Understanding the workload

Cheng Zhang and Aaron Zhao, Imperial College London

Introduction

The rule of thumb

- People only care about the models at any given time.
  - GPTs - (Transformer based, decoder-only)
  - Diffusion (Image generation)
  - CLIP - (Contrastive-learning)
  - SAM - (Segmentation foundation model)
  - Whisper (Neural ASR)
- You cannot trade-off model performance too much
  - Common performance engineers logic is to get $10\times$ speed-up with a 5% decrease in accuracy.
  - 5% accuracy drop on standard image classification benchmarks mean you use models that are from the previous generation!

Charactersitcs of workloads

The characteristics come from two aspects: the data and the model

I will breakdown the survey of different workload characteristics for different fields, this includes

- Computer Vision
- Natural Language Processing
- Graph Representation Learning

I will go through them in a fairly fast pace, it is expected you do extra readings following the links in the course wiki.

Computer Vision Workloads

Basic Building Blocks

We will mainly focus on tasks on 2D images.

Basic building blocks for CV networks are:

- Convolution
- Linear

We will later look at popular vision network building blocks

- Residual Blocks
- UNet
- Vision Transformer

We will look at the following tasks

- Classification
- Segmentation

Basic Building Blocks

Let’s unify our language, for each layer, we consider

- an input activation tensor (feature in), $\boldsymbol{X_l}$ for layer l.
- the free parameters tensor (weights), $\boldsymbol{W_l}$
- an output activation tensor (feature out), $\boldsymbol{X_{l+1}}$

Convolution

torch.nn.Conv2d takes input with size (N, C_{in}, H, W), and outputs (N, C_{out}, H, W), let's assume kernel size is K, stride is 1, and we are dealing a normal convolution (no grouping, etc.).

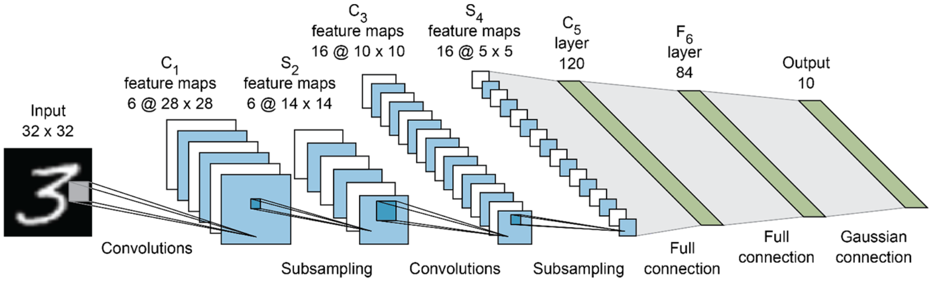
An Example If batch size is 1, for the first convolution, we have

$N = 1, C_{in} = 1, H = 32, W = 32, C_{out} = 6$

The convolution operator (f_{cov}) transforms an input volume (N, C_{in}, H, W) to an output volume (N, C_{out}, H, W) :

$$f_{conv} : \mathcal{R}^{1\times1\times32\times32} \rightarrow \mathcal{R}^{1\times6\times32\times32}$$

Convolution (ii)

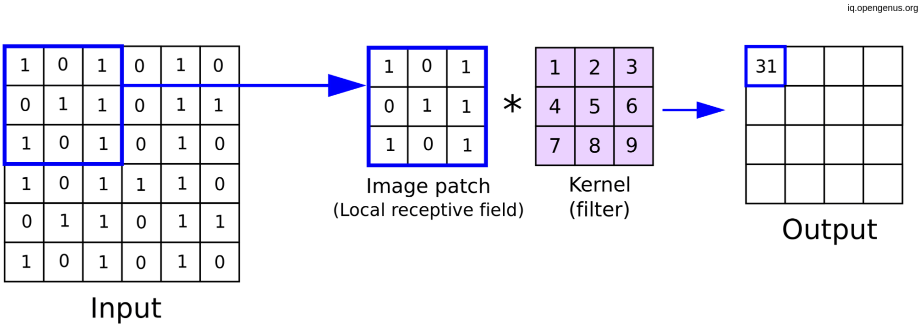


An Example Weights for a convolutional layer has the shape (C_{out}, C_{in}, K, K), where K is the kernel size.

