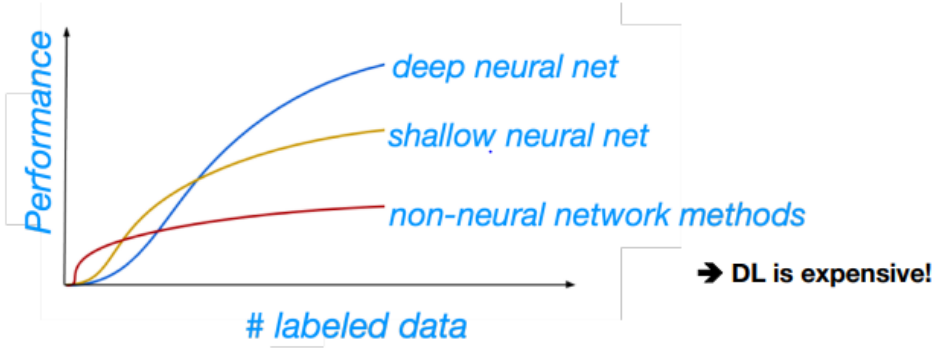
LVC 3 - Transfer Learning and GNNs

In the previous two lectures, we have studied the working of vanilla Neural Networks as well as Convolutional Neural Networks for image classification. In this lecture, we will study more modern state-of-art techniques in neural networks - **Transfer learning and Neural Networks for Graphs**. Let's go through these topics one by one.

Transfer Learning

Let's answer the most obvious question first - **why do we need transfer learning?**

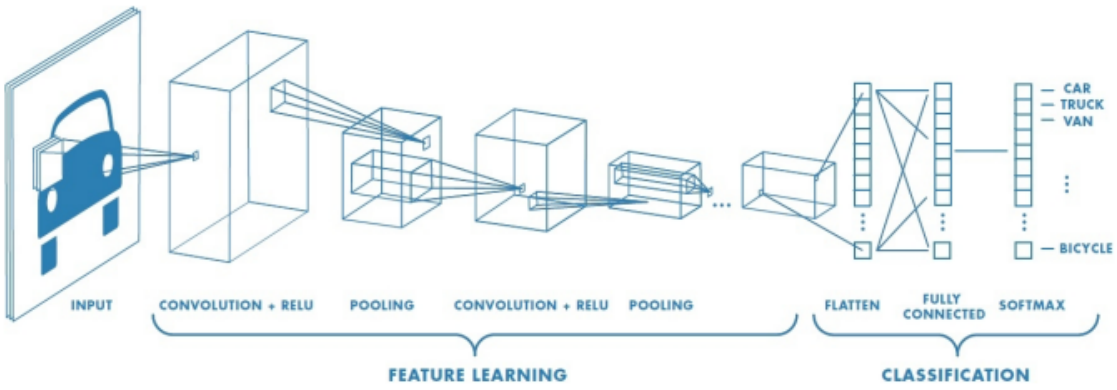
This is mainly due to the fact that **breakthroughs in deep learning require huge data** (usually in millions) and deep learning is not very effective without the same.
The below graph shows that deep neural networks are able to perform better than shallow neural networks and non-neural networks methods only when the amount of labeled data is high. For small amounts of data, non-neural network methods outperform deep learning.



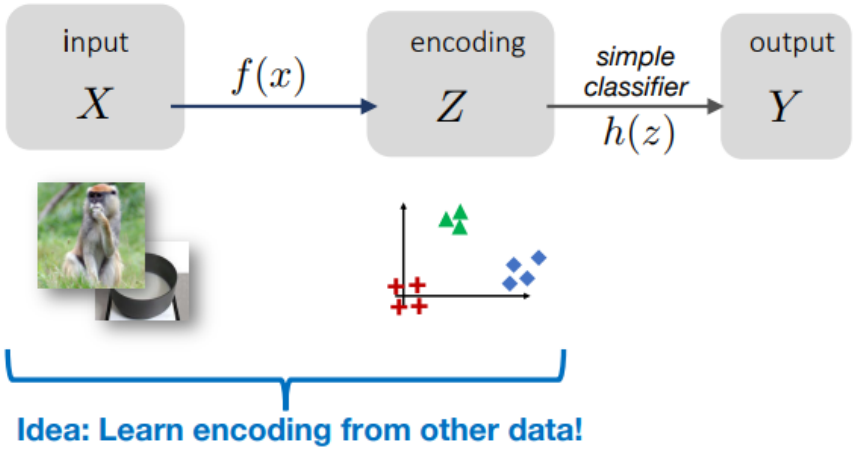
But generating huge amounts of labeled data is expensive in terms of monetary resources, manual labor, and time. As a matter of fact, for many tasks, we don't have such a huge amount of labeled data. For example, in medical imaging, we usually don't have millions of labeled images of medical conditions because we would need experienced healthcare professionals to do that which can be very expensive.

