# Modern Deep Learning Models

Tong Shu

DL-GPU Workshop 2024
Day 7

05/21/2024

# Graph Neural Networks

# Graph Learning

- Node classification
  - A task that involves predicting the category (class) of a node in a graph.
- Link prediction
  - A task that involves predicting missing links between pairs of nodes in a graph.
- Graph classification
  - A task that involves categorizing different graphs into predefined categories.
- Graph generation
  - A task that involves generating new graphs based on a set of desired properties.

# Graph Measures

- Degree centrality is simply defined as the degree of the node.
  - A high degree centrality significantly influences the network.
- Closeness centrality measures the average length of the shortest path between the target node and all other nodes in the graph.
  - A node with high closeness centrality can quickly reach all other vertices in the network.
- Betweenness centrality measures the number of times a node lies on the shortest path between pairs of other nodes in the graph.
  - A node with high betweenness centrality acts as a bottleneck or bridge between different parts of the graph.
- Density is the ratio between the actual number of edges and the maximum possible number of edges in the graph.
  - The density of a graph is calculated as the number of edges divided by the maximum number of edges.
  - A graph is considered dense if its density is greater than 0.5 and sparse if its density is less than 0.1.

# Word2Vec

- Word2Vec translates words into vectors (also known as embeddings) using large datasets of text.
- Example

```
vec(king) = [−2.1, 4.1, 0.6]
vec(queen) = [−1.9, 2.6, 1.5]
vec(man) = [3.0, −1.1, −2]
vec(woman) = [2.8, −2.6, −1.1]
```

  - Euclidian distance
  - Cosine similarity
    - $\text{cosine similarity}(\vec{A}, \vec{B}) = cos(\theta) = \dfrac{\vec{A} \cdot \vec{B}}{||\vec{A}|| \cdot ||\vec{B}||}$
  - Ability to solve analogies
    - vec(king) - vec(man) + vec(woman) ≈ vec(queen)

# Continuous Skip-Grams Model

- Feed a single word to the model and predict the words around it.
- Increasing the range of context words leads to better embeddings but also increases the training time.

**Skip-gram**

# Creating Skip-Grams

- Skip-grams are implemented as pairs of words (target word, context word)
  - The target word is the input
  - The context word is the word to predict.
  - The number of skip grams for the same target word depends on a parameter called context size.

| Context Size | Text | Skip-grams |
|---|---|---|
| 1 | **the *train*** was late. | ('the', 'train') |
| | ***the* train *was*** late | ('train', 'the')<br>('train', 'was') |
| | the ***train* was *late*** | ('was', 'train')<br>('was', 'late') |
| | the train ***was* late** | ('late', 'was') |
| 2 | **the *train* *was*** late | ('the', 'train')<br>('the', 'was') |
| | ***the* train *was late*** | ('train', 'the')<br>('train', 'was')<br>('train', 'late') |
| | ***the train* was *late*** | ('was', 'the')<br>('was', 'train')<br>('was', 'late') |
| | the ***train* *was* late** | ('late', 'train')<br>('late', 'was') |

# The Skip-Gram Model

- Imagine that we have a sequence of $N$ words $w_1, w_2, \cdots, w_N$. The probability of seeing the word $w_2$ given the word $w_1$ is written $p(w_2 \mid w_1)$.

- The goal is to maximize the sum of every probability of seeing a context word given a target word in an entire text: $\frac{1}{N}\sum_{n=1}^{N}\sum_{-C\leq j\leq C, j\neq 0}\log p(w_{n+j} \mid w_n)$
  - $C$ is the size of the context vector.

- The basic skip-gram model uses the softmax function to calculate the probability of a context word embedding $h_c$ given a target word embedding $h_t$: $p(w_x \mid w_t) = \frac{e^{h_c h_t^T}}{\sum_{i=1}^{|V|} e^{h_i h_t^T}}$
  - $|V|$ is the vocabulary of size.