Alternatively, you can view it as we have $(C_{out} \times C_{in})$ independent filters with each filter at the size of $K \times K$.

We take the image patch and multiply it with a filter. We then slide it across the whole input volume.

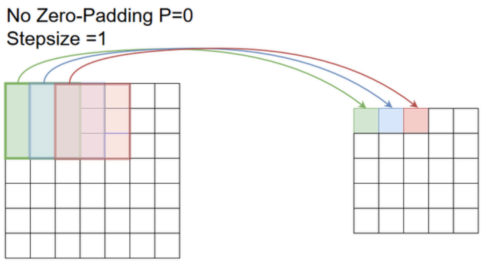
Convolution (iii)

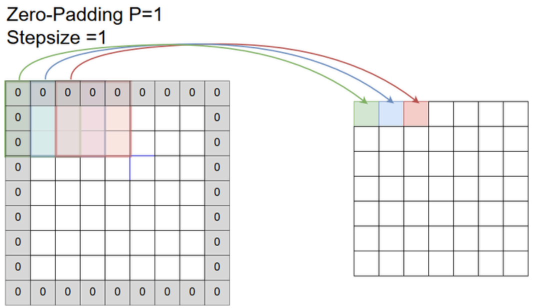


Striding For each filter, we then slide it across the whole input volume.

See in reading materials for more animations and mechanism about padding and striding.

Convolution (iv)





Convolution - The Actual Code

```
// output channels
for (co=0; co\<C\_out; co ++)
  // slide across the input volume
  for (h=0; h\<H; h++)
    for (w=0; w\<W; w++)
      // input channels
      for (ci=0; ci\<C\_in, ci++)
        // kernels
        for (kh=0; kh\<K; kh++)
          for (kw=0; kw\<K; kw++)
            Xnew[co,h,w] += X[ci,h+kh,w+kw]*w[ci,co,kh,kw]
```

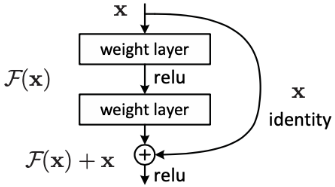
Linear

torch.nn.Linear simply performs

$$y = x\boldsymbol{W}^T + b$$

where, $\boldsymbol{W} \in \mathcal{R}^{i \times o}$ and i and o are the input and output feature dimensions.

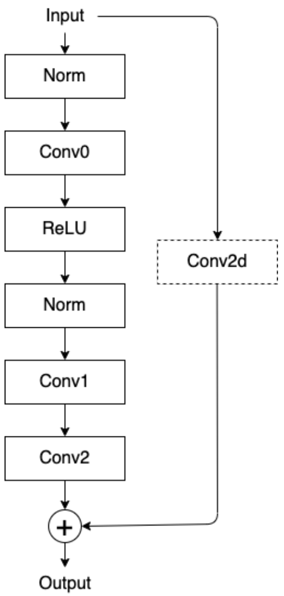
Vision Building Blocks: Residual Connections A residual connection (or a shortcut) provides an additional path for data to reach later parts of the network without doing any additional computation.



Vision Building Blocks: ResidualBlocks

- The parameterized layers only need to learn the different between the two.
- Gradient can have access to all layers, and it helps to mitigate the gradient vanishing problem with deep networks.
- Depending on whether *Conv0* is strided, a convolution block is added in shortcut.

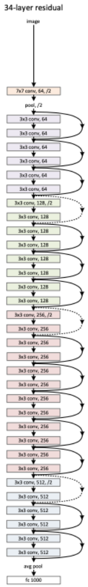
Vision Building Blocks: ResidualBlocks (ii)



ResNet and image classification

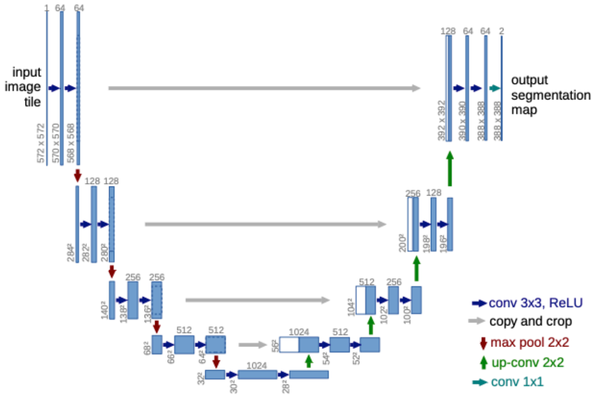
- One can stack a few ResidualBlocks to build different ResNets (eg. ResNet50, ResNet32)
- Image classification takes an image as an input and produce and produces a one-hot vector to determine the class of the image.

