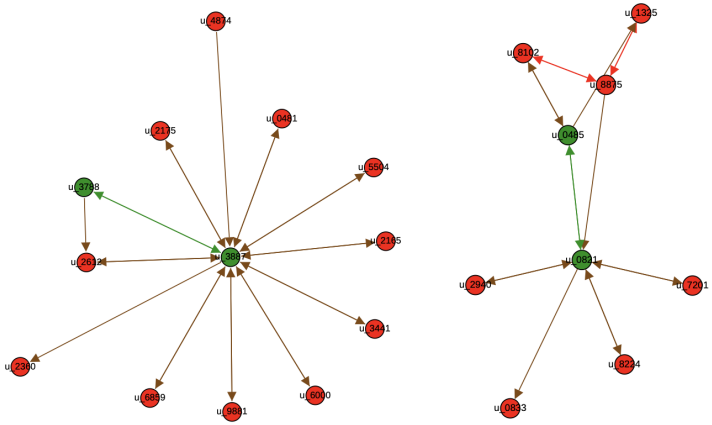
Intro to Graph Representation Learning

Dmitry Eremeev

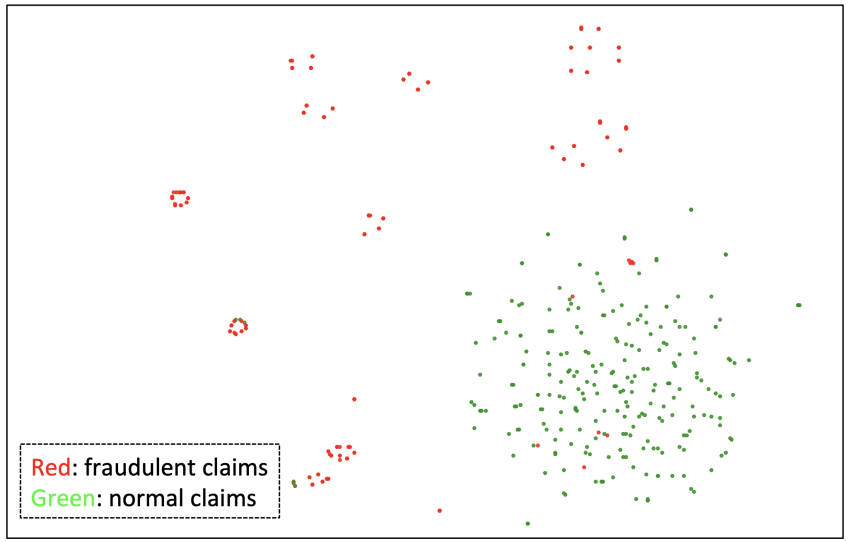
Moscow, 2021

Motivation

- Fraudulent claims in insurance.
- Graph of client calculations and transactions.

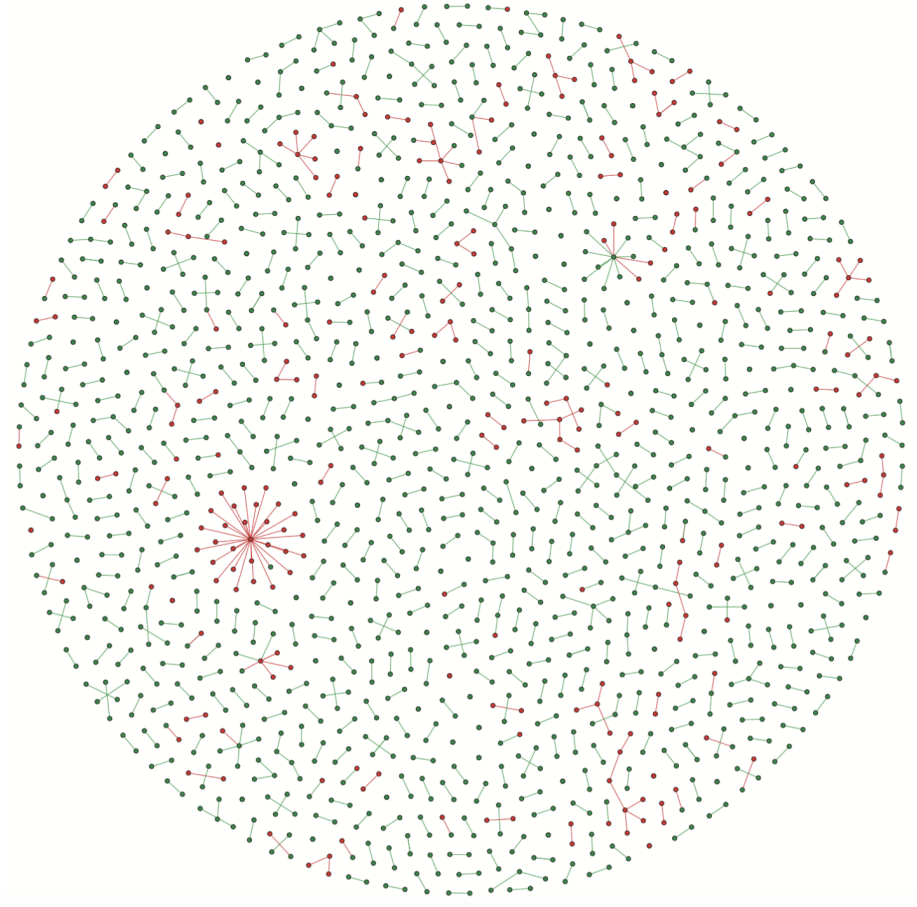


(a) Case 1 (b) Case 2



source

Motivation

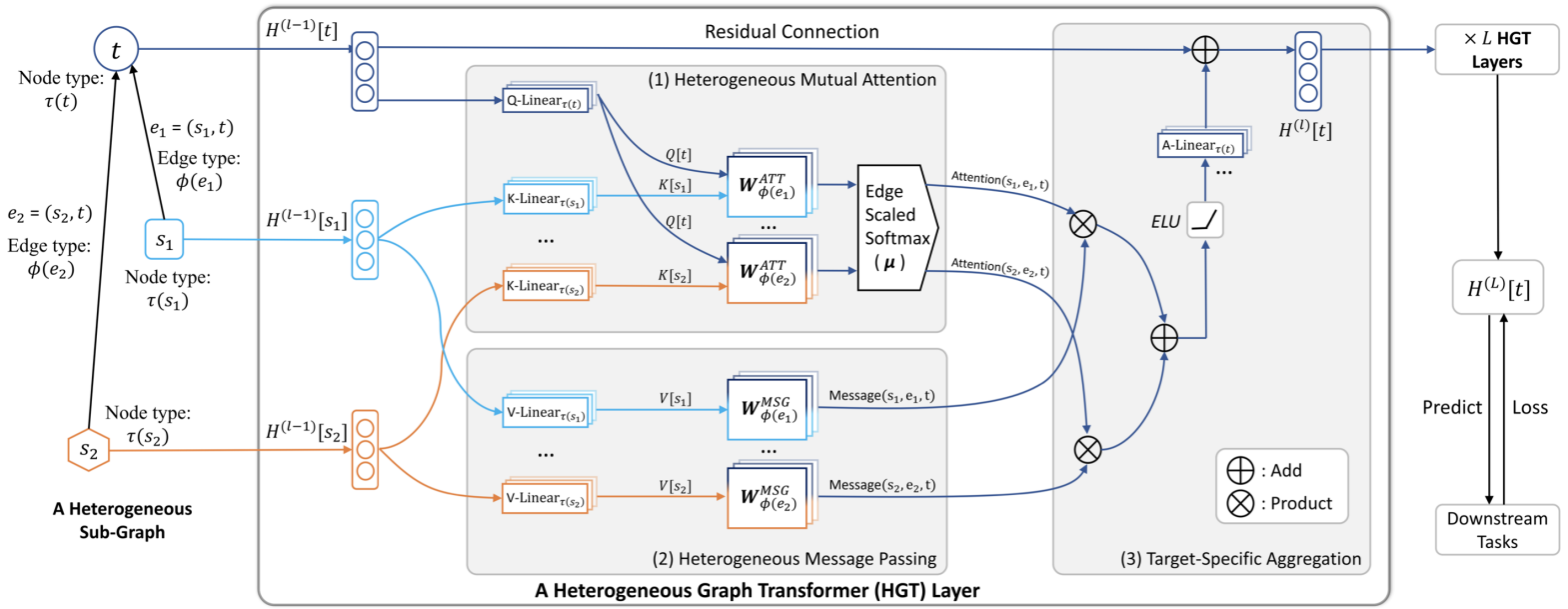


source

Heterogeneous Graph Transformer

arXiv:2003.01332 [cs.LG]

github.com/acbull/pyHGT

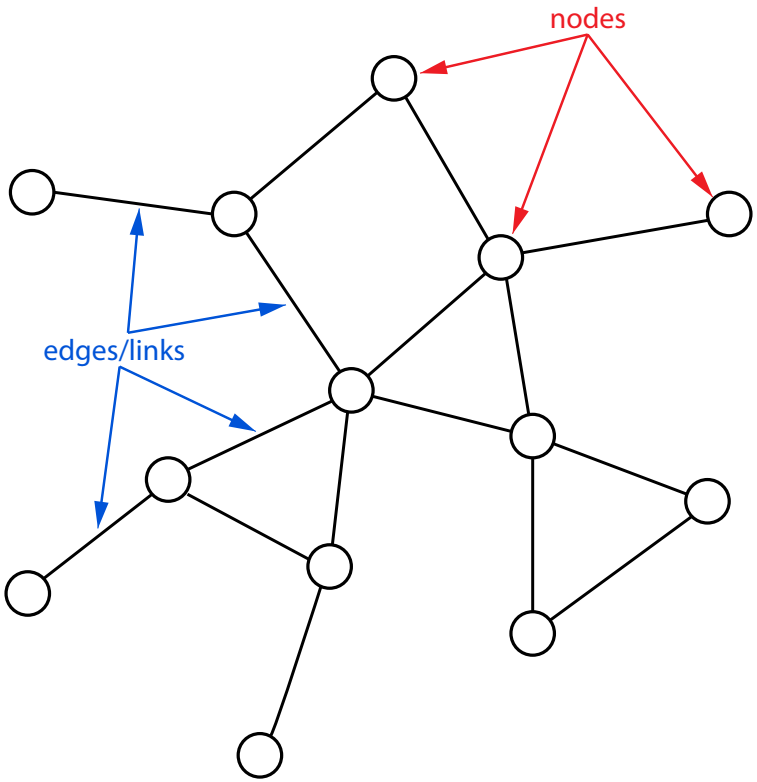


Intro

Graph definition

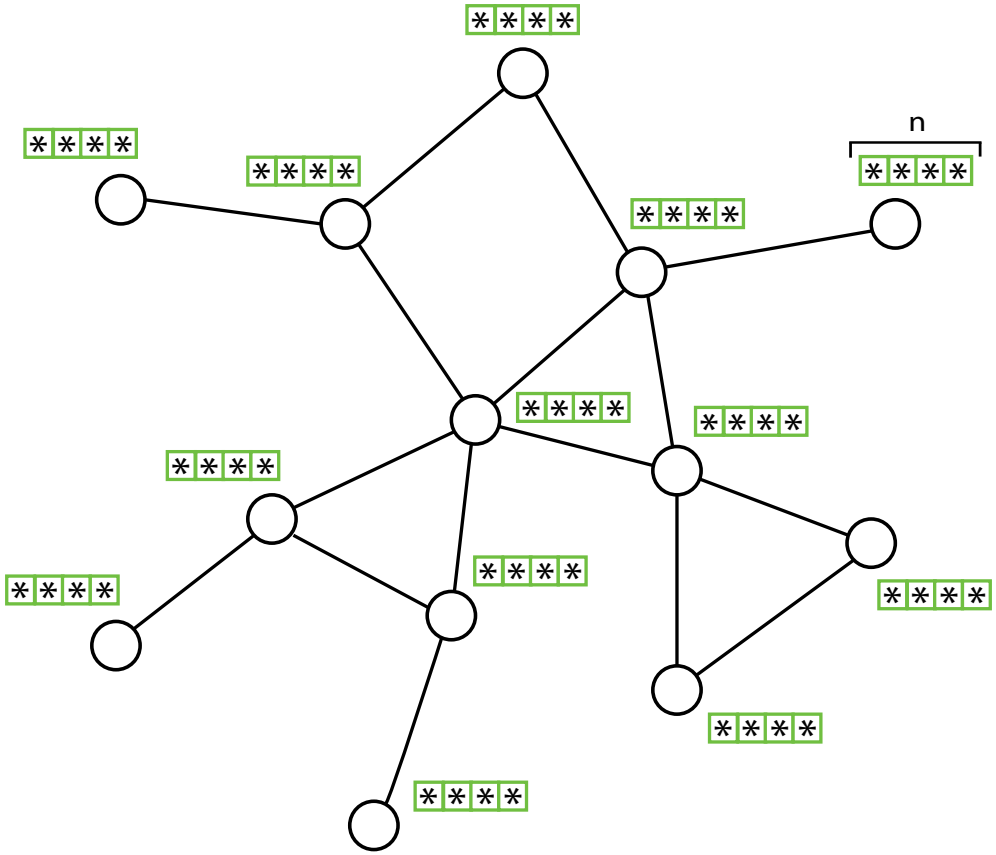
Graph $G = (V, E)$ is a pair of 2 sets:

- V - vertices [nodes],
- $E \in V \times V$ - edges [links] (paired vertices).



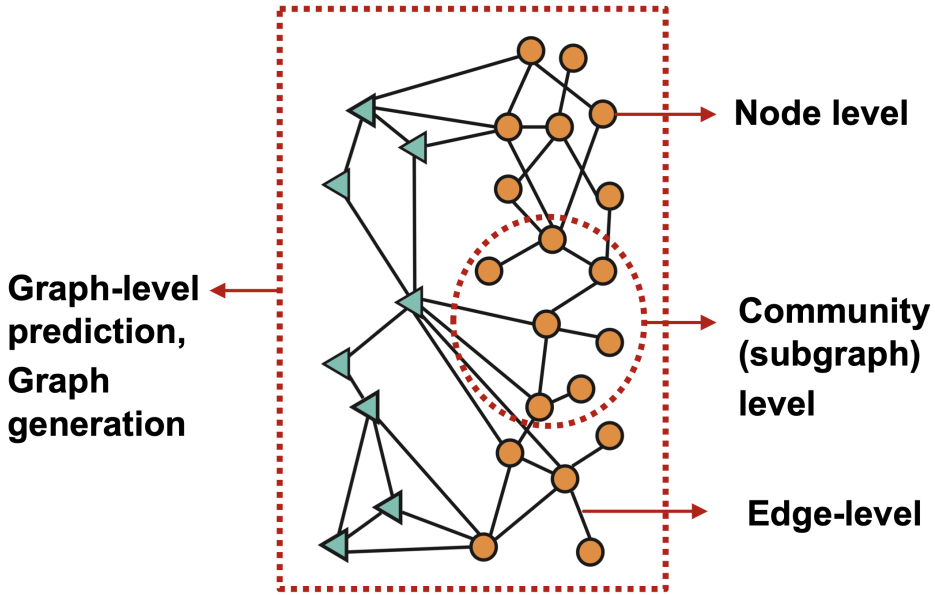
Graph features

Each node could be described by feature vectors



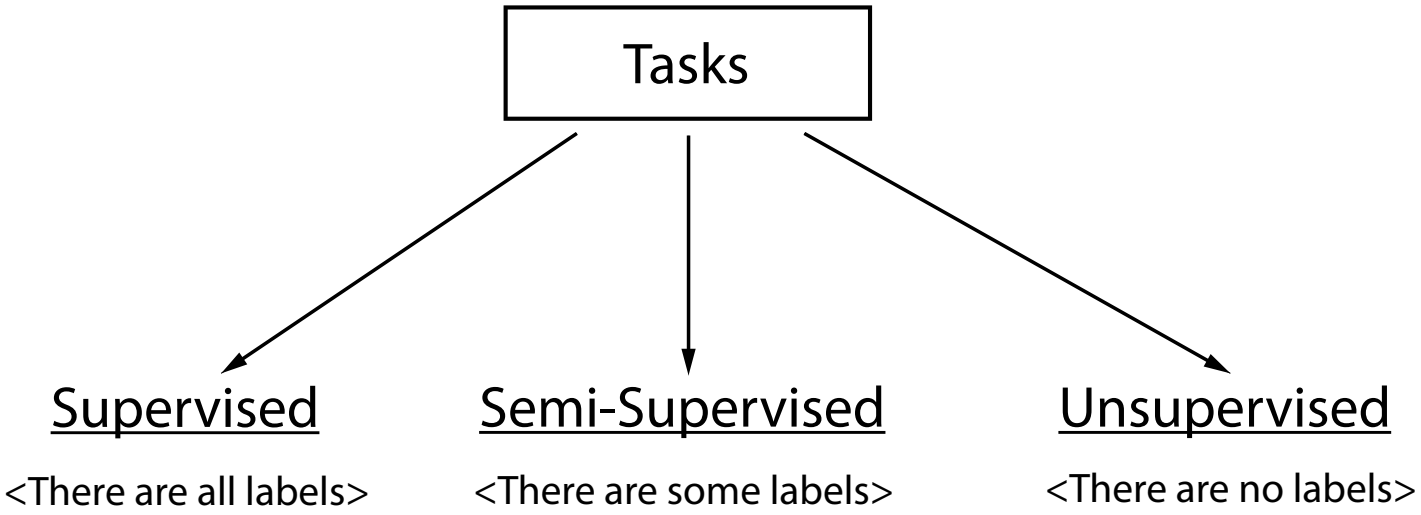
Types of tasks:

- Node Classification
- Link prediction
- Community detection
- Learning over the whole graph



source

Tasks

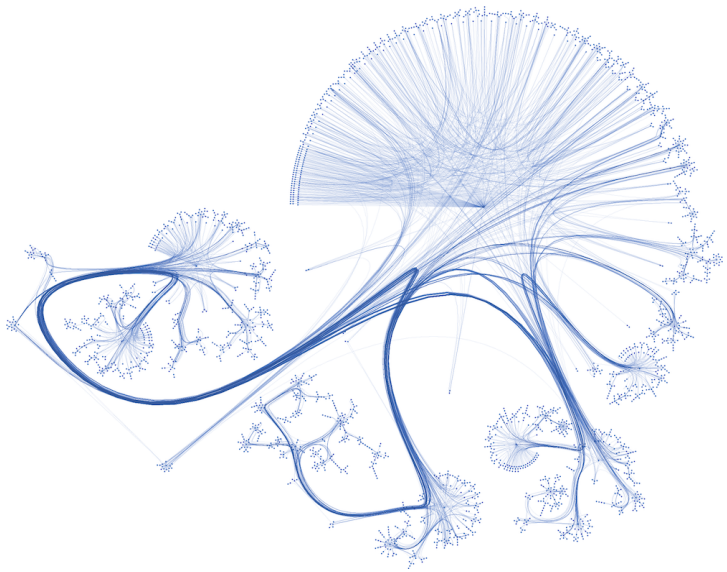

Tasks
Supervised
<There are all labels>
Semi-Supervised
<There are some labels>
Unsupervised
<There are no labels>

Networkx library

- „Hand-Crafted“ features.
- Node Degree, Node Centrality, Node Clustering Coefficient, etc.
- Check the Stanford CS224W course.