# The Skip-Gram Model (Cont.)

- Model structure
  - A projection layer with a weight matrix $W_{\text{embed}}$, takes a one-hot encoded-word vector as an input and returns the corresponding $N$-dim word embedding.
    - It acts as a simple lookup table that stores embeddings of a predefined dimensionality.
  - A fully connected layer with a weight matrix $W_{\text{output}}$, takes a word embedding as input and outputs $|V|$-dim logits.
    - A softmax function is applied to these predictions to transform logits into probabilities.
  - No activation function: Word2Vec is a linear classifier that models a linear relationship between words.

# The Skip-Gram Model (Cont.)

- Let's call $x$ the one-hot encoded-word vector the input.
- The corresponding word embedding can be calculated as a simple projection: $h = W_{\text{embed}}^{\text{T}} \cdot x$
- Using the skip-gram model, we can rewrite the previous probability as follows:

$$p(w_c \mid w_t) = \frac{e^{W_{\text{output}} \cdot h}}{\sum_{i=1}^{|V|} e^{W_{\text{output}_{(i)}} \cdot h}}$$

- The skip-gram model outputs a $|V|$-dim vector, which is the conditional probability of every word in the vocabulary: $\text{word2vec}(w_t) = \begin{bmatrix} p(w_1 \mid w_t) \\ p(w_2 \mid w_t) \\ \vdots \\ p(w_{|V|} \mid w_t) \end{bmatrix}$
- During training, these probabilities are compared to the correct one-hot encoded-target word vectors.
- The difference between these values (calculated by a loss function such as the cross-entropy loss) is backpropagated through the network to update the weights and obtain better predictions.

# Word2Vec Architecture

# Random Walks

- Random walks are sequences of nodes produced by randomly choosing a neighboring node at every step.

- Using random walks can generate meaningful sequences of nodes that act like sentences.
  - Sentences can be represented as graphs.

- Even if nodes are randomly selected, the fact that they often appear together in a sequence means that they are close to each other.

# Random Walks (Cont.)

- Each run
  - Input parameters
    - The starting node (start)
    - The length of the walk (length), i.e., the number of steps
  - Output: a node sequence with the given length
    - E.g., [‘0’, '4', '3', '6', '3', '4', '7', '8', '7', '4', '9']

- Repeated runs
  - If certain nodes are often found together in the same output, they are considered to be similar.

# DeepWalk Algorithm

- Goal: produce high-quality feature representations of nodes in an unsupervised way.

- Application: capture the similarity between nodes in a homophilic graph.
  - Core idea: under the network homophily hypothesis, nodes that are close to each other are similar and should have high similarity scores.
  - Process
    - Generate many random walks of a fixed length with uniformly distributed start nodes
    - Train a skip-gram model using the generated random walks to obtain node embeddings
    - Calculate the similarity score between two nodes based on their node embeddings

# Defining a Neighborhood

- The exploration process for the neighborhood of a node is called a sampling strategy.
  - Consider the three closest nodes in terms of connections
    - E.g., N(A) = {B, C, D}
    - Breadth-First Search (BFS) focuses on the local network around a node.
  - Select nodes that are not adjacent to previous nodes first.
    - E.g., N(A) = {D, E, F}
    - Depth-First Search (DFS) establishes a more macro view of the graph.

# Structural Equivalence v.s. Homophily

- Structural equivalence
  - Nodes are structurally equivalent if they share many of the same neighbors.
  - If nodes share many neighbors, their structural equivalence is higher.
- Homophily
  - Similar nodes are more likely to be connected.

- BFS is ideal to emphasize structural equivalence since this strategy only looks at neighboring nodes.
  - In these random walks, nodes are often repeated and stay close to each other.
- DFS emphasizes the opposite of homophily by creating sequences of distant nodes.
  - These random walks can sample nodes that are far from the source and thus become less representative.