ResNet and image classification (ii)



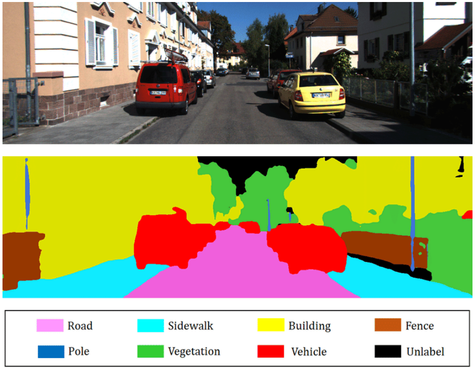
U-net and segmentation

- U-net builds residual connections in a special way, there is a shortcut at every resolution, from its encoder to the decoder.
- Downsample uses *MaxPool2D*, and upsample uses *ConvTranspose2d*.



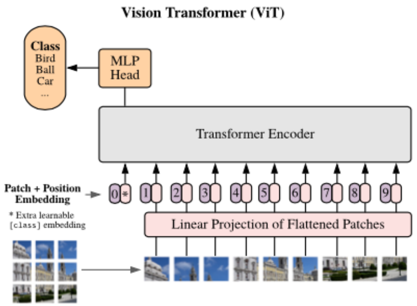
Semantic segmentation

- Semantic Segmentation categorizes each pixel in an image into a class or object.
- That’s why each the output has the same size as the input.
- Applications in Autonomous Driving (pedastrains, cars…), Robotics (object positions…), Medical Imaging (tumor or not)…



Vision Transformer

- A ‘kind of’ new idea of dealing with images.
- Instead of treating an image as an input volume, what if we make it a sequence?
- Split an image or an input feature volume into fixed-size patches, linearly embed each of them, so they are now a sequence!


Vision Transformer (ViT)
Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Patch + Position Embedding
* Extra learnable [class] embedding
0 *
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches

Natural Language Processing Workloads

NLP Building Blocks

We will take a look at the modern NLP building blocks (not LSTMs or GRUs).

- Attention layers
- The original transformer model (6-layer)
- BERT
- LLaMa

Tokens and Embeddings

The core idea is to transform texts to a sequence of vectors, so that a model can consume as inputs.

- Tokenization: it divides a sentence into individual units, known as tokens. Tokens can be words or punctuation marks.
- These tokens are then transformed into numbers.
- Map these numbers into continuous vectors, also called word embedding (can be very tricky)!

Most existing word embeddings are learned using the Continuous Skip-gram Modeling.

We will skip the detail of this training, since we only care about what happens at inference time for now.

Tokens and Embeddings

Why we need word embeddings?

In the latent space, we want

$$x_{people} - x_{person} \approx x_{cars} - x_{car}$$

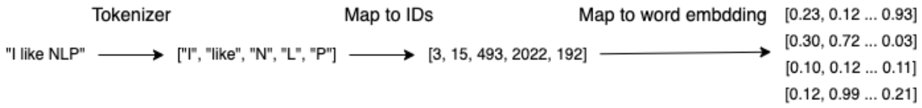
But

$$x_{person} \neq x_{car}$$

Interesting fact, in the word embedding latent space, because of the skip-gram modeling, words such as 'like' and 'hate' are clustered very closely!

- Tokenize input text.
- Map them to numerical ids.
- Map each id to the vector space, $\boldsymbol{X} \in \mathcal{R}^{N \times D}$, where N is the sequence length and D is the dimensionality of the word embedding.