Cora dataset

- 2708 scientific publications, classified into 7 classes.
- 5429 links - citations.
- Features: presence of the words from dictionary.



source

Open Graph Benchmark

OGB

Get Started | Updates | KDD Cup | Datasets | Leaderboards | Paper | Team | Github

The classification accuracy on the test and validation sets. The higher, the better.

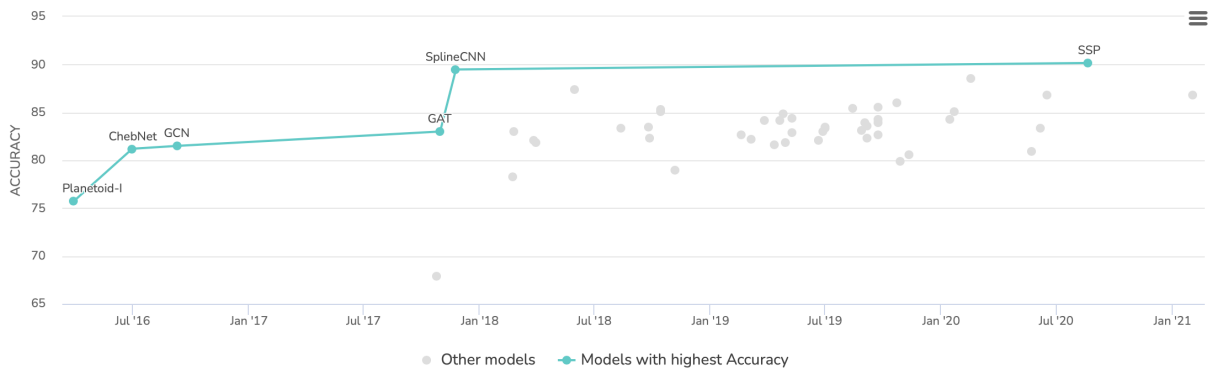
Package: >=1.1.1

Rank	Method	Test Accuracy	Validation Accuracy	Contact	References	#Params	Hardware	Date
1	**MLP + C&S**	0.8418 ± 0.0007	0.9147 ± 0.0009	Horace He (Cornell)	Paper, Code	96,247	GeForce RTX 2080 (11GB GPU)	Oct 27, 2020
2	**Linear + C&S**	0.8301 ± 0.0001	0.9134 ± 0.0001	Horace He (Cornell)	Paper, Code	10,763	GeForce RTX 2080 (11GB GPU)	Oct 27, 2020
3	**UniMP**	0.8256 ± 0.0031	0.9308 ± 0.0017	Yunsheng Shi (PGL team)	Paper, Code	1,475,605	Tesla V100 (32GB)	Sep 8, 2020
4	**Plain Linear + C&S**	0.8254 ± 0.0003	0.9103 ± 0.0001	Horace He (Cornell)	Paper, Code	4,747	GeForce RTX 2080 (11GB GPU)	Oct 27, 2020
5	**DeeperGCN+FLAG**	0.8193 ± 0.0031	0.9221 ± 0.0037	Kezhi Kong	Paper, Code	253,743	NVIDIA Tesla V100 (32GB GPU)	Oct 20, 2020
6	**GAT+FLAG**	0.8176 ± 0.0045	0.9251 ± 0.0006	Kezhi Kong	Paper, Code	751,574	GeForce RTX 2080 Ti (11GB GPU)	Oct 20, 2020

paperswithcode

Node Classification on Cora

Leaderboard Dataset



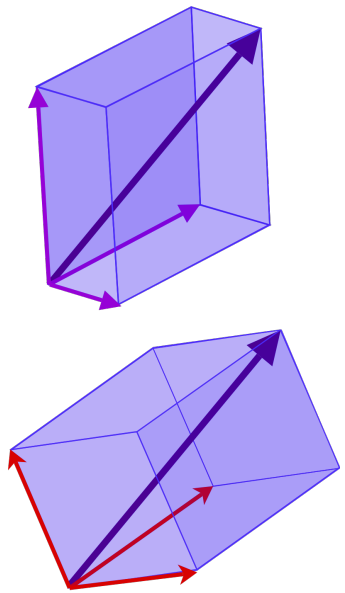
View Accuracy All models Edit

RANK	MODEL	ACCURACY	TRAINING SPLIT	VALIDATION	EXTRA TRAINING DATA	PAPER	CODE	RESULT	YEAR
1	SSP	90.16%			×	Optimization of Graph Neural Networks with Natural Gradient Descent			2020
2	SplineCNN	89.48%			×	SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels			2017

- Vectors: $\mathbf{v} \in V$, V - vector space.
- Basis: $B = \{\mathbf{b}_i\}$ - set of vectors that spans the whole vector space V:

$$\mathbf{v} = a_1\mathbf{b}_1 + a_2\mathbf{b}_2 + \cdots + a_n\mathbf{b}_n$$

$$\mathbf{v} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \in \mathbb{R}^n$$



- Matrix $\mathbf{A}$ with m rows and n columns:

$$\mathbf{A} = \begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ m \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \cdots & n \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \end{array} \in \mathbb{R}^{m \times n}$$

- Matrix as a set of vectors:

$$\mathbf{A} = \begin{bmatrix} — & \mathbf{a_1} & — \\ — & \mathbf{a_2} & — \\ & \vdots & \\ — & \mathbf{a_m} & — \end{bmatrix}$$

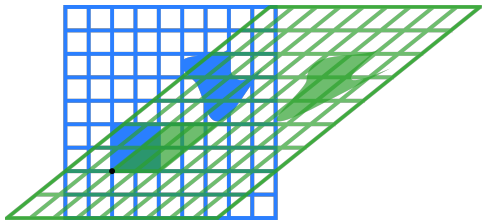
- Matrices define linear transformations:

$$\mathbf{A} \in \mathbb{R}^{m \times n} \iff \mathbb{R}^n \to \mathbb{R}^m$$

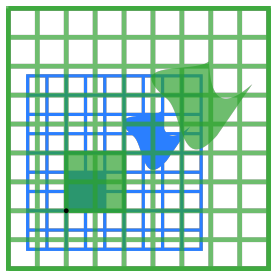
$$\mathbf{v} \mapsto \mathbf{v}' = \mathbf{A}\mathbf{v}.$$

- Examples: $\mathbb{R}^2 \to \mathbb{R}^2$

$$\mathbf{A} = \begin{bmatrix} 1 & 1.25 \\ 0 & 1 \end{bmatrix}$$



$$\mathbf{A} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$



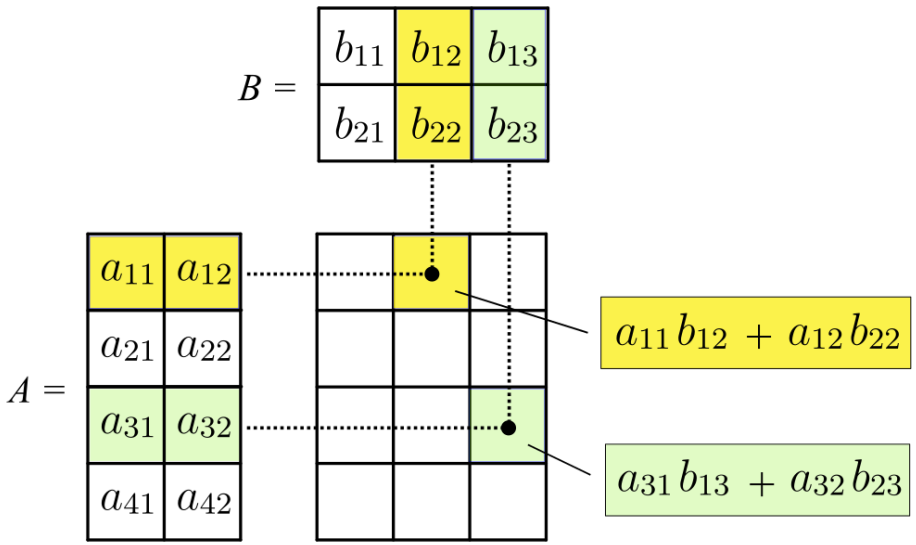
- Sum of matrices:

$$\mathbf{A} + \mathbf{B} := \begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ m \end{array} \begin{array}{c} \begin{array}{ccc} 1 & \cdots & n \end{array} \\ \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & \cdots & a_{2n} + b_{2n} \\ \vdots & \cdots & \vdots \\ a_{m1} + b_{m1} & \cdots & a_{mn} + b_{mn} \end{bmatrix} \end{array}$$

- Matrix multiplication:

$$\underbrace{\mathbf{A}}_{m\times p} \cdot \underbrace{\mathbf{B}}_{p\times n} = \underbrace{\mathbf{C}}_{m\times n}$$

- Elements of **C** are defined by scalar products of **A** rows and **B** columns:



27.3. A *tensor of type* (p, q) on V is an element of the space

$$T_p^q(V) = \underbrace{V^* \otimes \cdots \otimes V^*}_{p \text{ factors}} \otimes \underbrace{V \otimes \cdots \otimes V}_{q \text{ factors}}$$

isomorphic to the space of linear functions on $V \times \cdots \times V \times V^* \times \cdots \times V^*$ (with p factors V and q factors V^*). The number p is called the *covariant valency* of the tensor, q its *contravariant valency* and $p + q$ its *total valency*. The *vectors* are tensors of type $(0, 1)$ and *covectors* are tensors of type $(1, 0)$.

Let a basis $e_1, \ldots, e_n$ be selected in V and let $e_1^*, \ldots, e_n^*$ be the dual basis of V^*. Each tensor T of type (p, q) is of the form

$$T = \sum T_{i_1 \ldots i_p}^{j_1 \ldots j_q} e_{i_1}^* \otimes \cdots \otimes e_{i_p}^* \otimes e_{j_1} \otimes \cdots \otimes e_{j_q}; \tag{1}$$

the numbers $T_{i_1 \ldots i_p}^{j_1 \ldots j_q}$ are called the *coordinates of the tensor* T in the basis $e_1, \ldots, e_n$.

Let us establish how coordinates of a tensor change under the passage to another basis. Let $\varepsilon_j = Ae_j = \sum a_{ij} e_i$ and $\varepsilon_j^* = \sum b_{ij} e_i^*$. It is easy to see that $B = (A^T)^{-1}$, cf. 5.3.

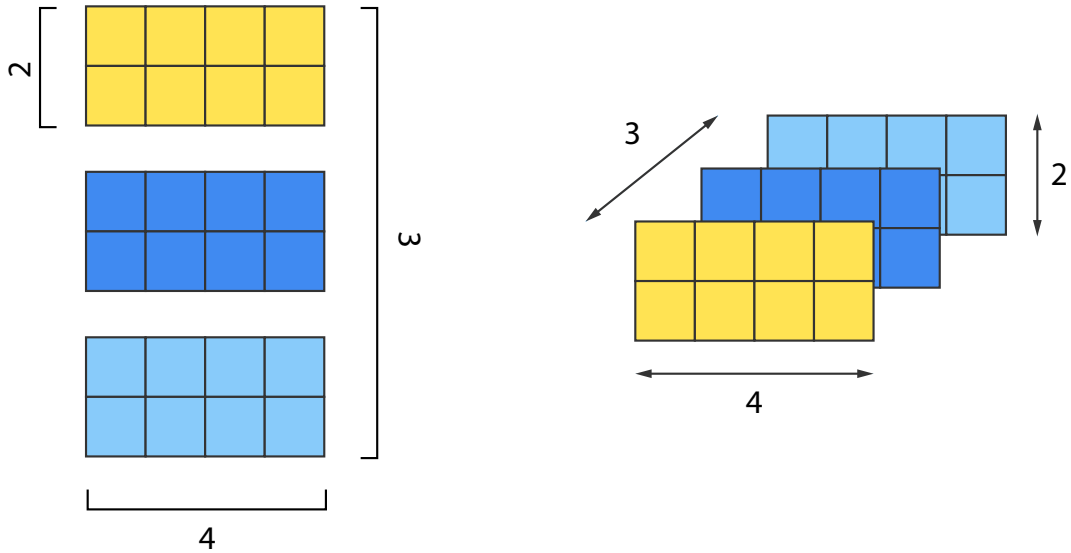
Introduce notations: $a_j^i = a_{ij}$ and $b_i^j = b_{ij}$ and denote the tensor (1) by $\sum T_\alpha^\beta e_\alpha^* \otimes e_\beta$ for brevity. Then

$$\sum T_\alpha^\beta e_\alpha^* \otimes e_\beta = \sum S_\mu^\nu \varepsilon_\mu^* \otimes \varepsilon_\nu = \sum S_\mu^\nu b_\alpha^\mu a_\nu^\beta e_\alpha^* \otimes e_\beta,$$

i.e.,

$$T_{i_1 \ldots i_p}^{j_1 \ldots j_q} = b_{i_1}^{l_1} \ldots b_{i_p}^{l_p} a_{k_1}^{j_1} \ldots a_{k_q}^{j_q} S_{l_1 \ldots l_p}^{k_1 \ldots k_q} \tag{2}$$

- Tensor of shape $[3, 2, 4]$ (a set of matrices $\mathbb{R}^{2\times 4}$):



- PyTorch: pytorch.org
- PyTorch Geometric: github.com/rusty1s/pytorch_geometric

Tensor multiplication in PyTorch:

In 1
```
import torch
```

In 2
```
a = torch.tensor([[1, 2],
                  [3, 4]])
print(a.shape)
a
```

```
torch.Size([2, 2])
```

Out 2
```
tensor([[1, 2],
        [3, 4]])
```

In 3
```
b = torch.tensor([[2, 0],
                  [0, 2]])
print(b.shape)
b
```

```
torch.Size([2, 2])
```

Out 3
```
tensor([[2, 0],
        [0, 2]])
```

In 4
```
torch.matmul(a, b)
```

Out 4
```
tensor([[2, 4],
        [6, 8]])
```

PyTorch supports broadcasting:

In 5
```
# tensor of shape (2, 2, 2)
a = torch.randint(0, 10, (2,2,2))
print(a.shape)
a
```

```
torch.Size([2, 2, 2])
```

Out 5
```
tensor([[[0, 3],
         [7, 9]],

        [[4, 6],
         [5, 0]]])
```

In 10
```
b = torch.tensor([[2, 0, 1],
                  [0, 2, 1]])

# broadcast: [2, 2, 2] x [2, 3]  --> [2, 2, 3]
c = torch.matmul(a, b)
print(c.shape)
c
```

```
torch.Size([2, 2, 3])
```