So what if labels are scarce? How can we ensure that we can build a model with high accuracy? This is where the method of **transfer learning** comes into play. So the question is, **what is transfer learning?**

Each convolutional neural network consists of three major parts - **the input**, **the feature learning** (this is also called **representation** or **encoding**), and **the classification**.



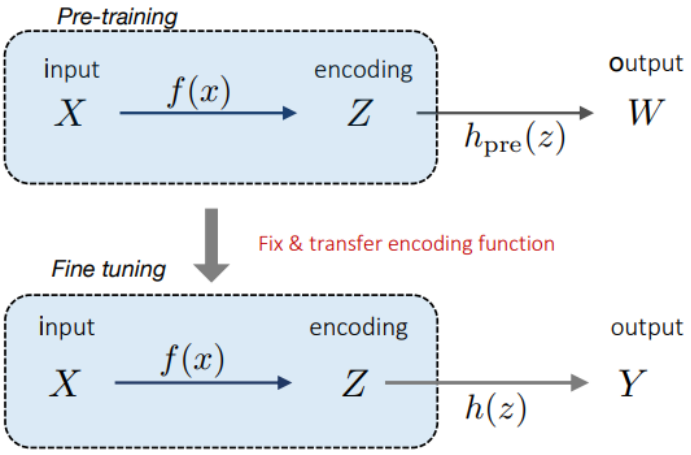
The idea of transfer learning is to **use pre-trained features** from some other data that has a lot of images, for example, ImageNet data. These pre-trained features can give a good representation of the data we have. Once we have that representation, a simple, data-efficient classifier can do well to solve the problem. The below picture depicts the concept of transfer learning.



There are mainly two stages in transfer learning - **pre-training and fine-tuning.** In pre-training, we learn the encodings from some other labeled data (say m data points) that might be related to a different task, called the **pretext task.** In fine-tuning, we use the learned encoding to have a good representation of the current task at hand, called the **target task**, and train a classifier with a much fewer number of data points (say n, such that n<<m).

Remark: The same pre-trained encoding may work for multiple target tasks.

The above diagram shows two stages of transfer learning.

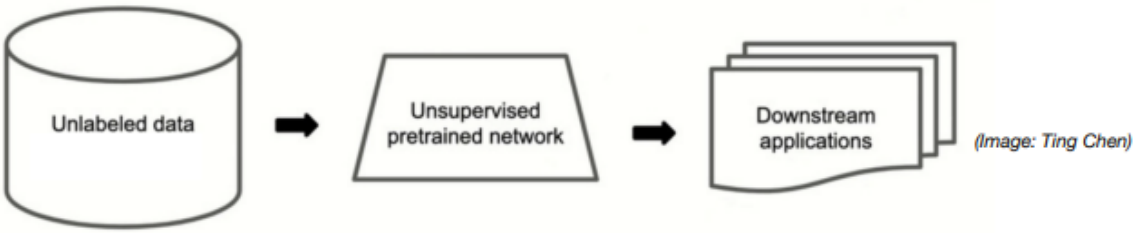


Now, there are mainly **two types of transfer learning** depending on the pretext task:

1. **Supervised Learning:** Here**,** the pretext task has properly labeled data, for example, ImageNet, which we can use for pretraining.
2. **Self-Supervised Learning**: Here, the pretext task does not have predefined labeled data but we generate labels for the data ourselves.

While the supervised method is a more straightforward and simple idea, the self-supervised learning method has become popular and has been known to outperform supervised pre-trained learning. The main idea behind **self-supervised learning** is to learn useful representations of the data from an unlabelled pool of data using self-supervision i.e. **create labels automatically.** If we can create labels, then we would have much more data to train the model simply because there is much more unlabeled data out there in comparison to labeled data, so it is not at all expensive to collate unlabeled data.

The below diagram shows the pipeline for the self-supervised learning method. The downstream task could be as simple as image classification or complex tasks like object detection, visual data mining, etc.



But how do we create labels automatically for the pretext task?
In self-supervised learning, the pretext task is different because we don't have labels. Let's look at an example of a pretext task for images.

Where does the patch belong?

(Doersch, Gupta, Efros 2015)

Suppose, we have an unlabeled image of a cat and we chop off the image into 9 patches. We take the center patch and randomly take one of the remaining 8 patches. Now, we ask the neural network where this new patch (shown with red border line on left) would go in the jigsaw puzzle on the right? Now, to actually perform this task, the network should be able to recognize that this is an animal and the other patch looks like the right ear of the animal and it should go on the top left i.e. it should recognize the pattern.

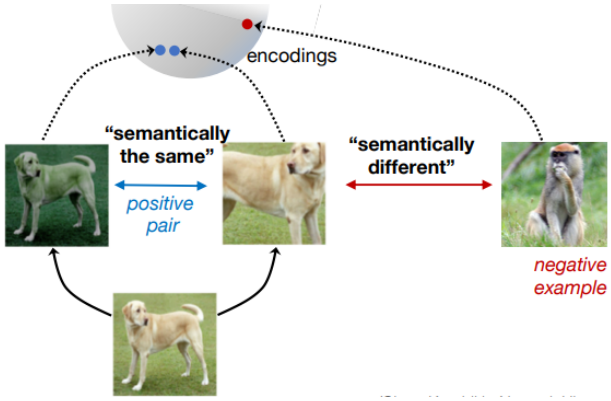
Basically, we are training the network to extract and recognize patterns without labels. This way we can collate a lot of unlabeled images and generate a huge amount of training data.

