} W^{(k)} \frac{h_u^{(k-1)}}{|N(v)|}\right)$$

2. **GraphSAGE**

$$h_v^{(k)} = \sigma\left(W^{(k)} \cdot concat(h_v^{(k-1)}, Agg(h_u^{k-1}))\right)$$

   For aggregation function, it can be **mean**, **pooling**, or **LSTM**. Then apply L_2 normalization at every layer.

3. **Graph Attention Network(GAT)**

$$h_v^{(k)} = \sigma(\sum_{u\in N(v)} \alpha_{u,v} W^{(k)} h_u^{(k-1)})$$

   A certain attention mechanism a is applied to compute attention coefficient $e_{u,v}$ and normalize into the final attention weight $\alpha_{u,v}$. To stabilize the learning process, multi-head attention is applied to learn different attention scores.

7.2.2 Stacking GNN Layers

To connect a graph neural network, the standard way is to stack GNN layers sequentially, starting from initial $h_v^{(0)}$ to h_v^L after L GNN layers. The problem of this method is the **over-smoothing problem**, where all the node embeddings converges to 1 due to the highly overlapping of receptive fields. We should set number of GNN layers to be a bit more than the receptive field we like.

1. Make a shallow GNN be more expressive:
    - increase the expressive power within each GNN layer: turn the linear aggregation into a deep neural network.
    - add layers that do not pass messages: add MLP layers as pre-process and post-process layers.
2. Add shortcut for deep GNN: **skip connection**.[7]

7.2.3 Graph Manipulation

For graph input without features, we can apply **feature augmentation**. For example, assigning unique one-hot id to each node, or calculate graph features(such as cycle feature). At structure level, for a sparse graph, we can add virtual nodes(connect to all the nodes in the graph) or virtual edges(connect 2-hot neighbors as well), while for a dense graph, we can sample neighbors when doing message passing.

7.3 GNN Training Pipeline

7.3.1 Prediction

The Node embeddings after a GNN will be send to the prediction head for different prediction tasks.

1. node-level: for a k-way prediction in classification or regression task

$$\hat{y}_v = W^{(H)} h_v^{(K)}$$

[7] an intuition come from residual network, making f(x) to be f(x)+x, providing 2 path

where $W^{(H)} \in \mathbb{R}^{k \times d}$, mapping node embeddings to prediction space for loss computation.

2. edge-level:

$$\hat{y}_{u,v} = Head(h_v^L, h_u^L)$$

where the Head can be concatenation of linear layers or dot product.

3. graph-level: similar to aggregation, calculate the global pooling.

7.3.2 Loss Function and Evaluation

7.3.3 Split Training

Splitting graph is special since all nodes are not independent. Solutions include **transductive setting**, where the input graph is visible in all the dataset splits and only split the node labels for training, validation, and testing; **inductive setting**, breaking the edges between splits to get multiple graphs.

7.4 Scale Up GNN

8 Foundation Models: LLM