# Introducing Biases in Random Walks

- Bias the randomness of these walks in Node2Vec
  - Promoting nodes that are not connected to the previous one (similar to DFS)
  - Promoting nodes that are close to the previous one (similar to BFS)

- Example
  - $j$: current node
  - $i$: previous node
  - $k$: future node
  - $\pi_{jk}$: unnormalized transition probability from node $j$ to node $k$
    - $\pi_{jk} = \alpha(i, k) \cdot \omega_{jk}$
    - $\omega_{jk}$: the weight of the edge from $j$ to $k$
    - $\alpha(i, k)$: the search bias between nodes $i$ and $k$

  - In DeepWalk, $\alpha(a, b) = 1$ for any pair of nodes $a$ and $b$.

# Introducing Biases in Random Walks (Cont.)

- In Node2Vec
  - The value of $\alpha(a, b)$ is defined based on the distance between the nodes

$$\alpha(a,b) = \begin{cases} 1/p, & \text{if } d_{ab} = 0 \\ 1, & \text{if } d_{ab} = 1 \\ 1/q, & \text{if } d_{ab} = 2 \end{cases}$$

  - $d_{ab}$ is the shortest path distance between nodes $a$ and $b$.
  - The return parameter $p$ approximates DFS
    - The walk starts from node $i$ and now arrives at node $j$. The probability of going back to the previous node is controlled by the return parameter $p$.
    - The higher it is, the more the random walk will explore new nodes instead of repeating the same ones and looking like DFS.
  - The in-out parameter $q$ approximates BFS.
    - The probability of going to node $k_2$ is controlled by the in-out parameter $q$.
    - The higher it is, the more the random walk will focus on nodes that are close to the previous one and look like BFS.
  - The unnormalized probability of going to $k_1$ is 1
    - Because this node is in the immediate neighborhood of the previous node, $i$.

# Application of Node2Vec

- With biased random walks, the implementation of Node2Vec is very similar to implementing DeepWalk.

- Node classification: classify each node of a graph as part of one of the two groups.
  - Use the node embeddings provided by Node2Vec as input to a machine-learning classifier (e.g., Random Forest).
  - Repeat this process 100 times and take the mean value to limit the randomness of the results.

# Parameter Analysis

- For a homophilic network, the homophily property is emphasized by DFS, which is encouraged by increasing the parameter *p*.
- Average accuracy and standard deviation for different values of *p* and *q*

| | p=1 | p=2 | p=3 | p=4 | p=5 | p=6 | p=7 |
|---|---|---|---|---|---|---|---|
| q=1 | 92.95% (± 4.61%) | 94.45% (± 4.19%) | 96.36% (± 4.69%) | 95.41% (± 4.14%) | 95.59% (± 4.30%) | 95.82% (± 4.67%) | 95.41% (± 3.94%) |
| q=2 | 93.64% (± 4.36%) | 93.95% (± 3.97%) | 95.09% (± 4.34%) | 95.55% (± 3.80%) | 96.27% (± 3.82%) | 96.18% (± 3.90%) | 97.45% (± 3.60%) |
| q=3 | 93.45% (± 3.82%) | 94.41% (± 4.11%) | 95.77% (± 3.59%) | 95.27% (± 3.63%) | 96.68% (± 3.90%) | 95.64% (± 3.69%) | 96.00% (± 3.82%) |
| q=4 | 94.14% (± 3.93%) | 94.14% (± 3.93%) | 95.45% (± 3.40%) | 95.05% (± 3.58%) | 95.95% (± 3.46%) | 96.41% (± 3.71%) | 95.59% (± 3.31%) |
| q=5 | 94.41% (± 3.68%) | 94.18% (± 3.64%) | 94.68% (± 3.58%) | 95.36% (± 3.75%) | 95.64% (± 3.34%) | 95.55% (± 3.58%) | 95.27% (± 4.01%) |
| q=6 | 94.91% (± 3.71%) | 94.55% (± 3.08%) | 94.59% (± 3.13%) | 95.05% (± 3.86%) | 95.77% (± 3.23%) | 94.55% (± 4.17%) | 95.05% (± 3.75%) |
| q=7 | 94.64% (± 4.03%) | 95.00% (± 3.78%) | 93.59% (± 3.97%) | 94.86% (± 3.67%) | 94.14% (± 3.87%) | 95.27% (± 3.74%) | 95.82% (± 3.38%) |

  - DeepWalk ( $p = 1$ and $q = 1$ ) performs worse than any other combination ( $1 \leq p \leq 7$ and $1 \leq q \leq 7$ ) for this dataset and shows how useful biased random walks can be.
  - High values of *p* lead to better performance, which validates that this is a homophilic network due to the connection between DFS and homophily.