There can be other kinds of pretext tasks. We are going to discuss one of the most popular pretext tasks, called **contrastive learning.**

Contrastive Learning

The goal of contrastive learning is to ensure that **encoding of similar (positive) pairs should be close** to each other and the **encoding of dissimilar (negative) pairs should be distant**.

In the below diagram, the positive pair are the images of a dog and the negative example is the image of a monkey.



Let's go into a little more detail about how contrastive learning actually works.

To train the model effectively, a **loss function** is required to enforce the similarity between positive pairs and/or dissimilarity between negative pairs. The below loss function. called **Noise Contrastive Estimation Loss,** is used for this purpose:

$$min_f E_{x,\, x^+, \{x_i^-\}_{i=1}^N} \left[-\, log \frac{e^{f(x)^T f(x^+)}}{e^{f(x)^T f(x^+)} + \sum_{i=1}^{N} e^{f(x)^T f(x^-)}} \right]$$

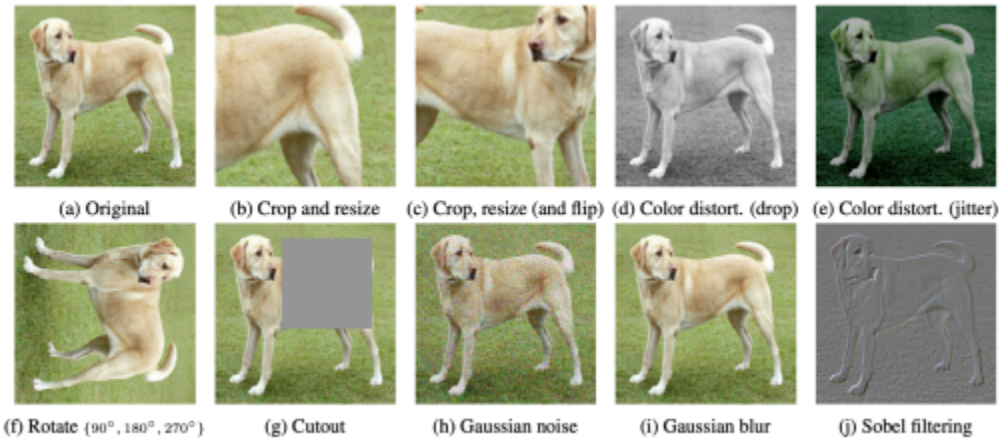
Where x is the original image, **called anchor,** x^+ is the positive sample, and x^- is the negative sample.

The equation finds the embedding, function f, that minimizes the loss function. To minimize this loss function, the model needs to find the embedding that makes the inner product of the original image with the positive pair large i.e. $e^{f(x)^T f(x^+)}$ is large and the inner product of the original image with the negative pair small i.e. $e^{f(x)^T f(x^-)}$ is small.

The inner product would be small if the angle between the two samples is small i.e. they are close to each other. Similarly, the inner product would be large if the angle between the two samples is large i.e. they are distant from each other. This is exactly the main goal of contrastive learning.

Now, the only thing we need to figure out is - since we don't have labeled data, **how do we get positive and negative samples?**

To get **positive samples**, we generate random combinations of **data augmentation** for the original image. For example, if the original image has a dog, the positive samples can be created as shown in the below image:



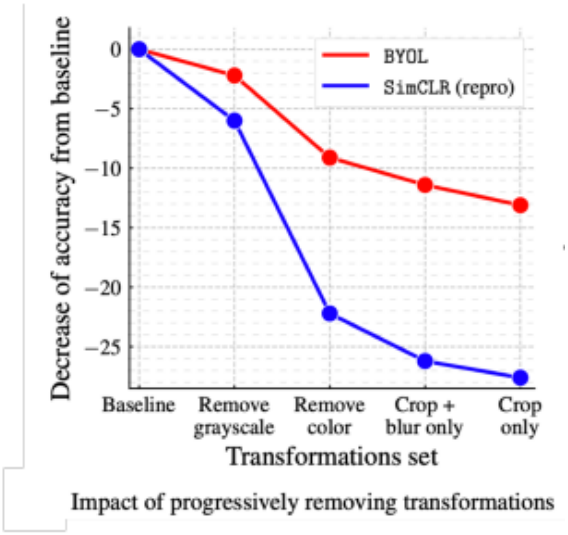
We can apply any of the two transformations, mentioned in the above image, and most likely both the images would still contain a dog and become the positive pair. This also helps the model to be **invariant or robust to slight variations** in the data.

To get **negative samples**, we simply **draw random samples** from the data, and most likely the randomly drawn image would be different from a dog.