Out 10
```
tensor([[[ 8, 16, 12],
         [10, 16, 13]],

        [[18,  4, 11],
         [ 8, 14, 11]]])
```

Computing gradients:

In 1
```
import torch
```

In 2
```
x = torch.tensor(10.0, requires_grad=False)
y = torch.tensor(5.0, requires_grad=True)
```

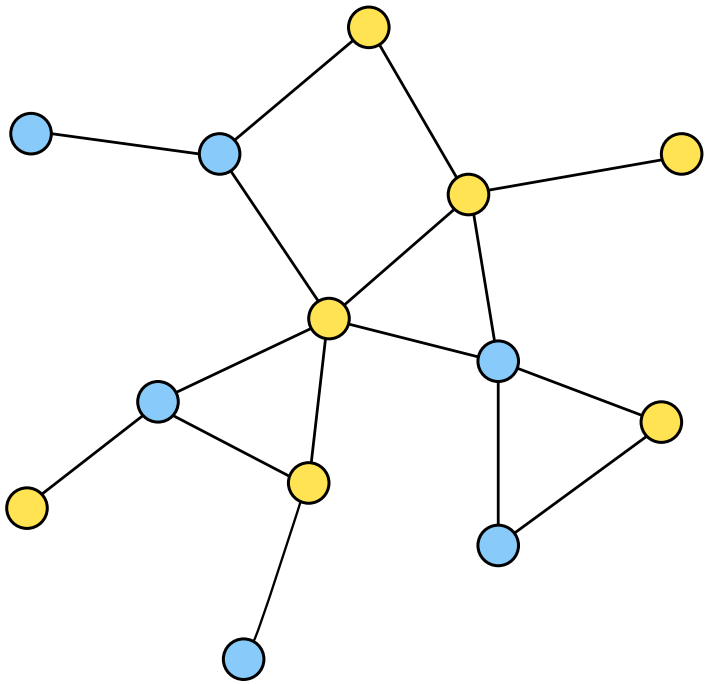
In 3
```
print(x)
print(y)
```

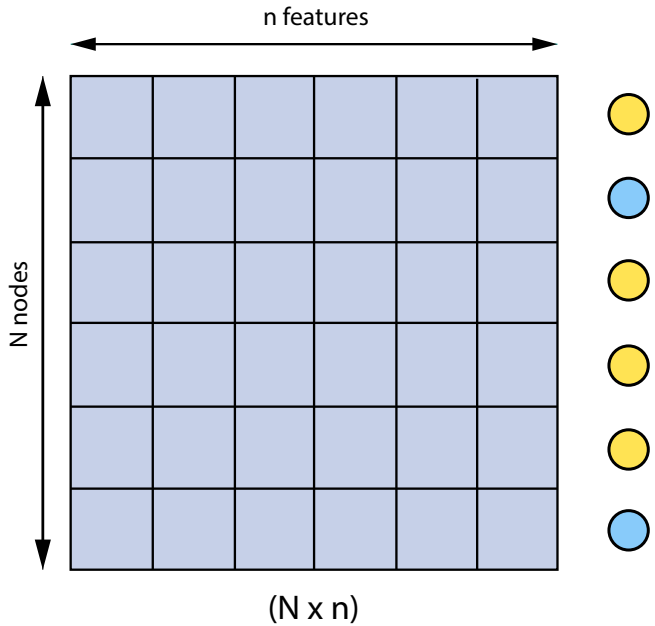
```
tensor(10.)
tensor(5., requires_grad=True)
```

In 4
```
# compute the gradient
# z(x, y) = x*y^2
z = x*y*y
z.backward()  # Autograd calculates and stores the gradient of z w.r.t to tensors
y.grad  # extract the value of gradient
```

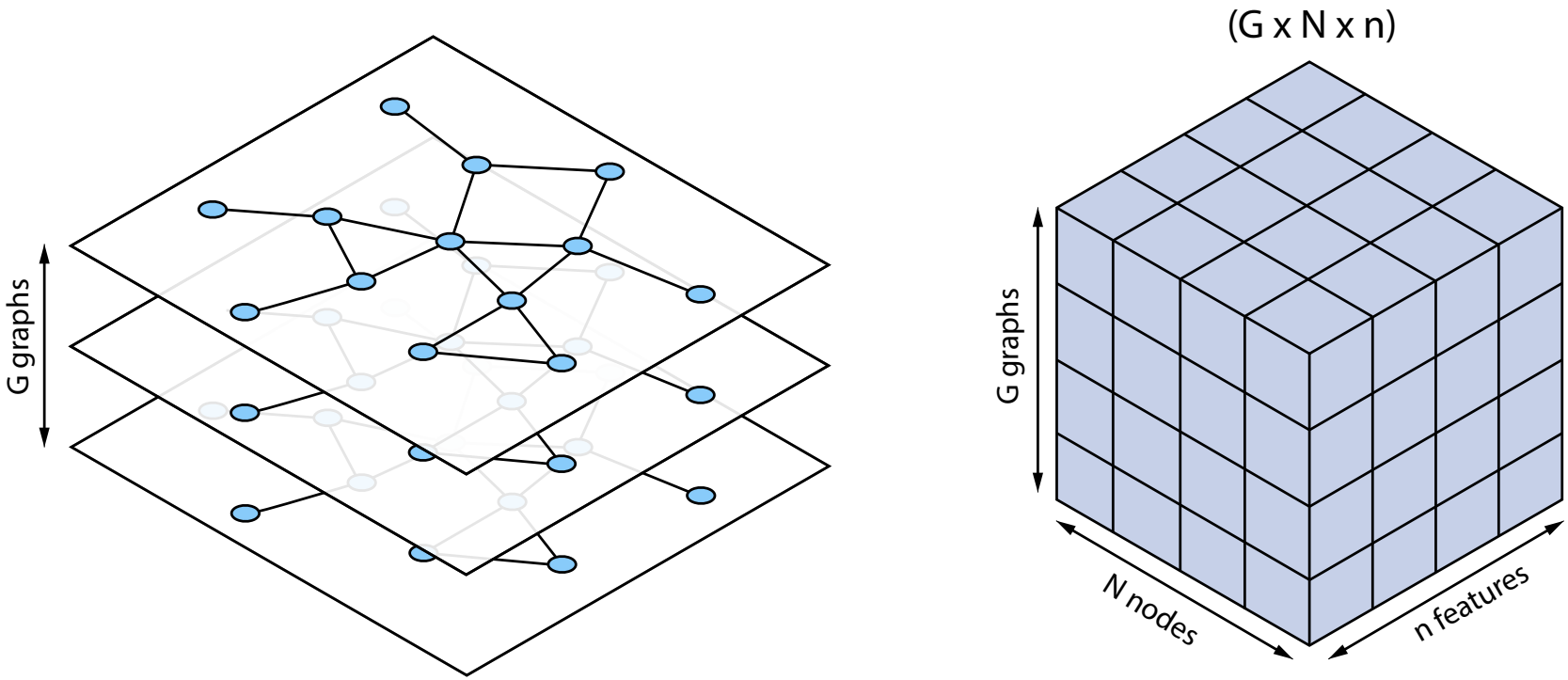
Out 4
```
tensor(100.)
```

Matrix representation





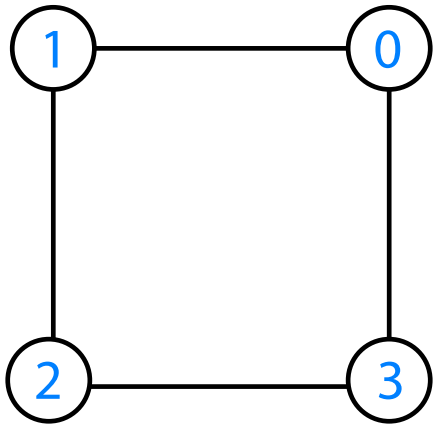
Tensor representation



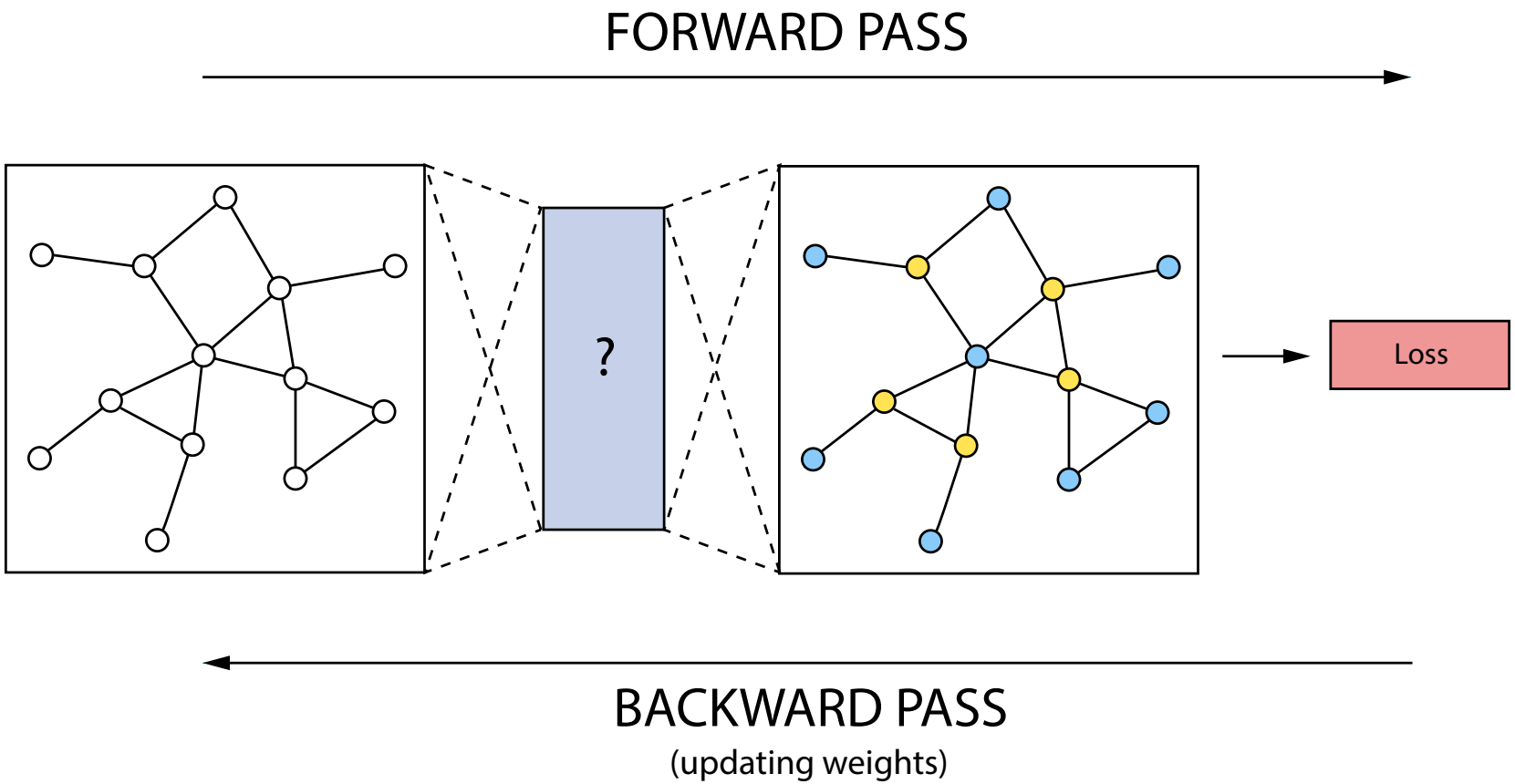
Learning over the whole graphs

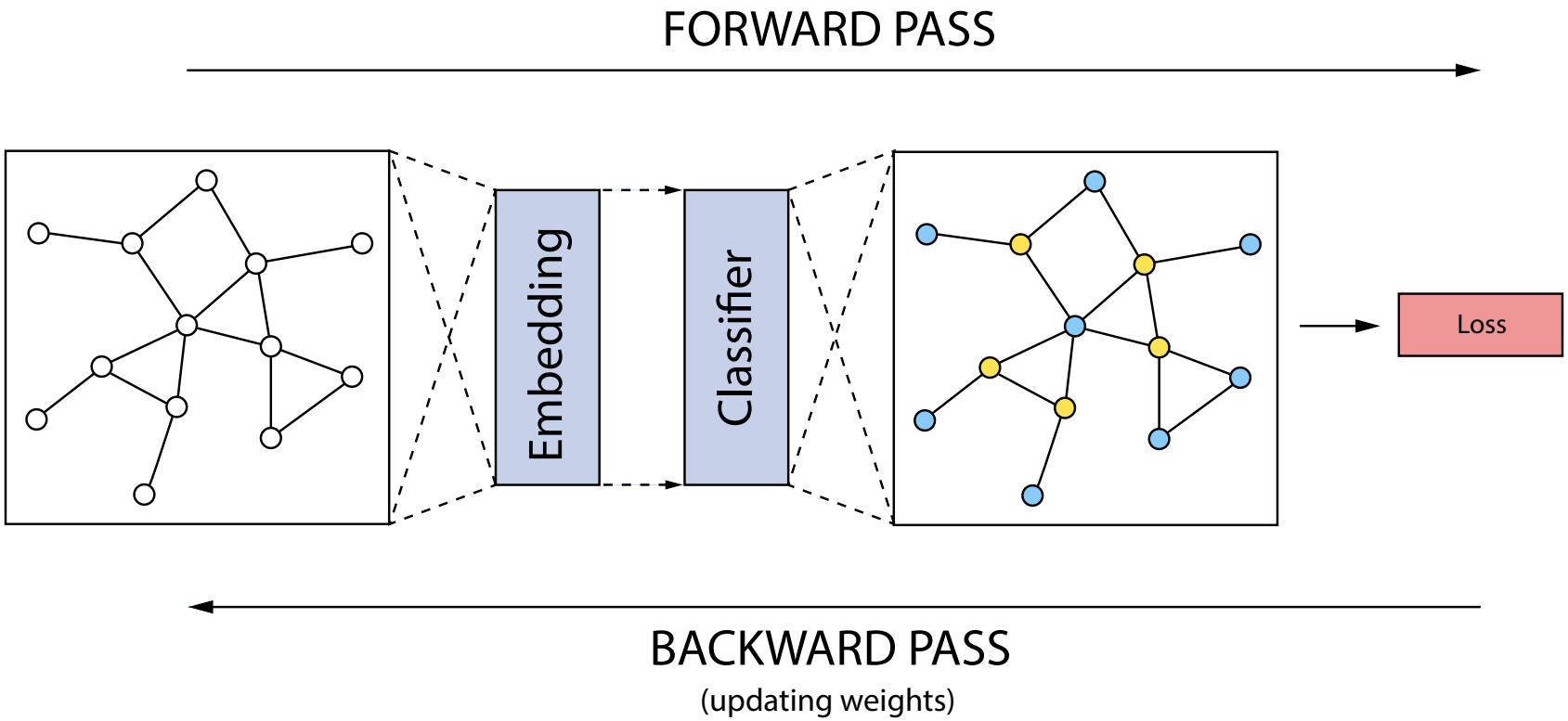
Adjacency Matrix

- Contains information whether pairs of nodes are adjacent or not.
- Handy for determining neighbours for each node and creating masks.