# Classifying Nodes

- Classifying nodes with vanilla neural networks (vanilla NNs)
  - A vanilla neural network (also called a multi-layer perceptron) takes the tabular dataset of node features as its input.
- Classifying nodes with vanilla graph neural networks (vanilla GNNs)
  - A neural network layer corresponds to a linear transformation $h_A = x_A W^T$.
    - $h_A$ is the embedding of node $A$
    - $x_A$ is the input vector (node features) of node $A$ and $W$ is the weight matrix.
  - The graph linear layer can be written as $h_A = \sum_{i \in \mathcal{N}_A} x_i W^T$
    - $N_A$ is the set of neighbors of node $A$.
    - Equation variants: we could have a weight matrix $W_1$ dedicated to the central node, and another one $W_2$ for the neighbors.
  - The equation of the linear layer can be rewritten as $H = XW^T$.
    - $X$ is the input matrix.
  - The graph linear layer can be rewritten as $H = \tilde{A}^T X W^T$.
    - Updated adjacency matrix $\tilde{A} = A + I$, where the adjacency matrix $A$ contains the connections between every node in the graph.

# Vanilla NNs v.s. Vanilla GNNs

- Summary of accuracy scores with a standard deviation
  - In order to obtain comparable results, the same experiment is repeated 100 times for each model on each dataset.

| | **MLP** | **GNN** |
|---|---|---|
| **Cora** | 53.47% (±1.81%) | 74.98% (±1.50%) |
| **Facebook** | 75.21% (±0.40%) | 84.85% (±1.68%) |

  - The MLP has poor accuracy on Cora. It performs better on the Facebook Page-Page dataset but is still surpassed by our vanilla GNN in both cases.
  - These results show the importance of including topological information in node features.
  - Instead of a tabular dataset, GNN considers the entire neighborhood of each node, which leads to a 10-20% boost in terms of accuracy in these examples.

# Designing the Graph Convolutional Layer

- The vanilla GNN layer does not take into account the difference in the number of neighbors.
  - The embedding of a node $i$ is calculated as $h_i = \sum_{j \in \mathcal{N}_i} x_j W^T$.
  - It consists of a simple sum without any normalization coefficient.

- Simple solution: dividing the embedding by the number of neighbors.
  - New GNN layer: $h_i = \frac{1}{\deg(i)} \sum_{j \in \mathcal{N}_i} x_j W^T$, where $\deg(A)$ is the degree of node $A$
  - The degree matrix $D$ counts the number of neighbors for each node.
  - The inverse of this matrix $D^{-1}$ directly gives us the normalization coefficients, $\frac{1}{\deg(i)}$.
  - Self-loops are added to the degree matrix $\tilde{D}^{-1} = (D + I)^{-1}$ like self-loops added to the graph by $\tilde{A} = A + I$.

# Example

- $$D = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}, D^{-1} = \begin{bmatrix} \frac{1}{3} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} \end{bmatrix}, \tilde{D}^{-1} = \begin{bmatrix} \frac{1}{4} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{3} \end{bmatrix}$$

- $$\tilde{D}^{-1}\tilde{A} = \begin{bmatrix} \frac{1}{4} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{3} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

- $$\tilde{A}\tilde{D}^{-1} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{4} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{3} \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{4} & \frac{1}{2} & 0 & 0 \\ \frac{1}{4} & 0 & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{4} & 0 & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

# Designing the Graph Convolutional Layer (Cont.)

- How to put the matrix of normalization coefficients into the formula $H = \tilde{A}^T X W^T$?
  - $\tilde{D}^{-1}\tilde{A}XW^T$ will normalize every row of features.
  - $\tilde{A}\tilde{D}^{-1}XW^T$ will normalize every column of features.
  - Naturally, the first option looks more appealing because it nicely normalizes neighboring node features.