Remark: We cannot ensure that the randomly drawn negative samples do not contain an image of a dog but this is as good as we can do without any labels. In practice, this is not a major concern and it works very well.

There are some techniques that can be used to **improve the performance of contrastive learning**:

- **Heavy use of data augmentation** i.e. generate a lot of images using different kinds of transformations. The below plot shows the decrease of accuracy in the model for dropping each type of augmentation and we can see that the accuracy dropped significantly when only one (crop only) or two (crop + blur) types of augmentations were used.



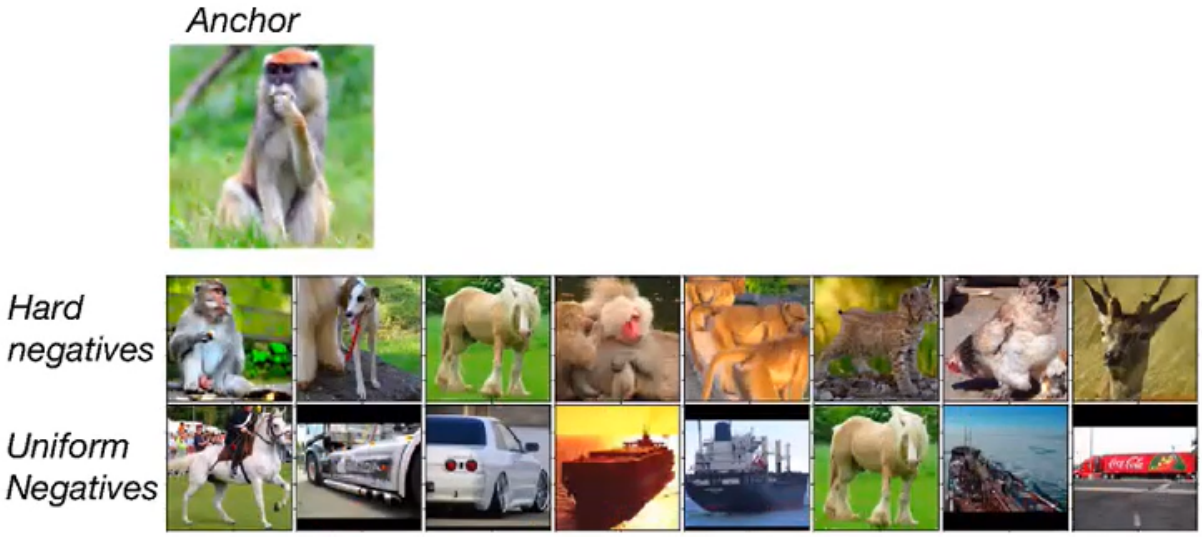
(figure: Grill et al 2020)

- **Minimize false negative examples** i.e. minimize the chance of choosing a negative sample when it should be a positive sample. The below image shows that the image of a dog has been selected as a negative sample when it should be a positive sample.

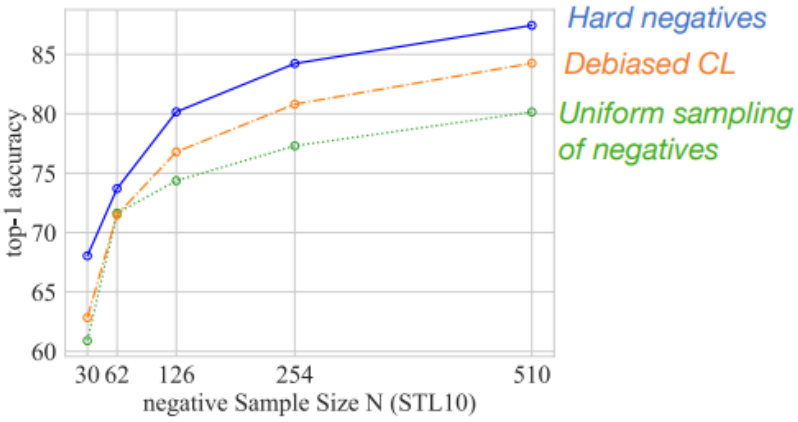
We can minimize false negatives using a technique called **debiasing.** The main idea of debiasing is to develop a correction for the sampling bias that yields a new, modified loss, called **debiased contrastive loss.**

Note: We will not dive deep into debiasing. Interested readers can read more about it from the original paper - *Debiased Contrastive Learning by Ching-Yao Chuang, Joshua Robinson, Lin Yen-Chen, Antonio Torralba, Stefanie Jegelka*

- **Focus on hard negatives** i.e. pick negative examples that are not too easy to differentiate. The below image shows the difference between hard negatives and uninformative negatives. The anchor image is a monkey and the hard negatives are more similar to the anchor in terms of shapes, textures, etc. as compared to uniform negatives. So, hard negatives allow the model to extract or learn more information from the data.



Finally, let's observe the below plot which shows how each of these techniques helps to improve the performance of the model.



The plot clearly shows that **debiased CL outperformed standard uniform sampling of negatives** and using **hard negatives further improves the performance** of the model.