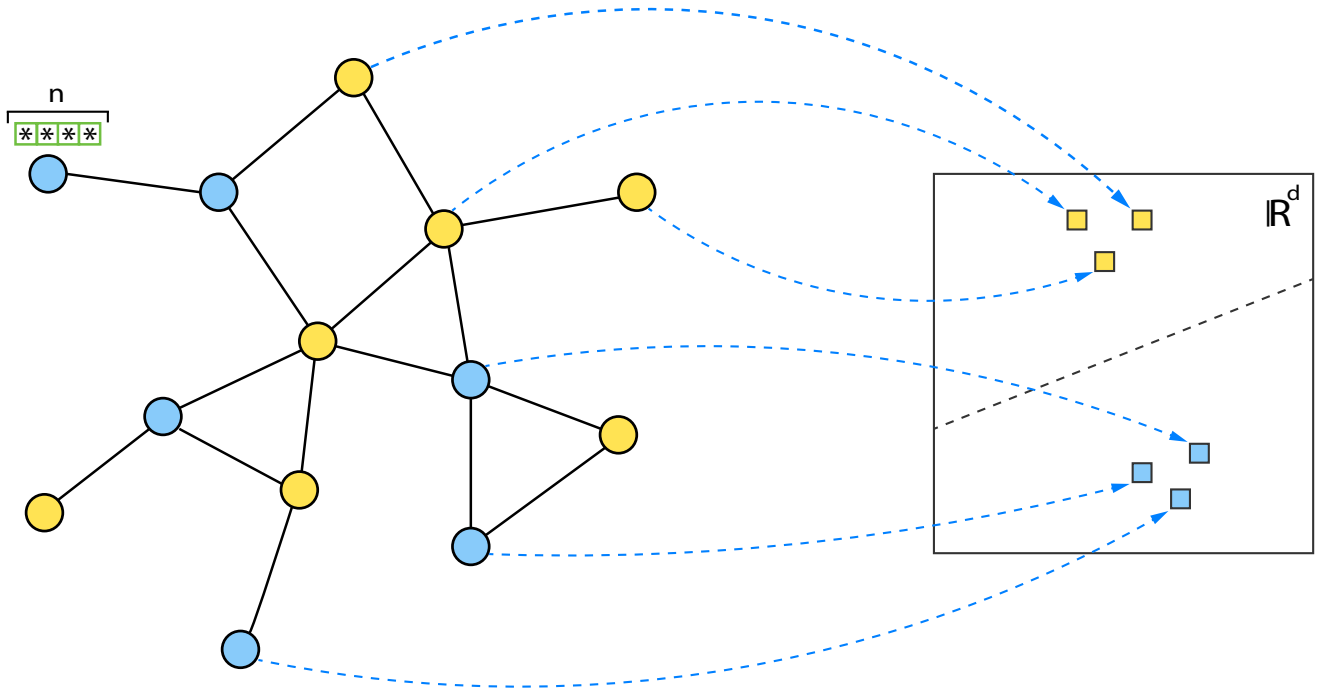
	0	1	2	3
0	0	1	0	1
1	1	0	1	0
2	0	1	0	1
3	1	0	1	0


FORWARD PASS
?
Loss
BACKWARD PASS
(updating weights)


FORWARD PASS
Embedding
Classifier
Loss
BACKWARD PASS
(updating weights)

- Embeddings - learnable vector representations of nodes.
- Embeddings extract structural features, taking into account information in some neighbourhood.
- Similarity of nodes u and v is defined by scalar product of their embeddings $\mathbf{z}_u$, $\mathbf{z}_v$:

$$\text{Similarity}(u, v) \sim \langle \mathbf{z}_u; \mathbf{z}_v \rangle$$



Model Preliminaries

Exponential distribution

- Assume each point Y_i in data $\mathcal{D}$ comes from exponential probability distribution p defined by parameters θ, ϕ and some functions a, c:

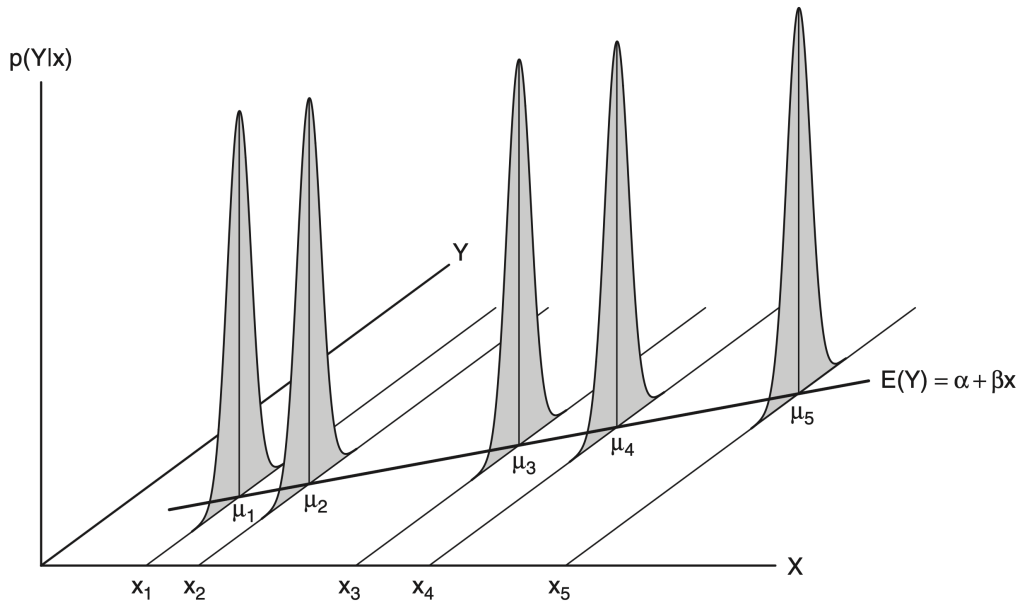
$$Y_i \sim p(y_i|\theta_i, \phi) = c(y_i, \phi) \exp\left(\frac{y_i\theta_i - a(\theta_i)}{\phi}\right).$$

- In other words:

$$P(Y_i = y_i) = p(y_i|\theta_i, \phi),$$
$$\mu_i \equiv \mathbb{E}(Y_i) = \dot{a}(\theta_i).$$

- Each point is described by factors $(X_1, X_2, \cdots, X_n) \in \mathbb{R}^n$ - feature vectors.
- Model expected value μ_i for each point via linear relationship with link function g:

$$\mu = g^{-1}(\eta) = g^{-1}(w_1 X_1 + w_2 X_2 + \cdots + w_n X_n).$$

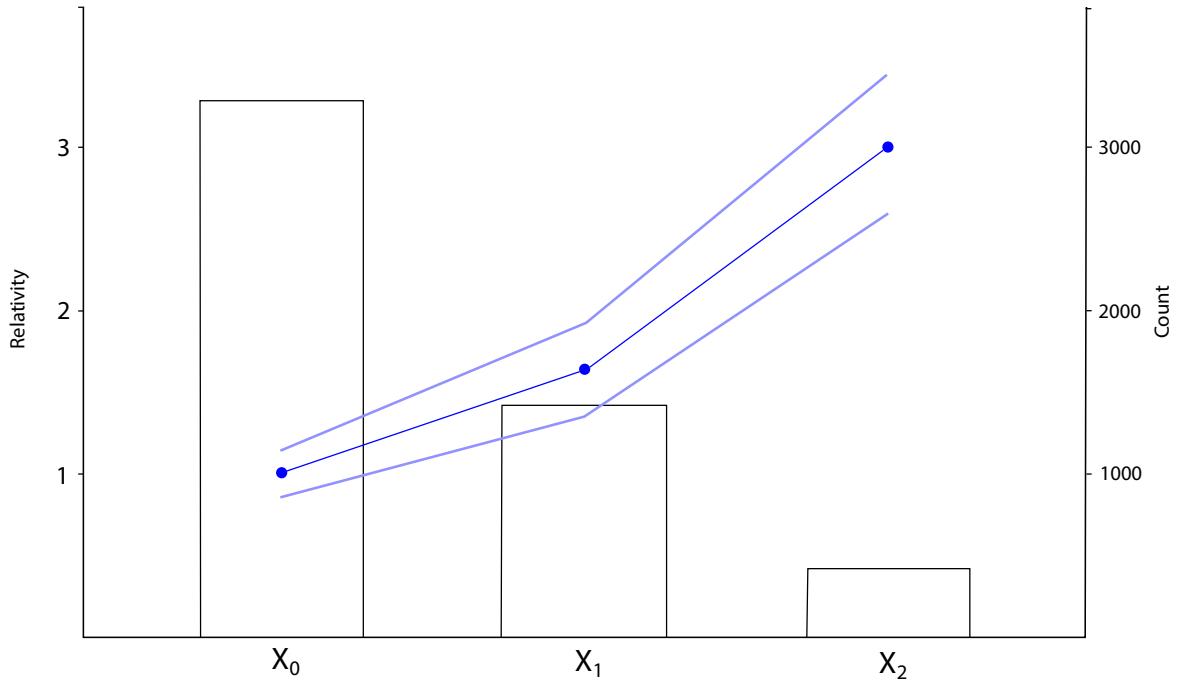


source

- The model can be (almost ☺) completely determined by parameters w_i.
- Estimate $\mathbf{w} = \{w_1, \ldots, w_n\}$ via maximizing (log –) likelihood function:

$$\ell(\mathbf{w}) = \log p(\mathcal{D}|\mathbf{w}) = \log \prod_i p(y_i|\theta_i, \phi) = \sum_i \log p(y_i|\mathbf{x}_i, \theta_i, \phi),$$

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \ell(\mathbf{w}).$$



- Bernoulli distribution

$$p(y_i) = \text{Ber}(y_i|\mu) = \mu^y (1-\mu)^{1-y} = \begin{cases} 1-\mu & \text{if y=0} \\ \mu & \text{if y=1} \end{cases}$$

- We want output to be in range from 0 to 1:

$$\mu \in [0, 1]$$

- Apply sigmoid (activation function) to satisfy this condition:

$$\mu = \sigma(\eta) = 1/\left(1 + e^{-\eta}\right),$$

$$\eta = w_1 X_1 + w_2 X_2 + \ldots + w_n X_n.$$

- Writing down log-likelihood for Bernoulli distribution leads to **LogLoss**:

$$\ell = \sum_i \log p(y_i|\mu_i) = -\sum_i \left(-y_i \log \mu_i - (1 - y_i) \log (1 - \mu_i)\right).$$

$$\text{LogLoss} = \sum_i \left(-y_i \log \mu_i - (1 - y_i) \log (1 - \mu_i)\right) \longrightarrow \text{minimize}$$

- LogLoss on 1 observation:

$$\text{LogLoss} = \begin{cases} -\log \mu_i & \text{if } y_i = 1, \\ -\log(1 - \mu_i) & \text{if } y_i = 0. \end{cases}$$

Linear Model: feature vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}, \quad \mathbf{x}^T = \begin{bmatrix} x_1 \; x_2 \; \dots \; x_n \end{bmatrix}.$$

$$\boxed{\eta = \mathbf{w}^T \mathbf{x} \in \mathbb{R}}$$

$$(1 \times n) \cdot (n \times 1) = (1 \times 1) \in \mathbb{R}$$

$$\eta = \begin{bmatrix} w_1 \; w_2 \; \dots \; w_n \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} = w_1 x_1 + w_2 x_2 + \dots + w_n x_n.$$

$$\eta = \mathbf{X}\mathbf{w}$$

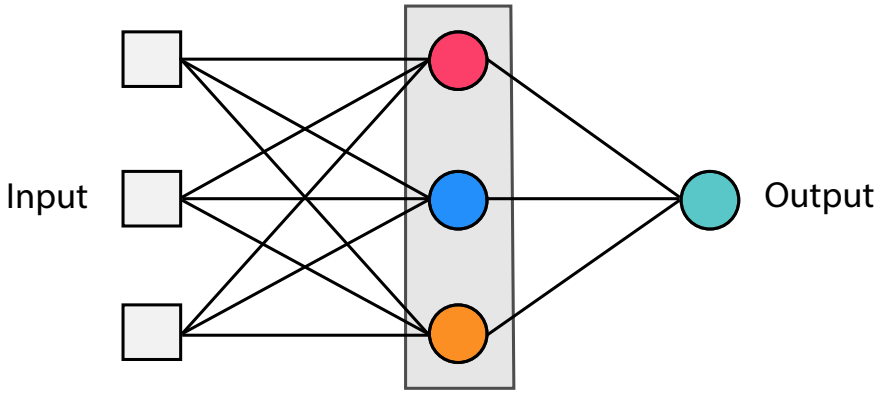
$$(N \times n) \cdot (n \times 1) = (N \times 1) \in \mathbb{R}^N$$

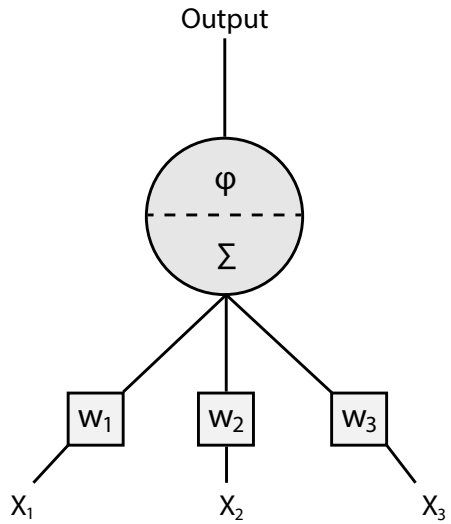
$$\begin{array}{c} \\ \text{Obs}_1 \\ \text{Obs}_2 \\ \dots \\ \text{Obs}_N \end{array} \overbrace{\begin{array}{c} \mathbf{X_1} \\ \\ \\ \\ \\ \end{array}}^{\text{Feature vectors}} \overbrace{\begin{array}{cc} \mathbf{X_2} & \dots & \mathbf{X_n} \\ & & \\ & & \\ & & \\ & & \end{array}}^{\text{Weights}} \begin{array}{c} \\ \\ \\ \\ \\ \end{array}$$

$$\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \dots & \dots & \dots & \dots \\ x_{N1} & x_{N2} & \dots & x_{Nn} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{w} \\ \\ \\ \\ \\ \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{bmatrix}$$

$$\begin{array}{c} \text{Obs}_1 \\ \text{Obs}_2 \\ \dots \\ \text{Obs}_N \end{array} \begin{bmatrix} w_1 x_{11} + w_2 x_{12} + \dots + w_n x_{1n} \\ w_1 x_{21} + w_2 x_{22} + \dots + w_n x_{2n} \\ \dots \\ w_1 x_{N1} + w_2 x_{N2} + \dots + w_n x_{Nn} \end{bmatrix}$$

Linear Layer (Projection)





$$\text{Output} : \phi(w_1 x_1 + w_2 x_2 + w_3 x_3) = \phi(\mathbf{x}^T \mathbf{w})$$

Linear Layer: feature vector

$$\overbrace{\begin{matrix} \mathbf{N_1} & \mathbf{N_2} & \dots & \mathbf{N_d} \end{matrix}}^{\text{Neurons}}$$

$$\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1d} \\ w_{21} & w_{22} & \dots & w_{2d} \\ \dots & \dots & \dots & \dots \\ w_{n1} & w_{n2} & \dots & w_{nd} \end{bmatrix}$$

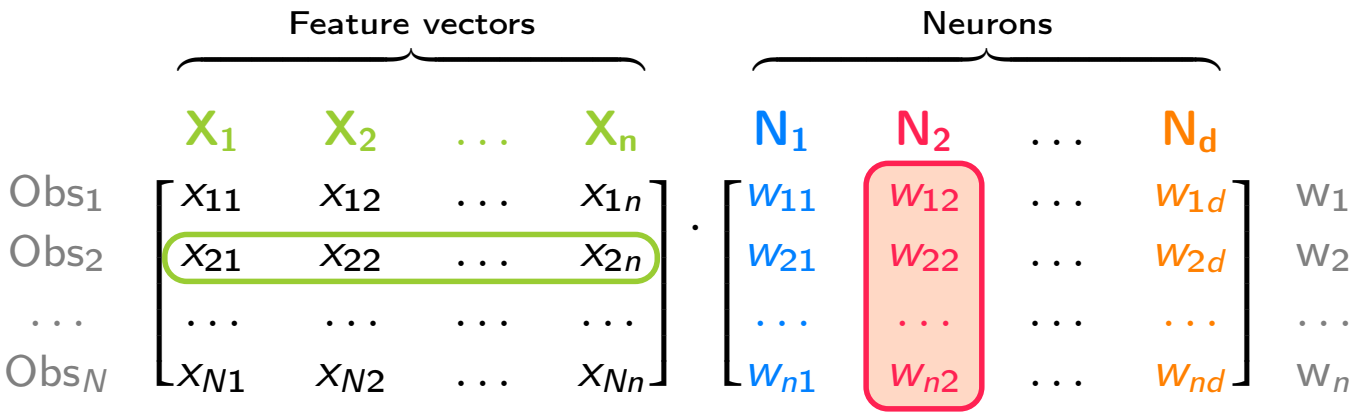
$$\eta = \mathbf{W}^T \mathbf{x} \in \mathbb{R}^d$$

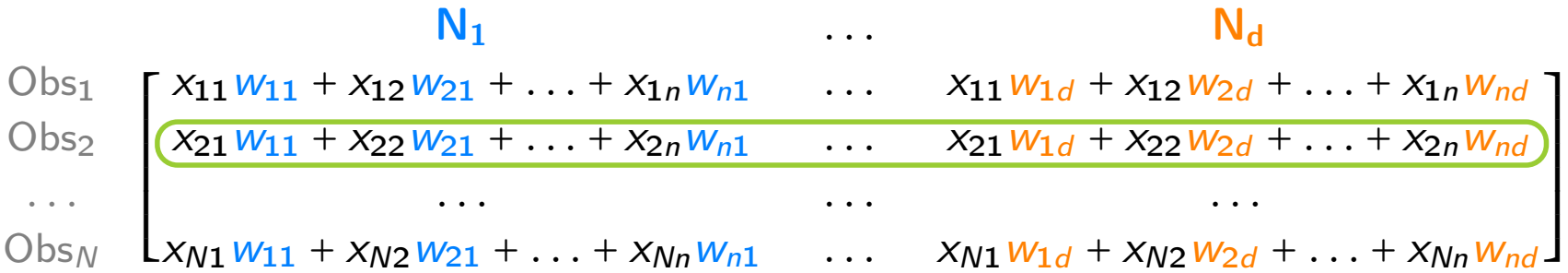
$$(d \times n) \cdot (n \times 1) = (d \times 1) \in \mathbb{R}^d$$