- Weakness of the first option: features from nodes with many neighbors spread very easily, unlike features from more isolated nodes.

- Solution: a hybrid normalization to counterbalance this effect.
  - Higher weights are assigned to nodes with few neighbors using the formula $H = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}XW^T$
  - In terms of individual embeddings, this operation can be written as

$$h_i = \frac{1}{\sqrt{\deg(i)}} \sum_{j \in \mathcal{N}_i} \frac{1}{\sqrt{\deg(j)}} x_j W^T$$

# Comparing Graph Convolutional and Graph Linear Layers

- Number of nodes with specific node degrees in the Cora dataset
  - This imbalanced distribution looks exponential with a heavy tail
  - It ranges from 1 neighbor (485 nodes) to 168 neighbors (1 node).
- Number of nodes with specific node degrees in the Facebook Page-Page dataset
  - This distribution of node degrees looks even more skewed, with a number of neighbors that ranges from 1 to 709.

# Graph Convolutional v.s. Graph Linear Layers

- Summary of accuracy scores with standard deviation

| | MLP | GNN | GCN |
|---|---|---|---|
| **Cora** | 53.47% (±1.81%) | 74.98% (±1.50%) | 80.17% (±0.61%) |
| **Facebook** | 75.21% (±0.40%) | 84.85% (±1.68%) | 91.54% (±0.28%) |

  - These high scores are attributed to the wide range of node degrees in these two datasets.
  - By normalizing features and considering the number of neighbors of the central node and its own neighbors, the GCN gains a lot of flexibility.

- Compared to the vanilla GNN, the main feature of the GCN is that it considers node degrees to weigh its features.

# Introducing the Graph Attention Layer

- Idea: some nodes are more important than others
  - In a graph convolutional layer, nodes with few neighbors were more important than others due to the normalization coefficient $\frac{1}{\sqrt{\deg(i)}\sqrt{\deg(j)}}$, but it only takes into account node degrees.
  - The goal of the graph attention layer is to produce weighting factors that also consider the importance of node features.
- Graph attention operator $h_i = \sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} x_j$
  - $h_i$ is the embedding of node $i$.
  - $x_j$ is the input vector (features) of node $j$ and $N_i$ is the set of neighbors of node $i$
  - $\mathbf{W}$ is the regular weight matrix.
  - The attention score $\alpha_{i,j}$ between the nodes $i$ and $j$ are weighting factors representing the importance between a central node $i$ and a neighbor $j$.
    - Self-attention: attention scores are calculated implicitly by comparing inputs to each other.

# Self-Attention

- Linear transformation: attention coefficient $a_{ij} = W_{att}^T[\mathbf{W}x_i \parallel \mathbf{W}x_j]$
  - $x_i$ and $x_j$ are vectors representing node features of nodes $i$ and $j$.
  - $\mathbf{W}$ is a classic shared weight matrix to compute hidden vectors.
  - The attention weight matrix $W_{att}^T$ is a dedicated learnable weight matrix.
- Activation function: $e_{ij} = LeakyReLU(a_{ij})$
  - Leaky Rectified Linear Unit (ReLU) fixes the dying ReLU problem.
  - Variant: Exponential Linear Unit (ELU)
    - The exponential version of the Leaky ReLU
    - $f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), \quad \alpha > 0, & \text{if } x \leq 0 \end{cases}$
  - $e_{ij}$ is not normalized.

- Softmax normalization: $\alpha_{ij} = softmax_j(e_{ij}) = \dfrac{exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} exp(e_{ik})}$
  - $\mathcal{N}_i$ is the neighboring nodes of node $i$.
  - $\alpha_{ij}$ is not very stable.