So far, we have learned about transfer learning and its different techniques. Now, let's move to the second topic of this lecture - Graph Neural Networks (GNNs).

Graph Neural Networks (GNNs)

A Graph is a data structure containing nodes and edges. The relationship/connection between the various nodes is defined by the edges.

GNNs are a set of deep learning methods that work on graphs i.e. the input is a graph. These networks have recently been applied in multiple areas and have become very popular. Some of the popular applications of GNNs include:

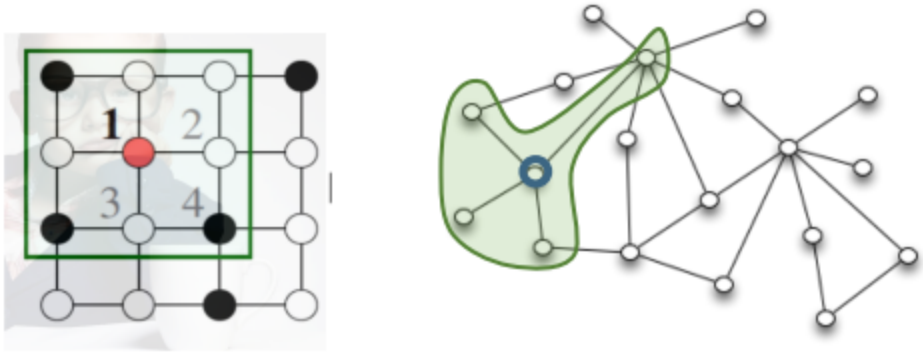
- Molecule Property Prediction
- Social Network
- Recommender System
- Polypharmacy Side Effects Prediction
- Learning to Simulate Physics
- Predicting traffic times

We will revisit one of these applications later and discuss it in detail.
For now, let's try to understand: **What are GNNs and how do they work?**

In the previous lecture, we observed local patches (small parts) of an image are really important as they contain information like patterns, textures, shapes, etc. In CNNs, we apply different filters and each filter encodes a local patch, also called the **local neighborhood**.

What about graphs? **Is local information important in graphs?**

Yes, the local information is important in graphs as well. Let's consider the social network example, the neighborhood of a node (person) might give information about who their friends are and who are friends of their friends. In social networks, this is an important piece of information to have.



Encode Local Neighborhoods (patches)

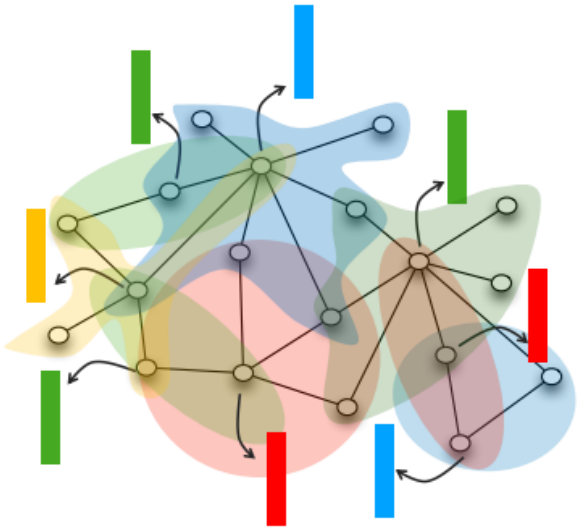
So, we will follow the same idea of encoding local patches in GNNs with a slight variation. In CNNs, we basically know the shape of local patches i.e. square and each non-border pixel will have the same number of neighbors.

But in graphs, we don't know the shape of local patches. For example, a person might have 2 friends and another person might have 100 friends on social networks. So, **how do we encode local patches in GNNs?**

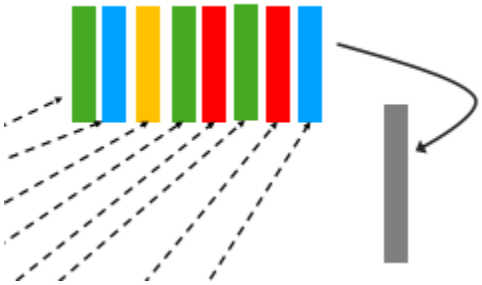
Big picture of Graph Neural Networks (GNNs)

A GNN works in two major steps - **Node Embeddings and Readout**.

- **Node Embeddings:** In the first step, each local patch is encoded. Since each local patch is centered on a node, we are basically encoding the neighborhood of each node. So, for each node, we get a vector representation of its neighborhood. In the below image, different colors represent different local patches, and each strip represents the encoded vector of that patch.



- **Readout:** If we want to make a prediction from a single node, we can train a classifier on the encoded vector for that node. But if we want to make predictions on the whole graph, then we need to **aggregate all the node embeddings**. The step of taking all the vectors and aggregating them **into a single vector** is called Readout. The single vector carries all the important information on the graph. In the below image, all the node embeddings have been aggregated to a single vector denoted by the gray colored bar.