Linear Layer: Batch

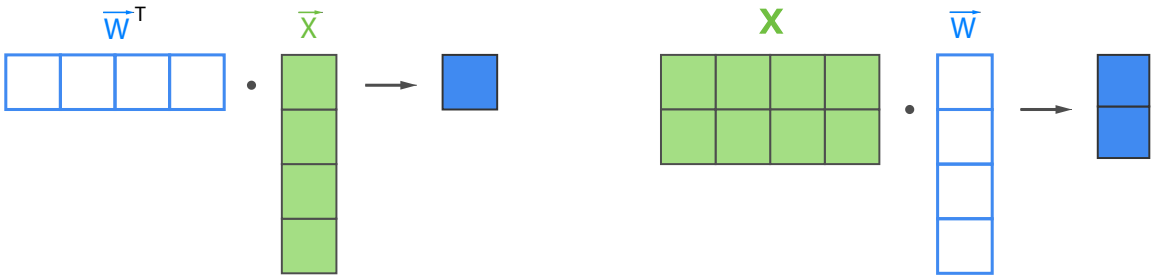
$$\mathcal{O} = \mathbf{XW}$$

$$(N \times n) \cdot (n \times d) = (N \times d) \in \mathbb{R}^{N \times d}$$

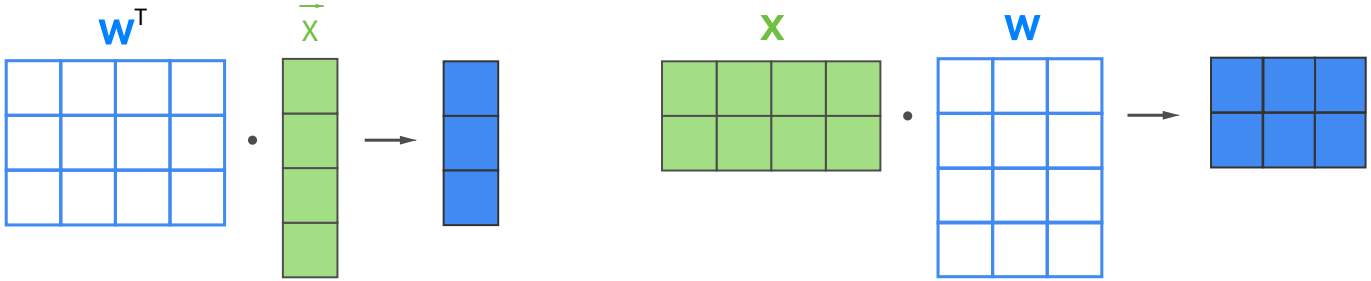




LINEAR MODEL



LINEAR LAYER



PyTorch: Linear Layer

In 1
```
import torch
import torch.nn as nn
```

In 2
```
linear = nn.Linear(4, 2, bias=False)  # in features: 4, out_features: 2
```

In 3
```
input = torch.randn(3, 4)
input
```

Out 3
```
tensor([[-0.2782,  0.0798, -0.3552, -0.2618],
        [ 0.6665,  1.9803,  0.2063, -1.0990],
        [-0.2645, -0.0047, -0.5869, -0.3289]])
```

In 4
```
# [3,4] * [2,4]^T = [3, 2]
output = linear(input)
output
```

Out 4
```
tensor([[ 0.1267, -0.0859],
        [ 0.8247,  0.8660],
        [ 0.0721, -0.0997]], grad_fn=<MmBackward>)
```

In 5
```
linear.weight
```

Out 5
```
Parameter containing:
tensor([[-0.3313,  0.4970,  0.0487, -0.0466],
        [ 0.4462,  0.1334,  0.1113, -0.2562]], requires_grad=True)
```

PyTorch: Model

In 1
```
import torch
import torch.nn as nn
```

In 2
```
class Model(nn.Module):
    """ Simple model: 2 linear layers and ReLu activation. """
    def __init__(self,
                 in_dim,
                 hid_dim):

        super().__init__()

        self.in_dim = in_dim
        self.linear1 = nn.Linear(in_dim, hid_dim, bias=False)
        self.linear2 = nn.Linear(hid_dim, 1, bias=False)
        self.relu = nn.ReLU()

    def forward(self, input):
        out = self.linear1(input)  # [batch_size, hid_dim]
        out = self.relu(out)  # activation function
        out = self.linear2(out)  # [batch_size, 1]
        return out
```

PyTorch: Model

In 3
```
model = Model(in_dim=4,
              hid_dim=2)
```

In 4
```
model.linear1.weight
```

Out 4
```
Parameter containing:
tensor([[-0.1320,  0.0753, -0.3863,  0.4888],
        [-0.2059, -0.2908,  0.4562, -0.0320]], requires_grad=True)
```

In 5
```
model.linear2.weight
```

Out 5
```
Parameter containing:
tensor([[0.3291, 0.3062]], requires_grad=True)
```

In 6
```
input = torch.randn((2,4), dtype=torch.float32)
input
```

Out 6
```
tensor([[-0.6946, -1.0389, -0.3978,  1.0273],
        [-0.0191,  0.8283,  0.3426,  2.6409]])
```

In 7
```
output = model(input)
output
```

Out 7
```
tensor([[0.2909],
        [0.4026]], grad_fn=<MmBackward>)
```

PyTorch: Parameters

- *Parameters* are *Tensor* sublasses.
- They're automatically added to list of *Module* parameters when assigned as attributes in constructor.

In 2
```
class Model(nn.Module):

    def __init__(self):
        super().__init__()
        self.W = nn.Parameter(torch.Tensor(2,2))
        self.T = torch.zeros(2, 2, requires_grad=True)
```

In 3
```
model = Model()
for p in model.named_parameters():
    print(p)
```

```
('W', Parameter containing:
tensor([[0., 0.],
        [0., 0.]], requires_grad=True))
```

In 4
```
# initialize weights
for param in model.parameters():
    stdv = 1. / math.sqrt(param.size(-1))
    param.data.uniform_(-stdv, stdv)
```

In 5
```
import torch.optim as optim

# optimizer will update only registered Parameters of the model
optimizer = optim.Adam(model.parameters())
```

Categorical Factors

Data

Color
Blue
Red
Grey
Grey
Blue

Label Encoding

Color
1
2
3
3
1

OHE vectors

Blue	Red	Grey
1	0	0
0	1	0
0	0	1
0	0	1
1	0	0

$$\mathbf{x} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \mathbf{W} = \begin{bmatrix} 5 & 2 & 1 & 0 \\ 3 & 6 & 9 & 5 \\ 2 & 1 & 0 & 7 \end{bmatrix}.$$

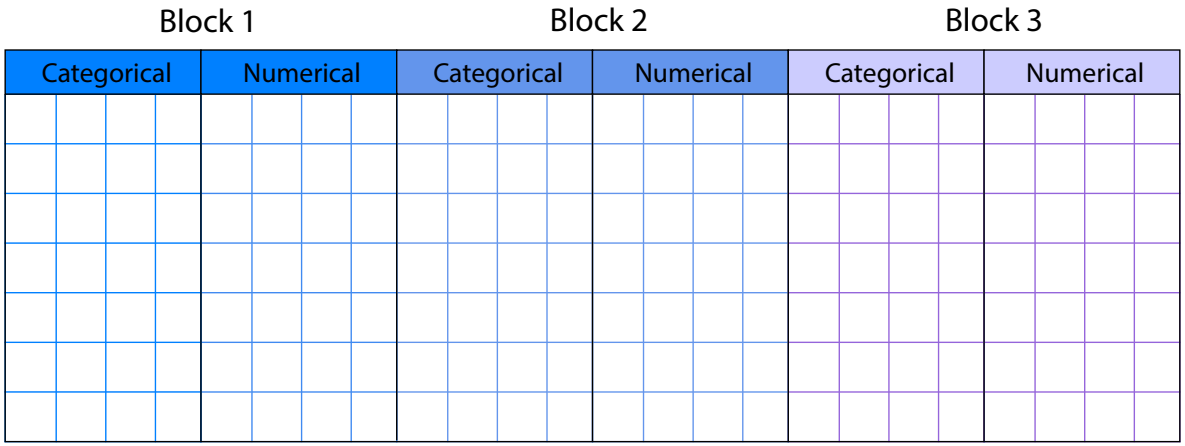
$$\mathbf{W}^T\mathbf{x} = ?$$

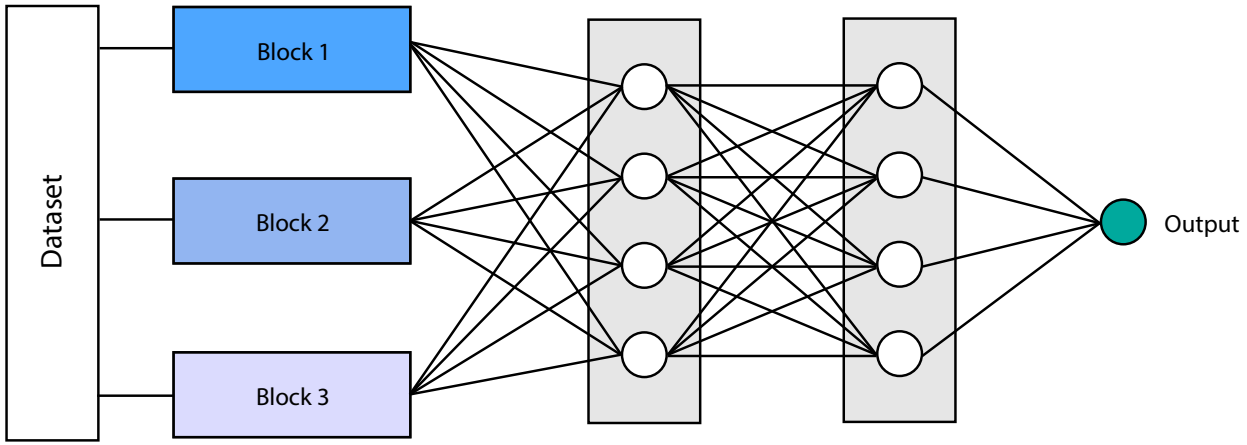
$$\mathbf{W}^T\mathbf{x} = \begin{bmatrix} 5 & 3 & 2 \\ 2 & 6 & 1 \\ 1 & 9 & 0 \\ 0 & 5 & 7 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 9 \\ 5 \end{bmatrix}.$$

$$\mathbf{W} = \begin{bmatrix} 5 & 2 & 1 & 0 \\ 3 & 6 & 9 & 5 \\ 2 & 1 & 0 & 7 \end{bmatrix}.$$

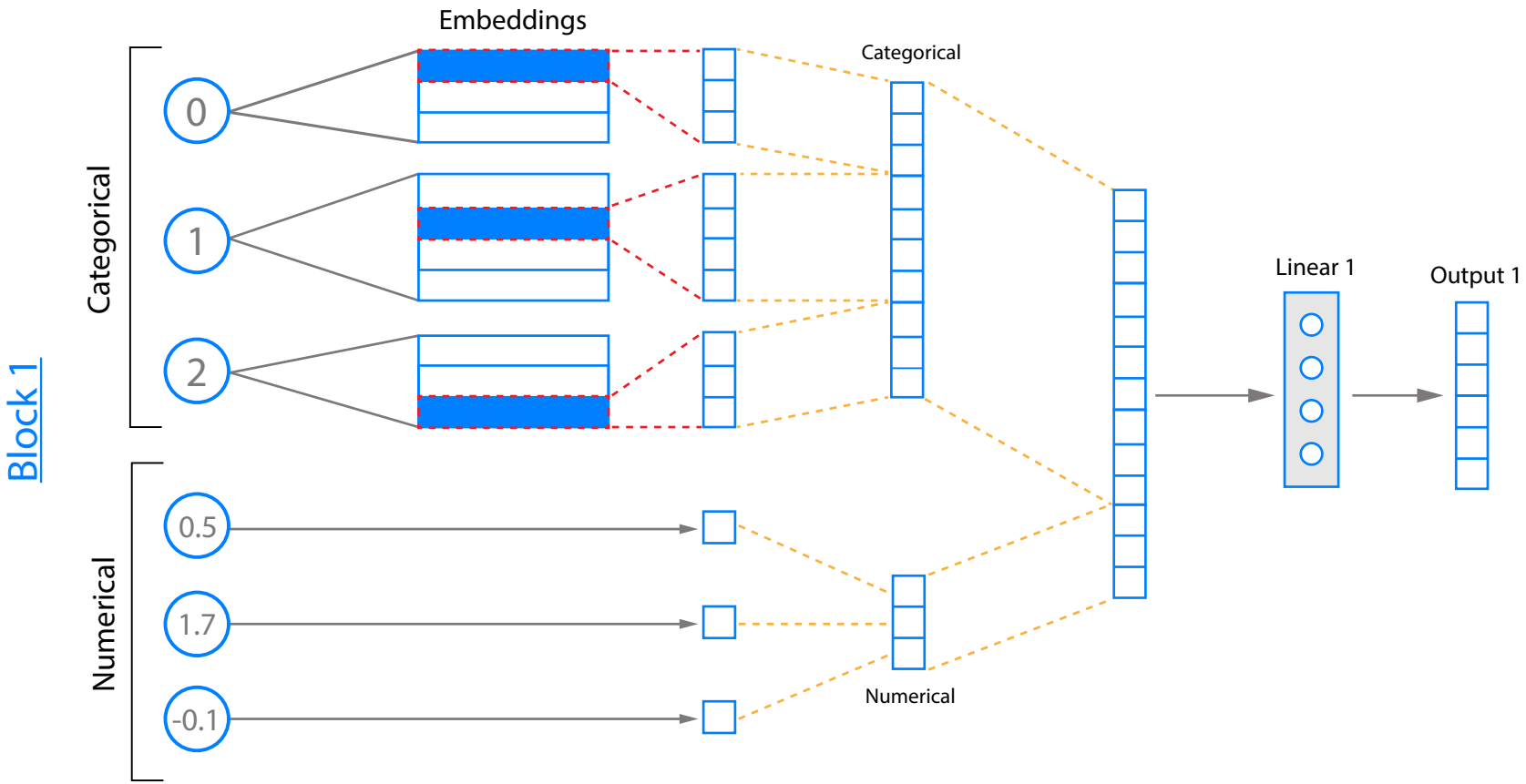
Good news: we don't need to OHE categorical variables.
Label-encode categories + initialize embedding matrix $\longrightarrow$ learnable lookup table.

Categorical Factors





Categorical Factors



Concatenate outputs from all blocks and pass to the remainder layers of NN.