# Self-Attention (Cont.)

- Multi-head attention
  - Calculating attention scores with multi-head attention

  - Calculate multiple embeddings with their own attention scores $h_i^k = \sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W} x_j$
    - For each produced embedding $h_i^k$, $k$ is the index of the attention head.
  - Two ways of combining these multiple embeddings
    - Averaging: $h_i = \frac{1}{n} \sum_{k=1}^{n} h_i^k = \frac{1}{n} \sum_{k=1}^{n} \sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k x_j$ for the last layer of the network.
    - Concatenation: $h_i = \|_{k=1}^{n} h_i^k = \|_{k=1}^{n} \sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k x_j$ for a hidden layer.

# Improved Graph Attention Layer

- Original Graph Attentional Operator, GAT
  - $$\alpha_{ij} = \frac{exp(LeakyReLU(W_{att}^T[\mathbf{W}x_i \parallel \mathbf{W}x_j]))}{\sum_{k \in \mathcal{N}_i} exp(LeakyReLU(W_{att}^T[\mathbf{W}x_i \parallel \mathbf{W}x_j]))}$$
  - Only computes a static type of attention
  - Implement a GAT in NumPy

- Modified graph attentional operator, GATv2
  - $$\alpha_{ij} = \frac{exp(W_{att}^T LeakyReLU(\mathbf{W}[x_i \parallel x_j]))}{\sum_{k \in \mathcal{N}_i} exp(W_{att}^T LeakyReLU(\mathbf{W}[x_i \parallel x_j]))}$$
  - Computes a strictly more expressive dynamic attention.
  - Implement a GATv2 in PyTorch Geometric

- GATv2 consistently outperforms the GAT.

# Implementing a GAT in NumPy

- Translate the original graph attention operator in terms of matrices.
  - Graph attention operator $h_i = \sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} x_j$
  - $H = \tilde{A}^T W_\alpha X \mathbf{W}^T$, where $\alpha$ is a matrix that stores every $\alpha_{ij}$.
    - The size of W is (dimension of a hidden vector, the number of nodes)

- The size of $W_{att}^T$ is (1, 2 $\times$ dimension of a hidden vector)
  - $a_{ij} = W_{att}^T [\mathbf{W} x_i \parallel \mathbf{W} x_j]$ is a scalar.

- Example

# Implementing a GATv2 in PyTorch Geometric

- Nodes with few neighbors are more difficult to classify correctly.
  - In general, the higher the node degree, the better the accuracy score.
  - Accuracy score per node degree (CiteSeer)

- Summary of accuracy scores with standard deviation
  - According to Schur et al. in Pitfalls of Graph Neural Network Evaluation

| Accuracy | GCN | GAT |
|---|---|---|
| Cora | 81.9% (±0.8%) | 82.8% (±0.6%) |
| CiteSeer | 69.5% (±0.9%) | 71.0% (±0.6%) |

# Transformer

# Attention

- Recurrent neural network

(a) RNN

(b) Time-layered representation of (a)

- From RNN to Transformer
  - Attention has replaced recurrence functions requiring an increase in the number of parameters as the distance between two words increases.

- Attention mechanism
  - A “word-to-word” (or token-to-token) operation.
  - Determines how each word is related to all other words in a sequence, including the word being analyzed itself.

# Model Architecture

- Encoder stack
  - Input embedding
  - Positional encoding (PE)
  - Sublayer 1: Multi-head attention
  - Sublayer 2: Feedforward Network (FFN)

- Decoder stack
  - Output embedding
  - Positional encoding
  - Attention layers
  - FFN sublayer
  - Post-Layer Normalization (Post-LN)
  - Linear layer

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. Attention is All You Need. Conference on Neural Information Processing Systems (NIPS), Long Beach, CA, USA, December 2017

# Encoder Stack

- Efficient constraint
  - The output of every sublayer of the model has a constant dimension $d_{\text{model}} = 512$, including the embedding layer and the residual connections.