Tokens and Embeddings (ii)



Attention

- Q, K, V are projected through a linear transformation with dimension d_k.
- They have size $\mathcal{R}^{N \times d_k}$, where N is the sequence length.
- *softmax* simply scales the output $\frac{e^{x_i}}{\sum_{j=0}^{n-1} e^{x_j}}$ to provide a probability.

$$Atten(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Let's say $d_k = 1$ and $N = 3$ for simplicity, we have

$$Q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \end{bmatrix}$$

Attention (ii)

$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V = \begin{bmatrix} a_{00}v_0 + a_{01}v_1 + a_{02}v_2 \\ a_{10}v_0 + a_{11}v_1 + a_{12}v_2 \\ a_{20}v_0 + a_{21}v_1 + a_{22}v_2 \end{bmatrix}$$

We simply computed a bunch of coefficients, controlled by learnable parameters, to re-scale our V!

Attention: A Conceptual View

$$V = \begin{pmatrix} \text{"I"} \\ \text{"like"} \\ \text{"football"} \end{pmatrix}$$

My result might be

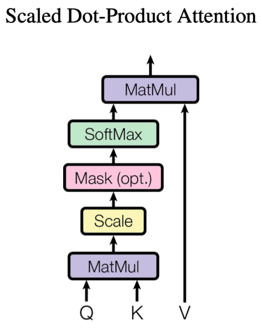
$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V = \begin{bmatrix} 0.01v_0 + 0.02v_1 + 0.97v_2 \\ 0.02v_0 + 0.03v_1 + 0.95v_2 \\ 0.03v_0 + 0.03v_1 + 0.96v_2 \end{bmatrix}$$

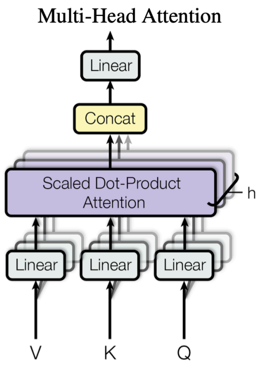
All entry may now pay ‘attention’ to v_2 (football)!

Multi-head Attention

We normally have a number of attention heads in parallel, this is also known as multi-head attention.

The parallelism in learning is similar to the number of parallel filter banks in CNNs!

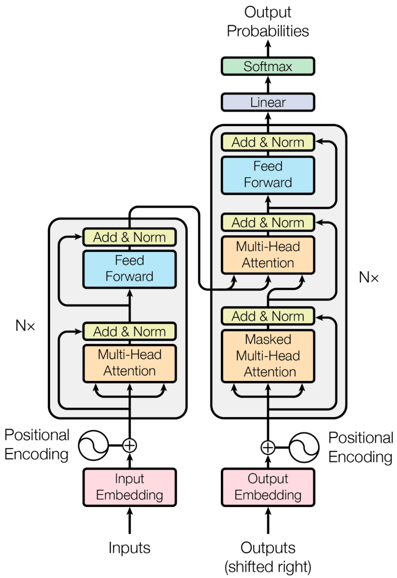




Canonical Transfomer

- The transformer model has two parts, the encoder part and the decoder part.
- Positional embedding adds the positional information to each token.
- Decoder takes not only encoded inputs but also the current output values.
- Mainly demonstrated on Machine Translation tasks (measured in BLEU scores).

Canonical Transfomer (ii)



BERT

- Bidirectional Encoder Representations from Transformers.
- The same as the Transformer architecture, but only the encoder part, duplicated many times.
- Uses MLM (masked language modeling) to pretrain the model and then fine-tune on other tasks, this is known as the pre-trian and then fine-tune paradigm.

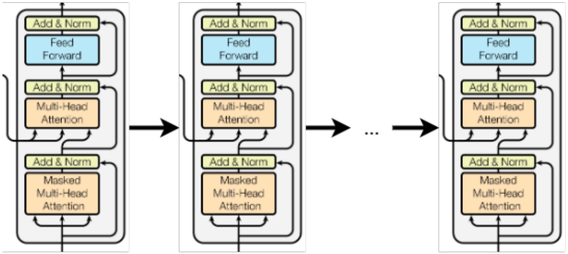
T5 models

- Similar to the Transformer architecture with both an encoder-decoder structure, but much larger in size!
- The support of a longer sequence length because of the relative positional encoding. Think about relative position between tokens instead of absolute positioning. This would have to modify the self-attention mechanism slightly, detail about this is in reading material.