Entity Embeddings of Categorical Variables
arXiv:1604.06737 [cs.LG]

Embedding layer in PyTorch:

In 1
```
import torch
import torch.nn as nn
```

In 2
```
# an Embedding layer: 4 vectors of size 3
embedding = nn.Embedding(4, 3)
embedding.weight
```

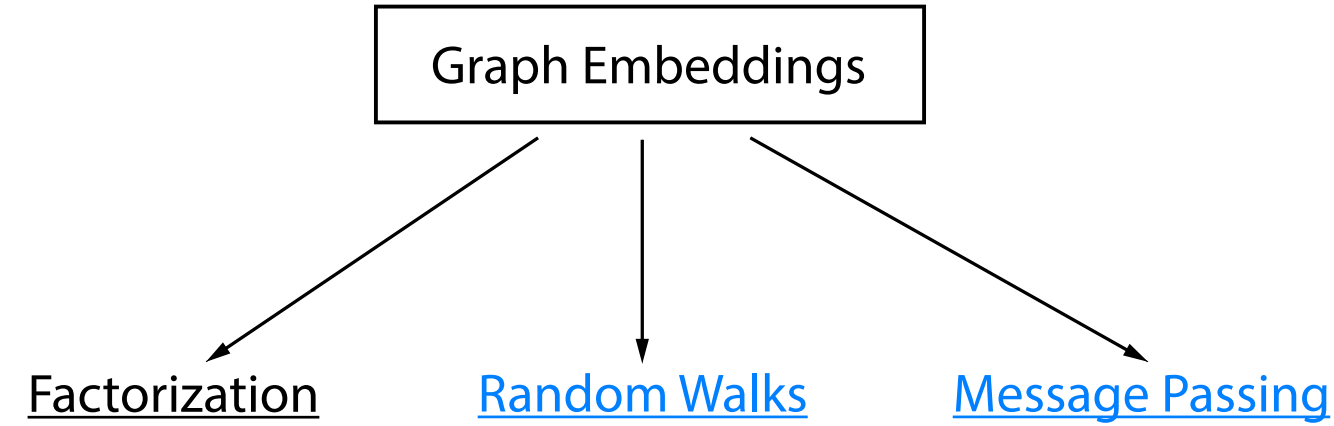
Out 2
```
Parameter containing:
tensor([[ 1.0122,  2.0579, -0.0501],
        [ 3.3124, -0.9464,  1.3357],
        [-0.2790,  0.2287, -2.1280],
        [ 0.7080, -1.1197, -0.1263]], requires_grad=True)
```

In 3
```
# a batch of 2 samples of 3 indices each
input = torch.LongTensor([[0,1,2],[3,2,1]])
embedding(input)
```

Out 3
```
tensor([[[ 1.0122,  2.0579, -0.0501],
         [ 3.3124, -0.9464,  1.3357],
         [-0.2790,  0.2287, -2.1280]],

        [[ 0.7080, -1.1197, -0.1263],
         [-0.2790,  0.2287, -2.1280],
         [ 3.3124, -0.9464,  1.3357]]], grad_fn=<EmbeddingBackward>)
```

Graph embeddings



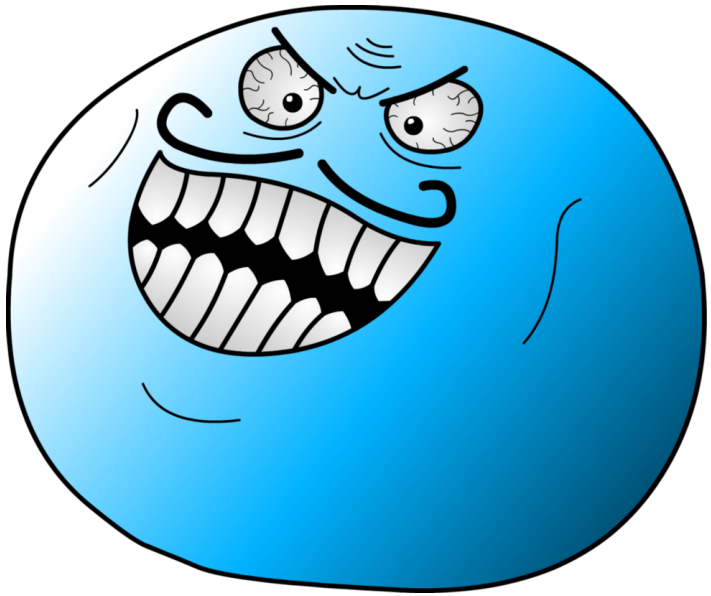
Representation Learning on Graphs: Methods and Applications

arXiv:1709.05584 [cs.SI]

Graph Representation Learning Book [Hamilton]

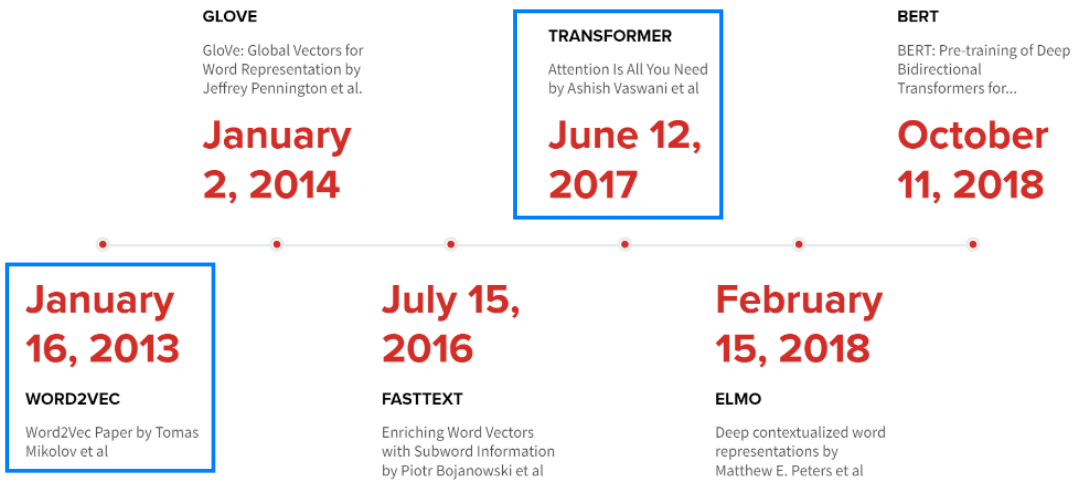
https://www.cs.mcgill.ca/ wlh/grl_book/

Embeddings



- Many approaches in modern Deep Learning are closely related to ideas from Natural Language Processing (**NLP**).

- Timeline of some major NLP methods:



source

- Check the Stanford CS224N course.

Word context

- Distributional Hypothesis: words that occur in the same contexts tend to have similar meanings [Harris, 1954]
- "A word is characterized by the company it keeps" [Firth, 1957]

- Can you guess the meaning of the word? source
  1. A bottle of tezgüino is on the table.
  2. Everyone likes tezgüino.
  3. Tezgüino makes you drunk.
  4. We make tezgüino for fun.

word2vec

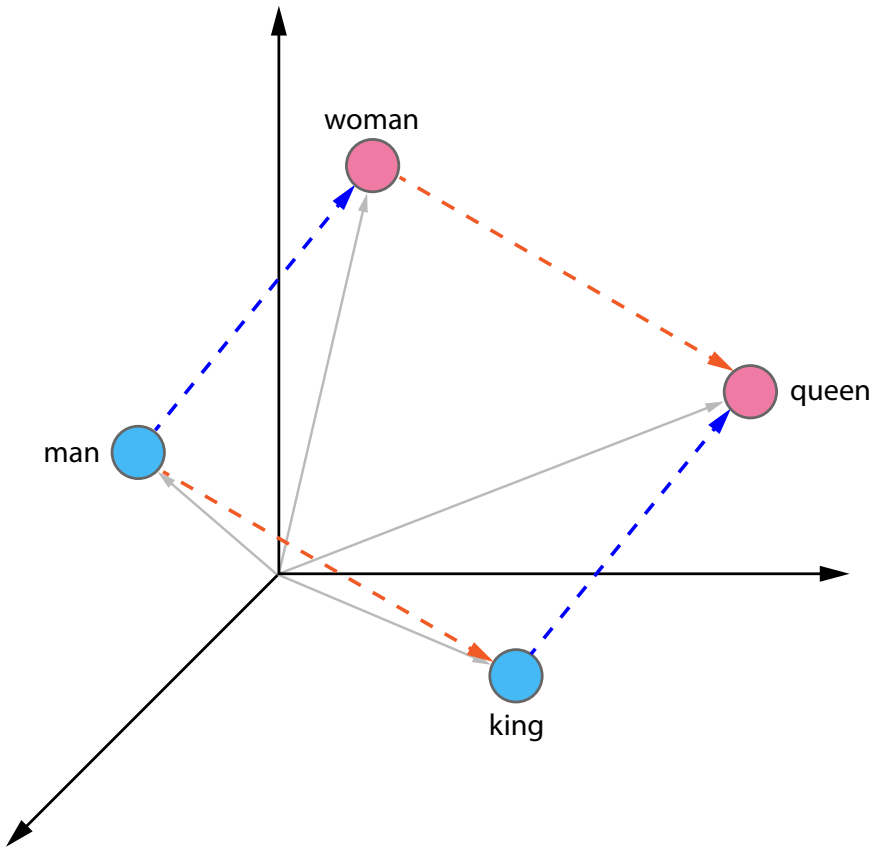
Efficient Estimation of Word Representations in Vector Space

arXiv:1301.3781 [cs.CL]

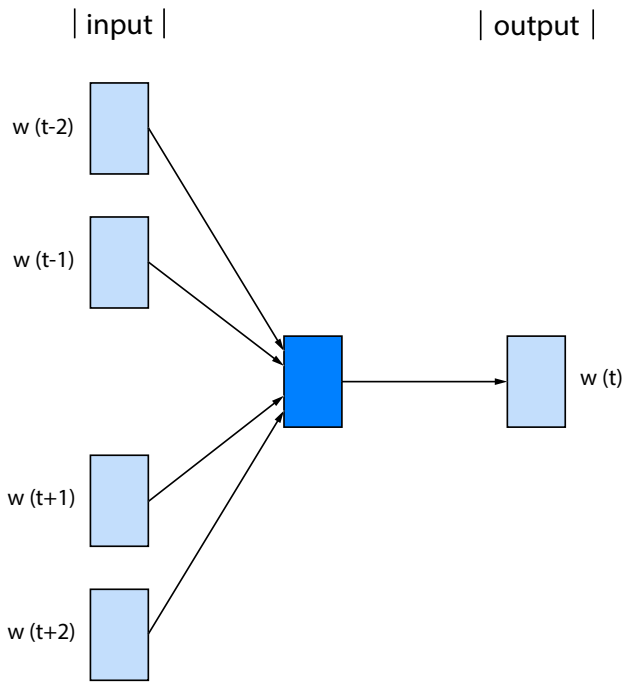
word2vec Parameter Learning Explained

arXiv:1411.2738 [cs.CL]

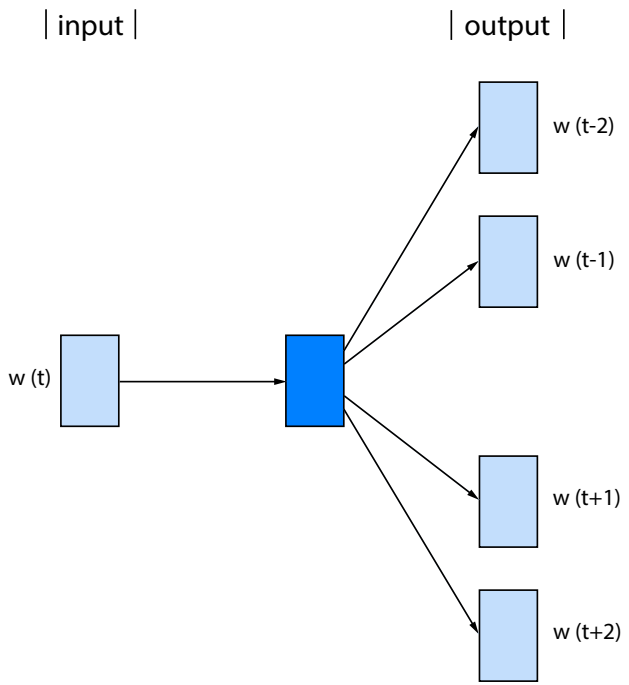
$$\text{vector}[\text{King}] - \text{vector}[\text{Man}] + \text{vector}[\text{Woman}] = \text{vector}[\text{Queen}]$$



CBoW, Skip-gram

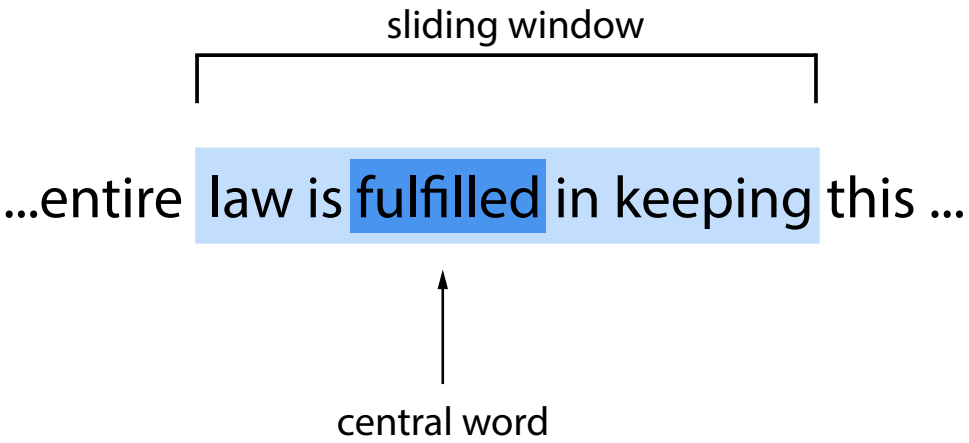


CBoW



Skip-gram

For the entire law is fulfilled in keeping this one command:
"Love your neighbor as yourself".



fulfilled	law
fulfilled	is
fulfilled	in
fulfilled	keeping

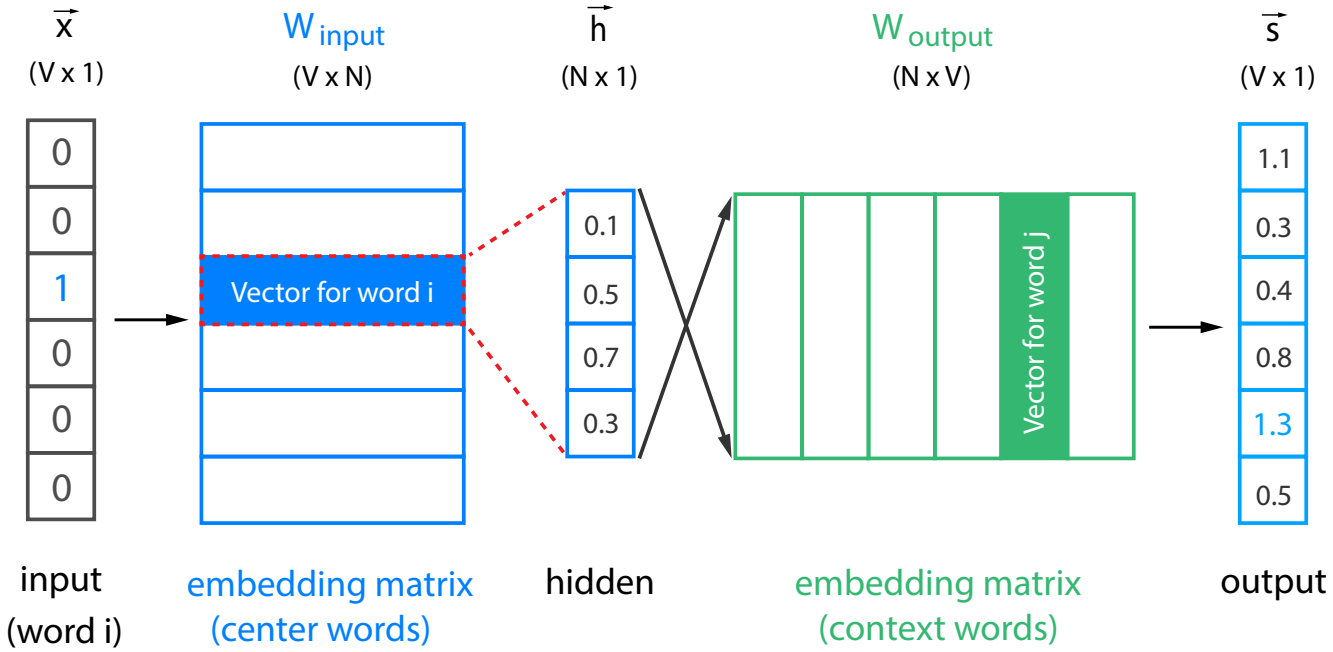
Keep on sliding

...entire law is fulfilled in keeping this ...

... law is fulfilled in keeping this one ...

fulfilled	law
fulfilled	is
fulfilled	in
fulfilled	keeping
in	is
in	fulfilled
in	keeping
in	this

Skip-gram



$$\mathbf{h} = \mathbf{W}_{\text{input}}{}^{T}\mathbf{x} = \mathbf{v}_{w_I}^{T}.$$

$$\mathbf{s} = \mathbf{W}_{\text{output}}^{T}\mathbf{h}.$$

$$\boxed{s_j = \langle \mathbf{v}'_{w_j}; \mathbf{v}_{w_I} \rangle}$$

Similarity of central and context words embeddings

Softmax: Definition

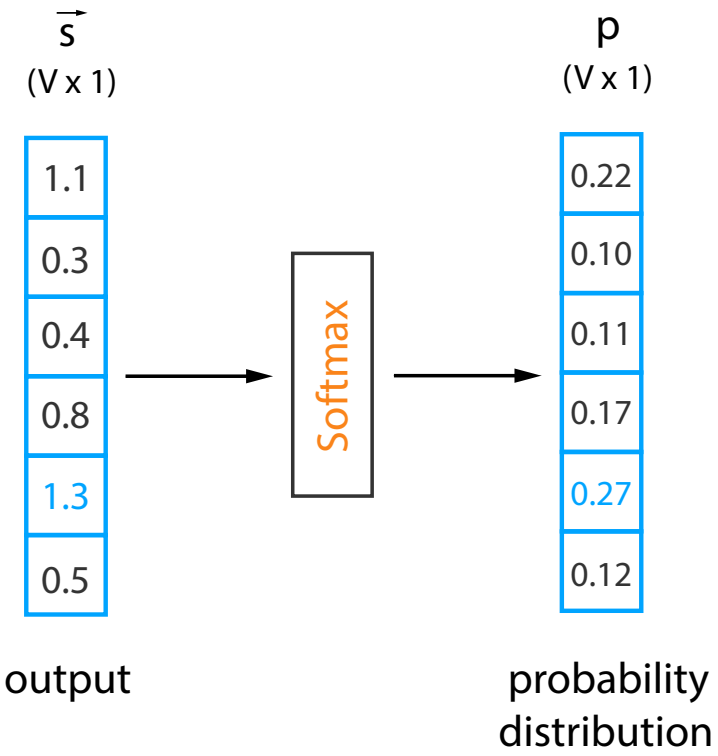
Softmax function / Boltzmann distribution

$$\mathbf{z} = (z_1, \cdots, z_K) \in \mathbb{R}^K.$$

$$\boxed{\sigma(\mathbf{z})_i = \frac{e^{\beta z_i}}{\sum_{j=1}^{K} e^{\beta z_j}}, \ \beta = 1/T.}$$

- $\sigma(\mathbf{z}_i) : \mathbb{R}^K \to [0,1]^K$.
- $\sum_{i=1}^{K} \sigma(\mathbf{z})_i = 1$.
- Normalizes output to **probability distribution** over predicted classes.
- T - temperature.
  At low temperatures the distribution puts most of its probability mass in the most probable state.
  At high temperatures it spreads the mass more uniformly.