- Input embedding
  - A tokenizer will transform a sentence/word into tokens.
  - The skip-grams model of the word2vec embedding approach
- Positional encoding (PE)
- A multi-headed attention mechanism
  - From layers 1 to 6, each layer learns from the previous layer and explores different ways of associating the tokens in the sequence.
- A fully connected position-wise feedforward network

# Positional Encoding

- Tokenizer
  - Use sine and cosine functions to generate different frequencies for the positional encoding for each position and each dimension $i$ of the $d_{\text{model}} = 512$ of the word embedding vector
    - $PE_{(pos,2i)} = \sin \dfrac{pos}{10000^{2i/d}\text{model}}$
    - $PE_{(pos,2i+1)} = \cos \dfrac{pos}{10000^{2i/d}\text{model}}$

- Adding positional encoding to the embedding vector
  - Creating independent positional vectors would have a high cost on the training speed of the Transformer and make attention sublayers overly complex to work with.
  - Idea: add a positional encoding value to the input embedding instead of having additional vectors to describe the position of a token in a sequence.
    - $\text{ie} \cdot d^2_{\text{model}} + \text{pe}$

# Multi-Head Attention

- The multi-head attention sublayer contains eight heads and is followed by post-layer normalization, which will add residual connections to the output of the sublayer and normalize it.
- Inside each head $h_n$ of the attention mechanism, the "word" matrices have three representations:
  - A query matrix ($Q$) that has a dimension of $d_q = 64$, which seeks all the key-value pairs of the other "word" matrices.
  - A key matrix ($K$) with a dimension of $d_k = 64$.
  - A value matrix ($V$) with a dimension of $d_v = 64$.
- To obtain $Q$, $K$, and $V$, we must train the model with their respective weight matrices $Q_w$, $K_w$, and $V_w$
  - $Q = x \cdot Q_w$, $K = x \cdot K_w$, $V = x \cdot V_w$
- Attention is defined as scaled dot-product attention, which is represented in the following equation in which we plug $Q$, $K$, and $V$:
  - $$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d_k}}\right)V$$

# Post-Layer Normalization

- The goal of the residual connections is to make sure critical information is not lost.

- Layer normalization is described as LayerNormalization(x + Sublayer(x))
  - Sublayer($x$) is the sublayer itself.
  - $x$ is the information available at the input step of Sublayer(x).
- LayerNormalization($v$) $= \gamma \dfrac{v - \mu}{\sigma} + \beta$, where $v = x$ + Sublayer($x$)
  - $\mu$ is the mean of $v$ of dimension $d$.
  - $\sigma$ is the standard deviation v of dimension d.
  - $\gamma$ is a scaling parameter.
  - $\beta$ is a bias vector.

# Feedforward Network (FFN)

- The FFNs in the encoder and decoder are fully connected.

- The FFN is a position-wise network.
  - Each position is processed separately and in an identical way.

- The FFN contains two layers and applies a ReLU activation function.

- The input and output of the FFN layers is $d_{\text{model}} = 512$, but the inner layer is larger with $d_{ff} = 2048$.

- $\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$

# Decoder Stack

- Output embedding and positional encoding
- Attention layers
- Feedforward network (FFN) sublayer
- Post-Layer Normalization (Post-LN)
- Linear layer
  - The output layer maps the outputs of the model to the size of the vocabulary to produce the raw logits of the prediction.
- The raw logits of the output can go through a softmax function, apply the values obtained to the tokens in the vocabulary, and choose the best probable token for the task requested.

# Attention Layers

- The masked multi-head attention sublayer 1 only lets attention apply to the positions up to and including the current position.
  - The future words are hidden from the Transformer, and this forces it to learn how to predict.

- The multi-head attention sublayer 2
  - Only attends to the positions up to the current position the Transformer is predicting to avoid seeing the sequence it must predict.
  - Draws information from the encoder by taking encoder ($K$, $V$) into account during the dot-product attention operations.
  - Draws information from the masked multi-head attention sublayer 1 (masked attention) by also taking sublayer 1($Q$) into account during the dot-product attention operations.
- The decoder uses the trained information of the encoder.

- The input of the self-attention multi-head sublayer of a decoder
  - Input_Attention=(Output_decoder_sublayer-1(Q),Output_encoder_layer(K,V))

# Question & Answer