Now, the question is **how do we get node embeddings?**

We use a technique called **message passing** to get these node embeddings. It is called message passing because each node gets information from each of its immediate neighbor nodes, which in turn, get information from their immediate neighbor nodes.

There are two steps at **each round** in message-passing - **Aggregate and Update**.

Let$v$ be the center node, $N(v)$ be the neighborhood, and h_v be the embedding of node v. At each round k, we first aggregate the embeddings, denoted by $m_{N(v)}^{(k)}$, of neighbor nodes from the $(k-1)^{th}$ round, and then we update the embedding of node v by combining $m_{N(v)}^{(k)}$ with h_v. The algorithm works in the following way:

At each round k:

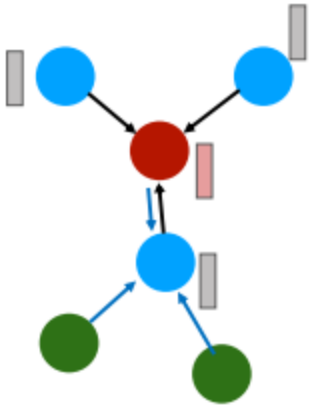
- **Aggregate** over neighbors:

$$m_{N(v)}^{(k)} = AGGREGATE^{(k)}(\{h_u^{(k-1)} : u \in N(v)\})$$

- **Update**: Combine with the current node

$$h_v^{(k)} = COMBINE^{(k)}(h_v^{(k-1)}, m_{N(v)}^{(k)})$$

This message passing is done by each node, so we have new embeddings for each node at each round. We can have several rounds in GNNs. In the first round, each node would get information from its immediate neighbors, in the second round, nodes might get information from nodes that are one step away from them, and so on. The below image shows this.



In the first round, the red node would get information from only blue nodes as they are the immediate neighbors of the red node and the bottom blue node would get information from green nodes. In the second round, since the bottom blue node contains information of green nodes, the red node would indirectly get information from the green nodes as well.

Remark: Each round is a layer in a GNN and the number of rounds is the number of layers in GNNs.

We have talked about aggregation and updating the embedding, but **how do we perform these operations?**

- Aggregation: There are multiple operations that can be used to perform aggregation of neighbor embeddings. Some of the popular operations include:
    - **Averaging:** It takes the average of all the embedding from the neighborhood.
    - **Coordinate-wise min/max:** Since each embedding is a vector of the same dimension, we can take coordinate-wise minimum or maximum to get a single aggregated vector.
    - **Using small neural networks:** Here, Multi-Layer Perceptrons (MLP), i.e. shallow feed-forward neural networks, are used to **learn** aggregation. There are weights that can be learned during training and extract the most useful features or information from neighbor embeddings. Mathematically, it is written as:

$$m_{N(v)} = MLP_{\theta}\left(\sum_{u \in N(v)} MLP_{\phi}(h_u)\right)$$

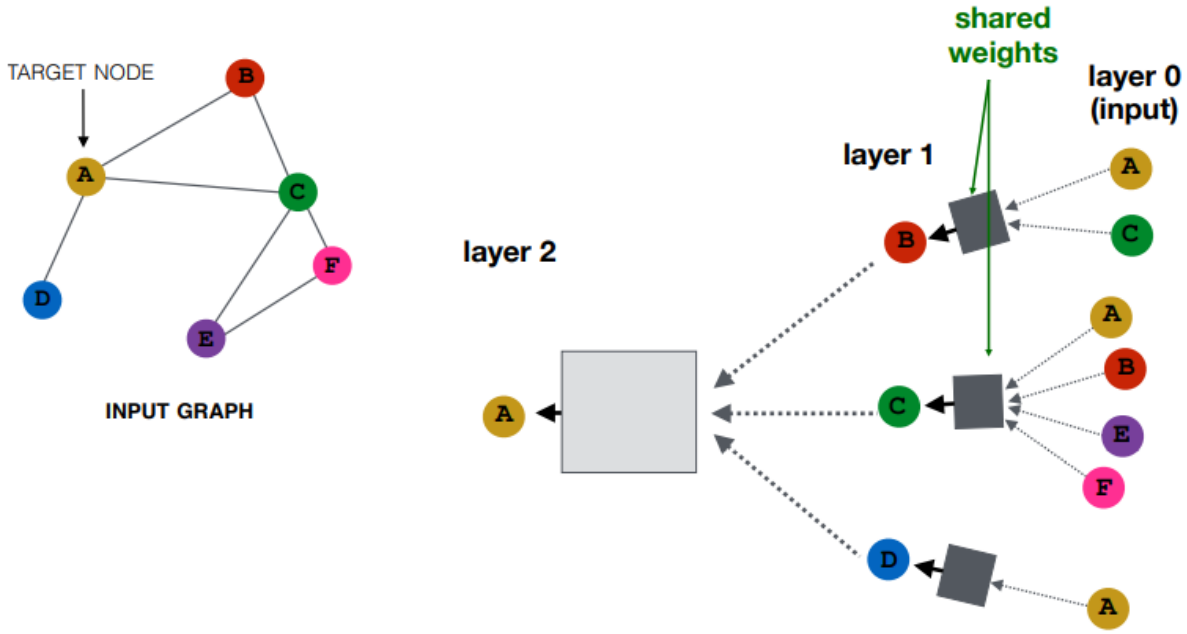
    Here, MLP_{ϕ} is used to learn features of each embedding, and MLP_{θ} is used to again learn features from the output of adding features of individual embeddings.