Softmax

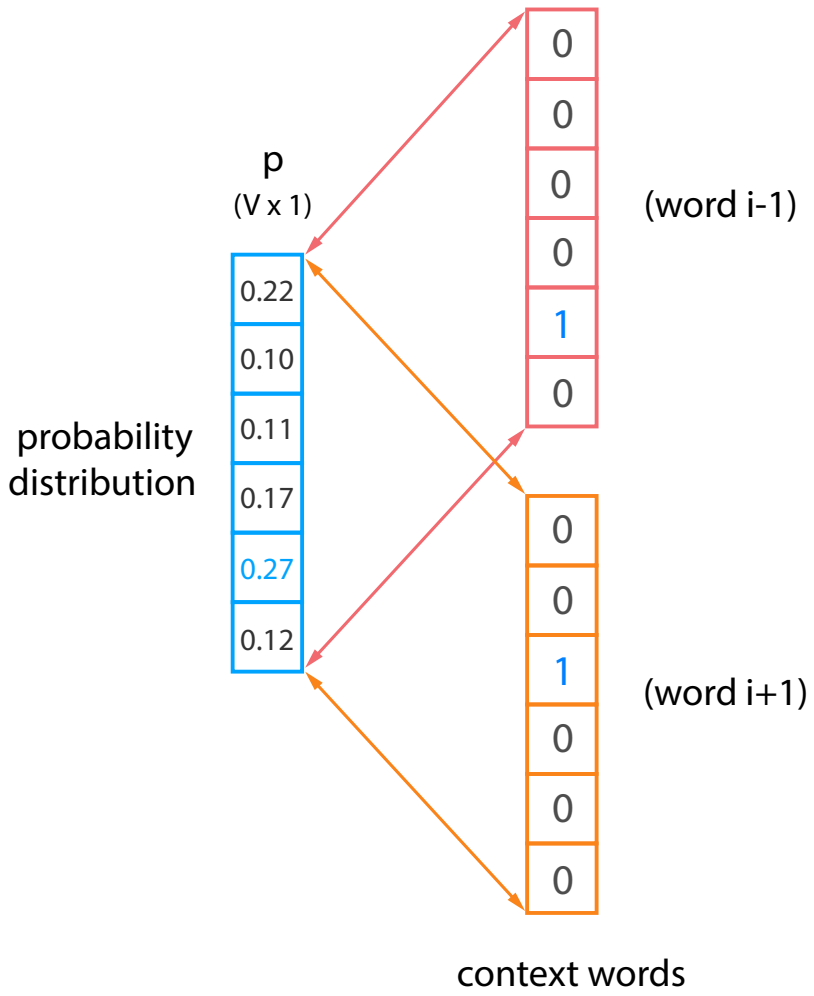


- Probability of observing the actual output word w_O given the input word w_I:

$$p(w_O|w_I) = \frac{\exp\left(\langle \mathbf{v}'_{w_O}; \mathbf{v}_{w_I} \rangle\right)}{\sum_{j=1}^{V} \exp\left(\langle \mathbf{v}'_{w_j}; \mathbf{v}_{w_I} \rangle\right)}.$$

- V can be extremely large. Calculating denominator is computationally complex.

Skip-gram Likelihood



Likelihood

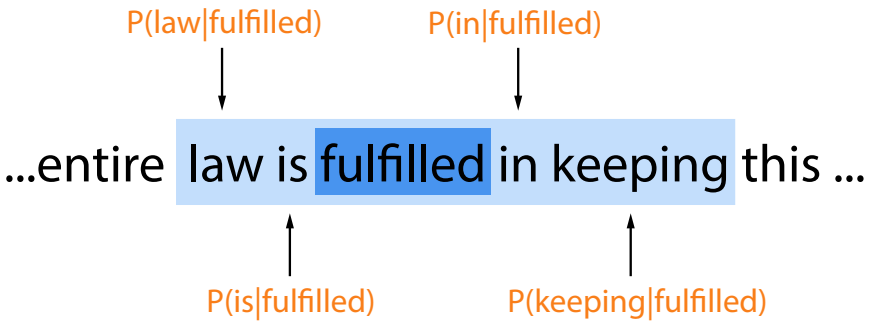
- Compute probabilities of a context words for each central word in corpus:

$$\mathcal{L}_\theta = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P(w_{t+j} \mid w_t).$$

$t = 1, \ldots, T$ – positions of words,

m – sliding window size.

Skip-gram Loss



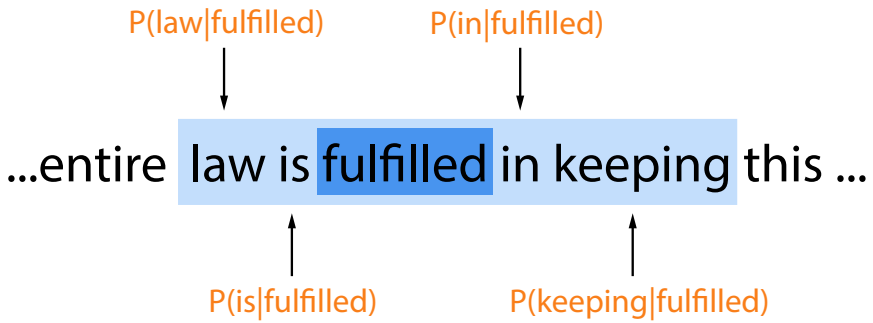
- Loss function $J(\theta)$ - average negative log-likelihood:

$$J(\theta) \triangleq -\frac{1}{T} \log \mathcal{L}_\theta = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log P(w_{t+j} \mid w_t).$$

- w_t = fulfilled, w_{t+j} = law, $j = -2$:

$$-\log P(\text{law} \mid \text{fulfilled}) = -\log \frac{\exp\left(\langle \mathbf{v}'_{\text{law}}; \mathbf{v}_{\text{fulfilled}} \rangle\right)}{\sum_{j=1}^{V} \exp\left(\langle \mathbf{v}'_j; \mathbf{v}_{\text{fulfilled}} \rangle\right)}.$$

Skip-gram Loss



- Loss term:

$$J_{t,j}(\theta) = -\langle \mathbf{v}'_{\text{law}}; \mathbf{v}_{\text{fulfilled}} \rangle + \log \sum_{j=1}^{V} \exp\left(\langle \mathbf{v}'_{j}; \mathbf{v}_{\text{fullfilled}} \rangle\right).$$

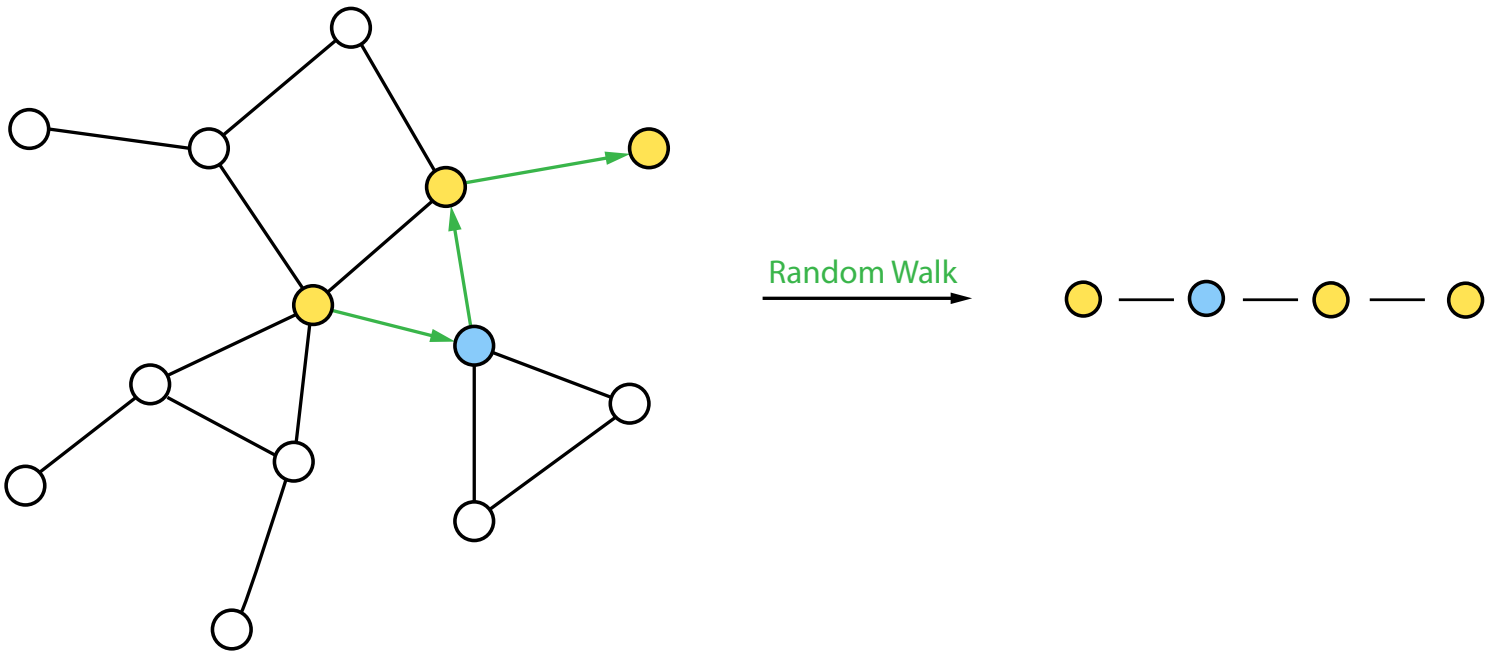
- How to minimize Loss:
  1. increase similarity between central and context vector: $\langle \mathbf{v}'_{\text{law}}; \mathbf{v}_{\text{fulfilled}} \rangle$.
  2. decrease similarity between central and non-context words: $\log \sum_j \exp(\ldots)$.

node2vec

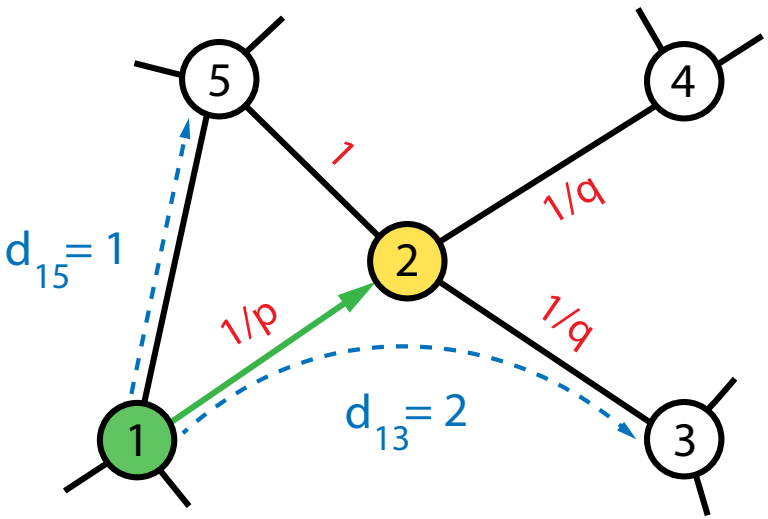
node2vec: Scalable Feature Learning for Networks

arXiv:1607.00653 [cs.SI]

- Unsupervised / self-supervised embeddings - task independent.
  1. Not using node labels.
  2. Not using node features.
- Source node $u \longrightarrow$ neighbourhood $N_S(u)$:



- Biased random walk of fixed length: interpolate between local (BFS) and global (DFS) views of neighbourhoods.
- Parameters:
  1. p - return parameter
  2. q - in-out parameter

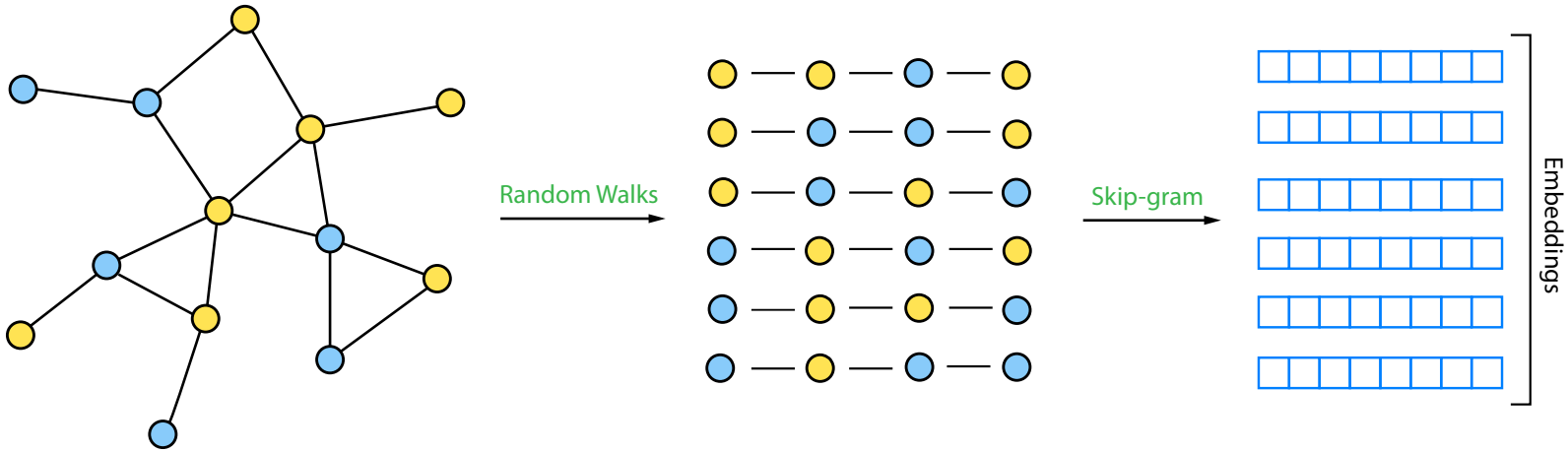


Transitioned 1 → 2, evaluating next step.

Unnormalized transition probabilities:

$$\alpha(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

d_{tx} - shortest distance between t and x.