LLaMA

- Normally (but not always), Bidirectional models (trained with MLM) are paired with encoder-decoder architecture.
- Decoder-only architecture are normally unidirectional (eg. GPT, OPT …).
- Uses CLM (causal language modeling) to pre-train the model.
- We then apply prompts to apply pre-trained models to downstream tasks in a zero-shot manner. This is known as the pre-train and then prompting paradigm.
- Will cover in more detail in the next lecture

LLaMA (ii)


Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
...

Graph Representation Learning Workloads

GNN Building Blocks

Graph Neural Networks (GNNs) are used to handle tasks that have graphs as inputs.

- GCN: Graph Convolutional Networks
- GAT: Graph Attention Networks

are the most popular building blocks.

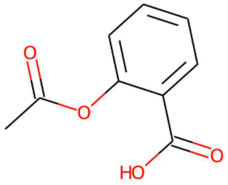
Graph Learning

In graph representational learning, we are handling graph data.

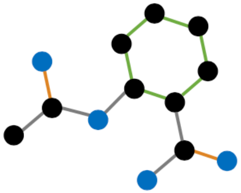
- Graph-level tasks: predict certain properties of a graph, this is normally on small-scale graphs (eg. proteins).
- Node/edge-level tasks: predict the properties of certain nodes and edges (eg. recommendation systems).

There are also other graph tasks (such as graph generation).

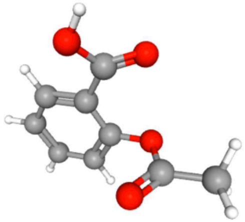
Graph Learning



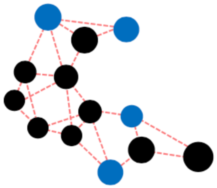
Aspirin in 2D



2D topological graph



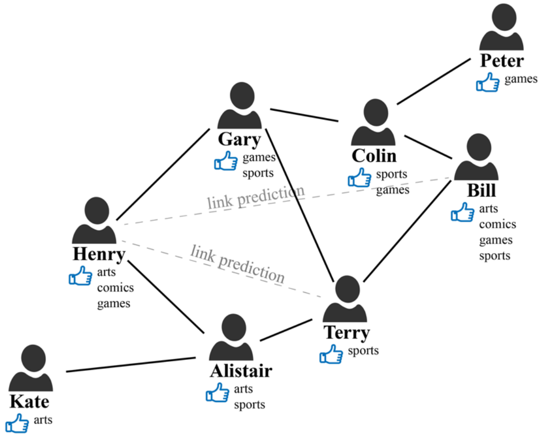
Aspirin in 3D



3D geometric graph

- ● Carbon
- 🔵 Oxygen
- — Single bond
- — Double bond
- — Aromatic bond
- - - - Interatomic Distance

Graph Learning (ii)



The Message Passing Framework

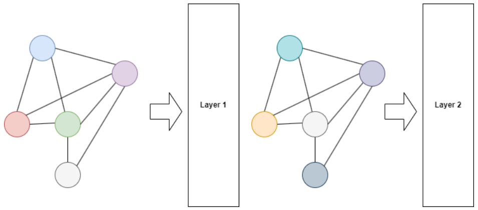
A graph is defined as $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of nodes, and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ denotes the set of edges.

- $A \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ is the adjacency matrix, with each entry a_{ij} representing an edge (if any) between nodes i and j; note that this is different from the conventional $\{0, 1\}^{|\mathcal{V}| \times |\mathcal{V}|}$ adjacency matrix format, since there are different types of bonds (i.e., single, double, triple, aromatic).
- $H \in \mathbb{R}^{|\mathcal{V}| \times d}$ is the feature matrix, $\boldsymbol{h}_i \in \mathbb{R}^d$ is the d-dimensional features of node i.

The Message Passing Framework

All the GNNs we consider can be abstracted as Message Passing Neural Networks (MPNNs).

An MPNN operation iteratively updates the node features $\boldsymbol{h}_i^{(l)} \in \mathbb{R}^d$ from layer l to layer $l + 1$ via propagating messages through neighbouring nodes $j \in \mathcal{N}_i$:



The Message Passing Framework

Both MESSAGE and UPDATE are learnable functions.