- Update: Mathematically, the update step in GNN message passing is defined as:

$$h_v^{(k)} = \sigma\left(W_{self} h_u^{(k-1)} + W_{neigh} m_{N(v)}^{(k)} + b\right)$$

Where W_{self} is the weight of the assigned to the node and W_{neigh} is the weight assigned to the aggregation embedding, b is the bias term, and σ denotes an elementwise non-linearity (e.g., a tanh or ReLU). These weights are learned while training the model. They help the model to decide how much the node should retain its current information and how much should learn from the neighbors.

We have seen how we can use multiple MLP for each node embedding and update the embedding using nonlinearities. This conveys the idea of viewing a graph neural network as a big neural network of smaller neural networks, also called a **structured neural network**. Let's try to visualize node embeddings as a neural network.



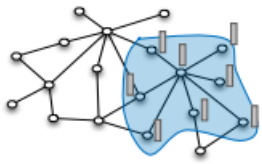
Node embeddings as neural network

The figure above gives an **overview of how a single node aggregates messages from its local neighborhood**. The model aggregates messages from A's local graph neighbors (i.e., B, C, and D), and in turn, the messages coming from these neighbors are based on information aggregated from their respective neighborhoods, and so on. This visualization shows a two-layer version of a message-passing model. Each gray box represents a neural network (aggregations functions) that we learn while training the model. Notice that the **computation graph of the GNN forms a tree structure** by unfolding the neighborhood around the target node.

Now that we have a conceptual understanding of graph neural networks and how they work. Let's understand the steps involved in training a GNN.

Training a GNN

- **What is a data point?** The first step is to decide the nature of the data point i.e. the input. The input may consist of a node and all its neighbor nodes or it might be a full graph, eg: molecular structure, and the dataset can be many different graphs.



Node + its neighborhood (computation tree)

Full graph

- **What to specify?** The second step is to specify the architecture of the model - the aggregate function, the combine function for the update operation, loss function on prediction as per the nature of the problem - classification or regression, etc.
- **Train with SGD -** The final step is to train the graph neural network just like we train other types of neural networks. We can use Stochastic Gradient Descent to train the model.

In practice, there are multiple libraries that can be used to build and train graph neural networks. For example, PyTorch, Deep Graph Library, etc.

Finally, let's go through an example to get a better understanding of the applications of GNNs in the real world.

Example: Polypharmacy Side Effects Prediction

Original Paper - Modeling polypharmacy side effects with graph convolutional networks By Marinka Zitnik, Monica Agrawal, Jure Leskovec

Problem Overview:

The problem is to predict the risk of adverse side effects if two drugs are taken together. It is a big issue that can be severe in many cases. About 15% of the US population is affected by these side effects and this results in an annual cost of more than $177 billion.

This problem cannot be solved by brute force clinical testing because there can be a lot of combinations and it is a rare occurrence that might not be observed in clinical testing. So, we can use the machine learning model to predict the risk, and then perform clinical testing on those drugs.

The input would be feature vectors of 2 drugs and output would be the probabilities for different types of side effects. So, it is a multi-class classification problem for predicting probabilities for different types of side effects.

Data Collection:

The first question is what information/data can be used to solve this problem?

We can collect data on drug-drug interactions (types of side effects) and side effects of individual drugs but the problem is there is very little data available on drug-drug interactions due to their rare occurrence, as mentioned above. So, what else can we use?

We can use the information on the proteins that each drug works upon. Further, we can include the interactions between different proteins as a drug might not directly affect a protein but it may affect it indirectly due to the interaction of that protein with the target protein of the drug.

The data for this study has been collected from multiple sources to collect three pieces of information - drug-drug, drug-protein, and protein-protein interactions.

Exploratory Data Analysis:

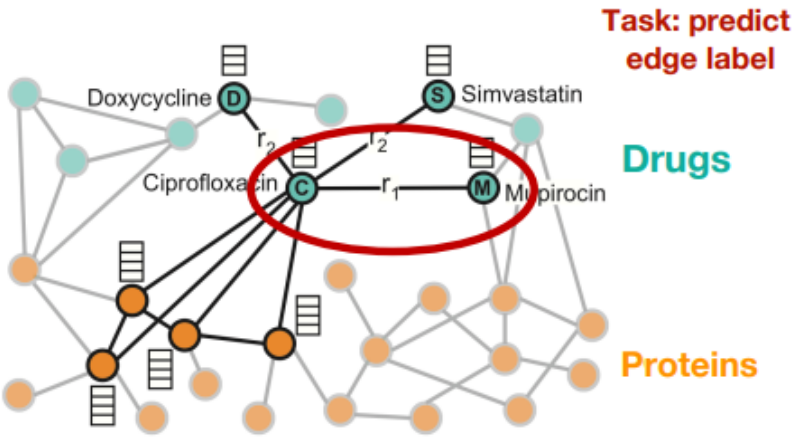
The following points were observed in EDA:

- Drugs that are prescribed together tend to have more target proteins in common than random drug pairs. In general, more than 68% of drugs have no target protein in common.
- There is a wide range of how frequently certain side effects occur in combinations (>53% of side effects in <3% of combinations). So, we have very little data on rarer side effects.
- Side effects do not occur independently of each other i.e. there might be some correlation between different side effects. For example, hypertension co-occurs with anxiety, but it is negatively correlated with fever. If the model predicts fever, then it is less likely for that person to have the side effect of hypertension. So, we can take this into account and then predict the side effects jointly using these correlations.
- So, the input network looks like the below image:

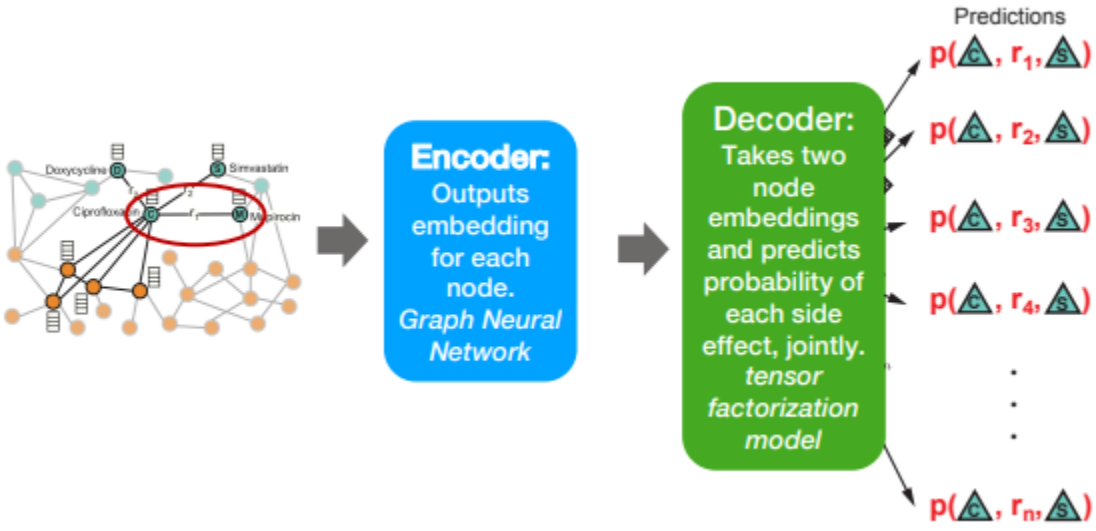
Solution Overview:

- The input graph of the model is a fairly large network containing:
    - 964 types of polypharmacy side effects

- 645 drug nodes, 19,085 protein nodes
- 3 kind of edges: 715,612 protein-protein, 4,651,131 drug-drug, 18,596 drug-protein



- There is label imbalance in the data i.e. the data has many pairs that do not result in an adverse side effect. There are more “no” than “yes”. To tackle this, subsampling is done for the “no” samples.
- The goal is to find the encoding of drug nodes. The encoding of each node would occur in almost the same way that we discussed in the previous section with one difference. There are 3 different kinds of edges in this graph. So, aggregation is done on each type of edge separately and then taken their average to get the node embedding.
- SGD algorithm with adam optimizer has been used to train the network.
- The below image shows the pipeline of the model. The first part is an encoder that takes the graph as input and outputs the embedding for each node using a GNN. Then the decoder applies matrix factorization to predict the probability of each side effect, jointly.



Note: Matrix Factorization is part of the next course - Recommendation Systems.

Model Evaluation:

How would you evaluate the performance of such a model? The first step is to simply observe the prediction performance i.e. how many does the model get correct. The second step is to check where the model is making the most mistakes.

The model discussed here performs especially well for side effects with molecular underpinnings i.e. protein interactions.

The model was actually able to predict new side effects that were not part of the training data but exist in the literature. The below table shows some of these new predicted side effects:

Polypharmacy effect	Drug i	Drug j
Sarcoma	Pyrimethamine	Aliskiren
Breast disorder	Tolcapone	Pyrimethamine
Renal tubular acidosis	Omeprazole	Amoxicillin
Muscle inflammation	Atorvastatin	Amlodipine
Breast inflammation	Aliskiren	Tioconazole

Links for Additional Reading/Practice

- Original Paper on Debiased Contrastive Learning
- Original Paper on Modeling polypharmacy side effects with graph convolutional networks
- GitHub Repository for papers related to GNNs - https://github.com/thunlp/GNNPapers#basic-models
- Implementation of GNNs
    - Library for GNNs - https://www.dgl.ai/
    - Implementation of GNNs using PyTorch - https://github.com/pyg-team/pytorch_geometric