$\mathcal{N}_i = \{j \mid (i, j) \in \mathcal{E}\}$ is the (1-hop) neighbourhood of node i

$\bigoplus$ is a permutation-invariant local neighbourhood aggregation function, such as sum, mean or max.

$$\boldsymbol{h}_i^{(l+1)} = \text{UPDATE}\left(\boldsymbol{h}_i^{(l)}, \bigoplus_{j \in \mathcal{N}_i} \text{MESSAGE}\left(\boldsymbol{h}_i^{(l)}, \boldsymbol{h}_j^{(l)}, \boldsymbol{e}_{ij}\right)\right)$$

The graph embedding $\boldsymbol{h}_G \in \mathbb{R}^d$ can be obtained via a READOUT function:

$$\boldsymbol{h}_G = \text{READOUT}_{i \in \boldsymbol{V}}\left(\boldsymbol{h}_i^{(k)}\right)$$

Graph Convolutional Networks (GCN)

- c_{ij} is a normalisation constant for each edge $\mathcal{E}_{ij}$ which originates from using the symmetrically normalised adjacency matrix $\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}$ with $\boldsymbol{D}_{ii} = \sum_j A_{ij}$ is the degree matrix.
- $\boldsymbol{W}^{(l)}$ is a learnable weight matrix
- σ is a non-linear activation function (eg. ReLU)

$$\boldsymbol{h}_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}_i} c_{ij}\boldsymbol{W}^{(l)}\boldsymbol{h}_j^{(l)}\right)$$

This is actually very similar to the convolution in computer vision!

Graph Attention Networks

GAT applies attention-based neighbourhood aggregation as its aggregation function to obtain sufficient expressive power.

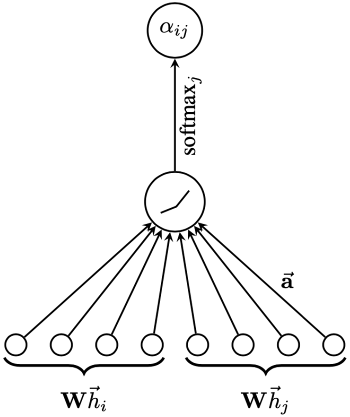
$$\forall j \in \mathcal{N}_i, \alpha_{ij} = \frac{\exp\left(\text{LeakyReLU }\left(\boldsymbol{a}\left[\boldsymbol{W}\boldsymbol{h}_i \parallel \boldsymbol{W}\boldsymbol{h}_j\right]\right)\right)}{\sum_{k \in \mathcal{N}_i} \exp(\text{LeakyReLU }(\boldsymbol{a}[\boldsymbol{W}\boldsymbol{h}_i \parallel \boldsymbol{W}\boldsymbol{h}_k]))}$$

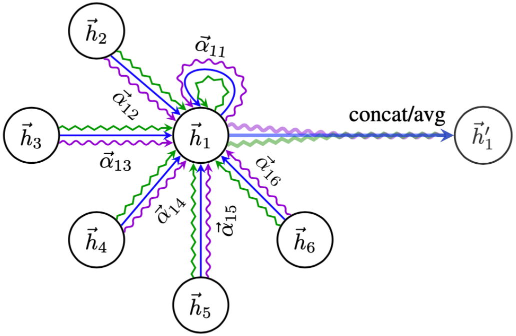
$\parallel$ denotes concatenation and $\boldsymbol{a}$ is a learnable weight vector for the attention.

$$\boldsymbol{h}_i^{(l+1)} = \parallel_{k=1}^{K} \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \boldsymbol{W}^k \boldsymbol{h}_j^{(l)}\right)$$

This is actually very similar to the self-attention in NLP!

Graph Attention Networks


α_{ij}
softmax_j
$\vec{\mathbf{a}}$
$\mathbf{W}\vec{h}_i$
$\mathbf{W}\vec{h}_j$


$\vec{h}_1$
$\vec{h}_2$
$\vec{h}_3$
$\vec{h}_4$
$\vec{h}_5$
$\vec{h}_6$
$\vec{\alpha}_{11}$
$\vec{\alpha}_{12}$
$\vec{\alpha}_{13}$
$\vec{\alpha}_{14}$
$\vec{\alpha}_{15}$
$\vec{\alpha}_{16}$
concat/avg
$\vec{h}'_1$