Maximize log-probability of observing neighbourhood $N_S(u)$ for node u:

$$J(\theta) \sim -\log \mathcal{L}_\theta = -\sum_{u \in V} \log P\left(N_S(u) \mid \mathbf{z}_u\right) = -\sum_{u \in V} \sum_{v \in N_S(u)} \log P(v \mid \mathbf{z}_u).$$

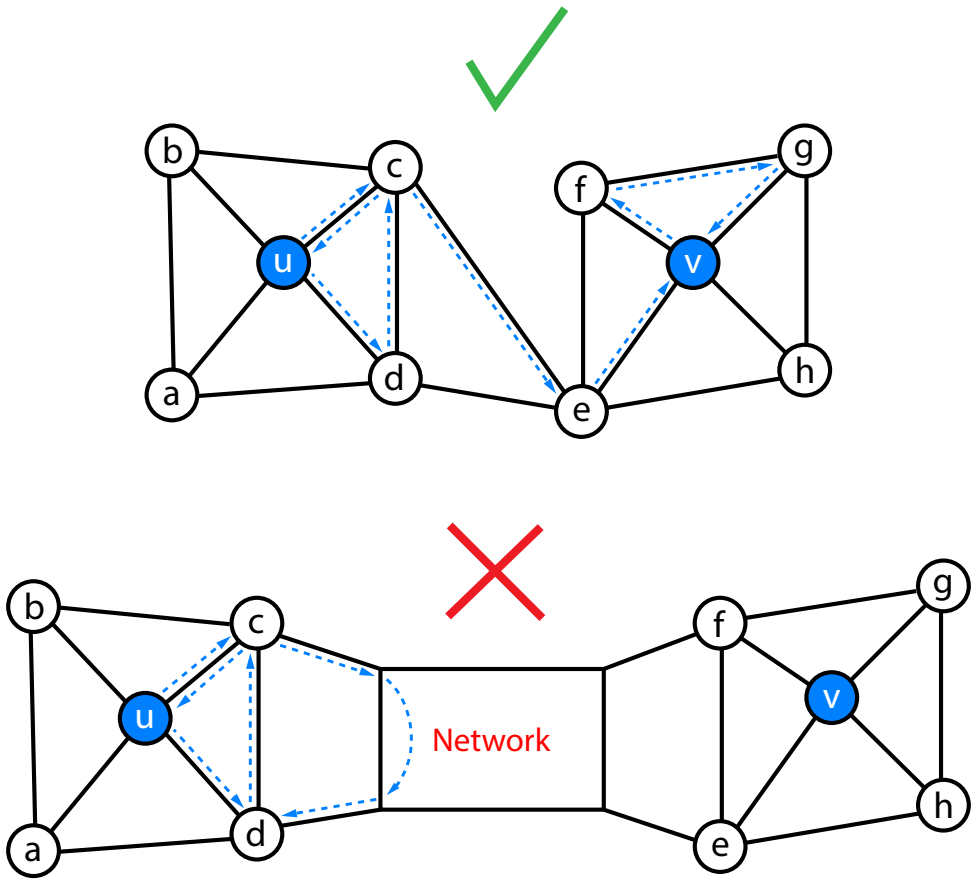
$$P(N_S(u)|\mathbf{z}_u) = \prod_{n_i \in N_S(u)} P(n_i|\mathbf{z}_u)$$

Probability parametrized using softmax:

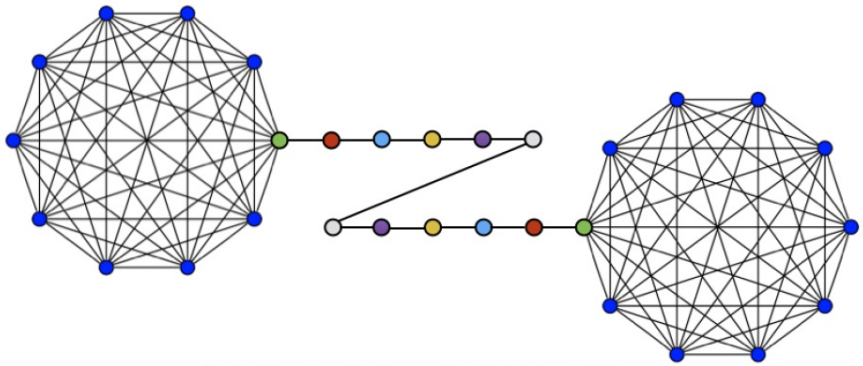
$$P(v \mid \mathbf{z}_u) = \frac{\exp(\langle \mathbf{z}_v; \mathbf{z}_u \rangle)}{\sum_{n \in V} \exp(\langle \mathbf{z}_n; \mathbf{z}_u \rangle)}$$

node2vec drawbacks

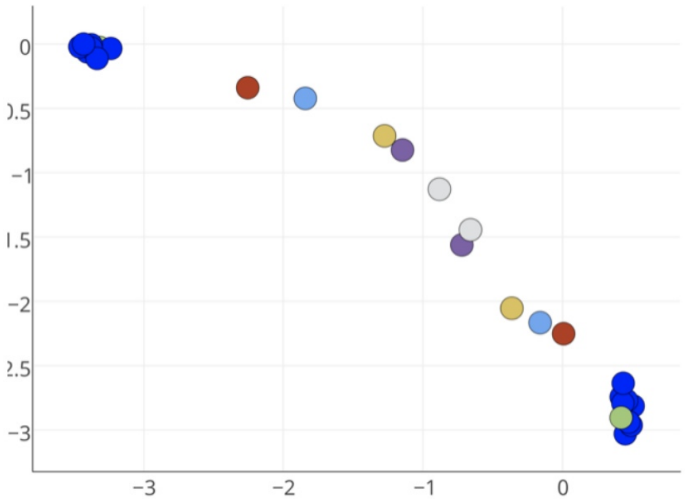
- node2vec tend to fail in structural equivalence tasks.
- Nodes that are far in the network tend to be separated in latent representation.



node2vec drawbacks



(a) Barbell Graph B(10, 10)



(d) *node2vec*

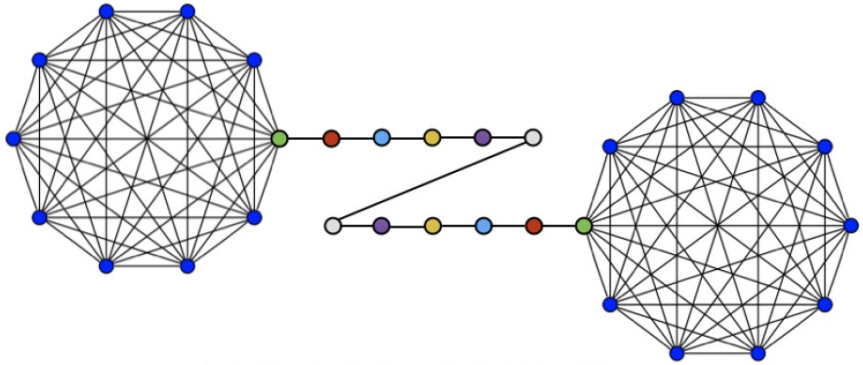
struc2vec

struc2vec: Learning Node Representations from Structural Identity

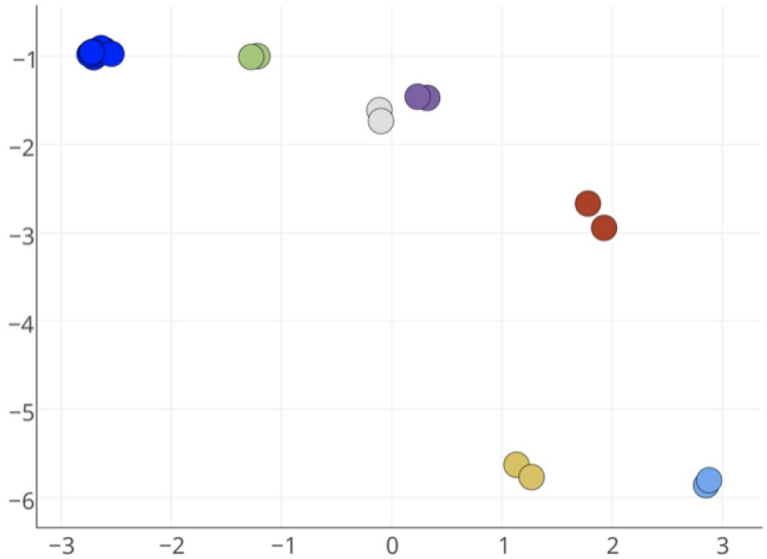
arXiv:1704.03165 [cs.SI]

https://github.com/leoribeiro/struc2vec

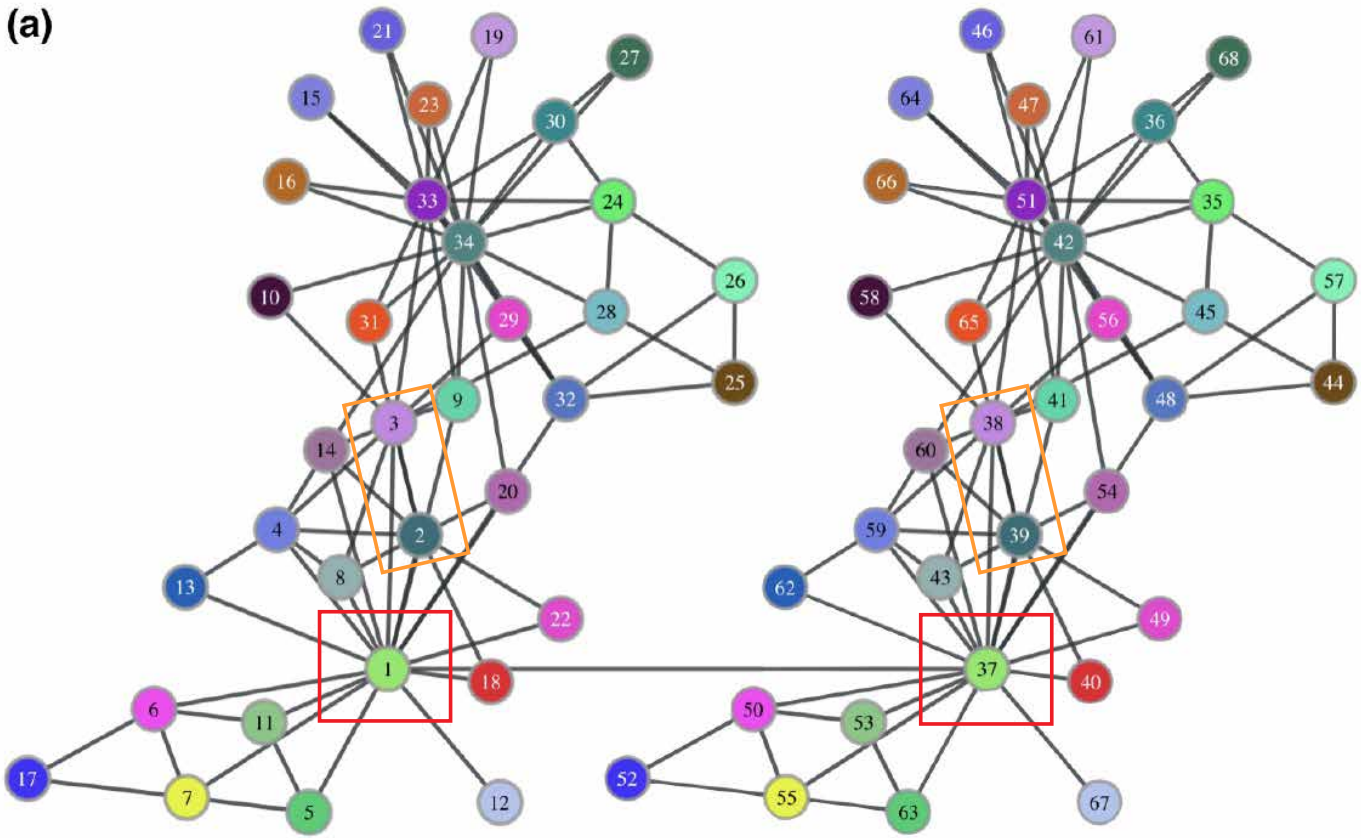
- Latent representations for the structural identity of nodes
- Biased random walks on multilayer graph

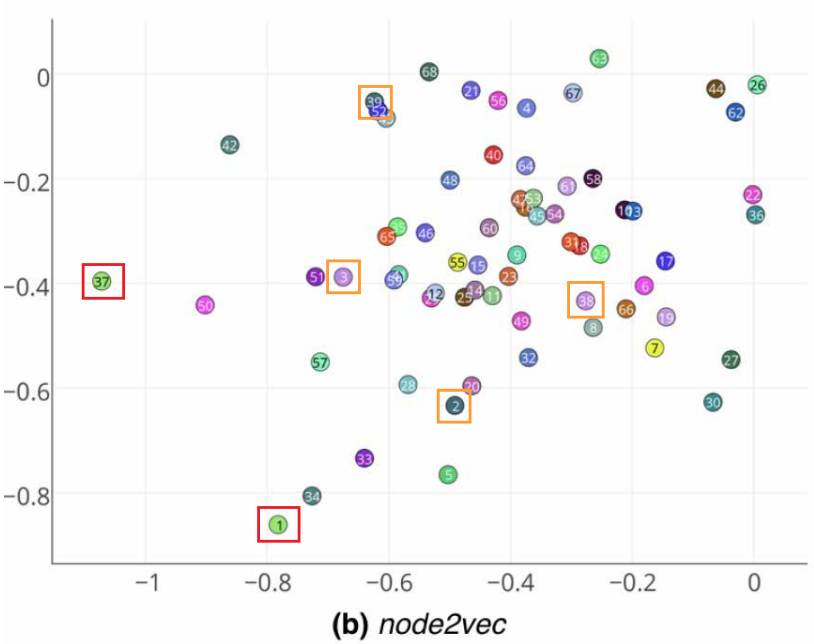


(a) Barbell Graph B(10, 10)

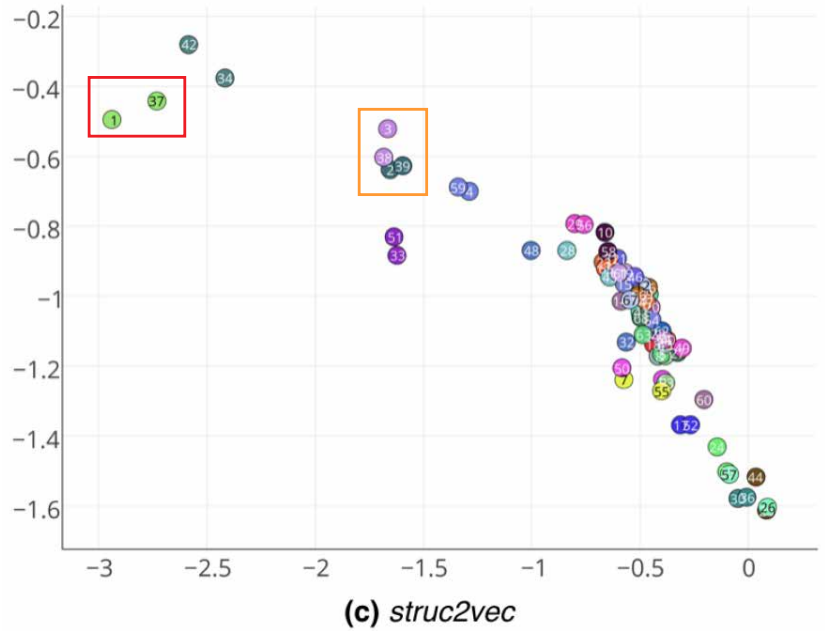


(e) *struc2vec*


(a)
21
19
27
15
23
30
16
33
24
34
10
26
31
29
28
25
9
32
3
14
20
4
2
8
13
22
1
18
6
11
17
7
5
12
46
61
68
64
47
36
66
51
35
42
58
57
65
56
45
44
41
48
38
60
54
59
39
43
62
49
37
40
50
53
52
55
63
67



(b) *node2vec*



(c) *struc2vec*

Message Passing

Neural Message Passing for Quantum Chemistry

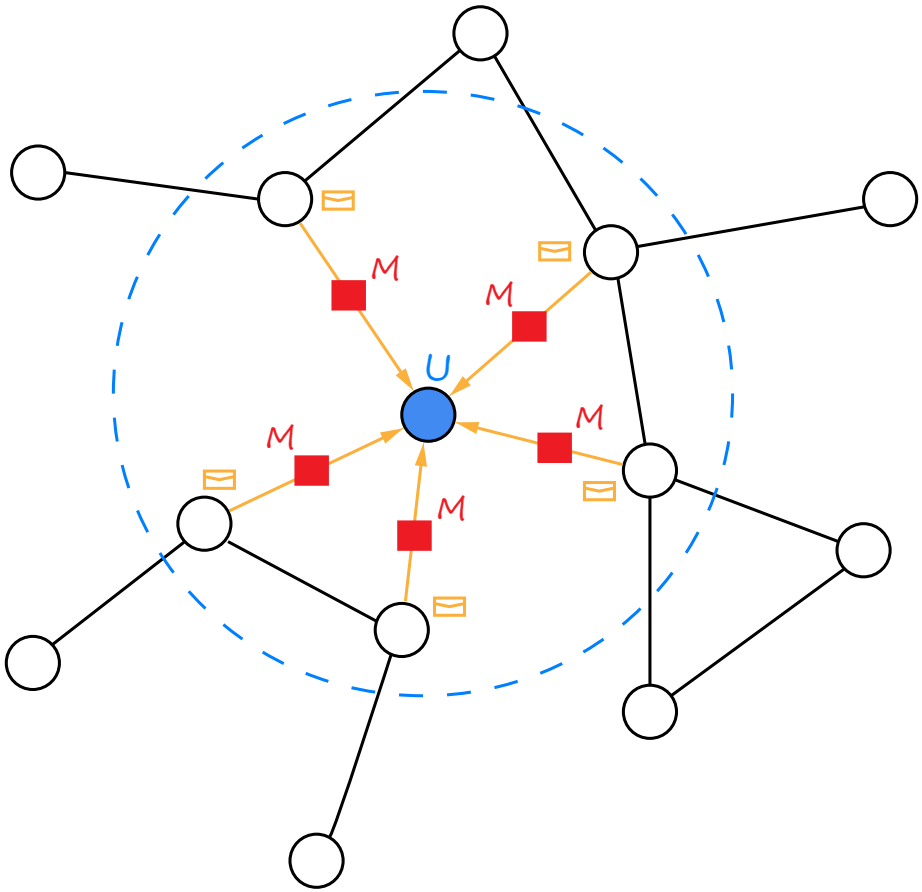
arXiv:1704.01212 [cs.LG]

Embedding $h_i^{(t)}$ for each node V_i is obtained by aggregating a function $\mathcal{M}$ of features $h_j^{(t-1)}$ (and possibly edges) of all neighbouring nodes V_j with the node V_i itself:

$$h_i^{(t)} = \mathcal{U}^{(t)}\left(h_i^{(t-1)}, \square_{j \in \mathcal{N}(i)} \mathcal{M}^{(t)}(h_i^{(t-1)}, h_j^{(t-1)}, e_{\{j,i\}})\right),$$

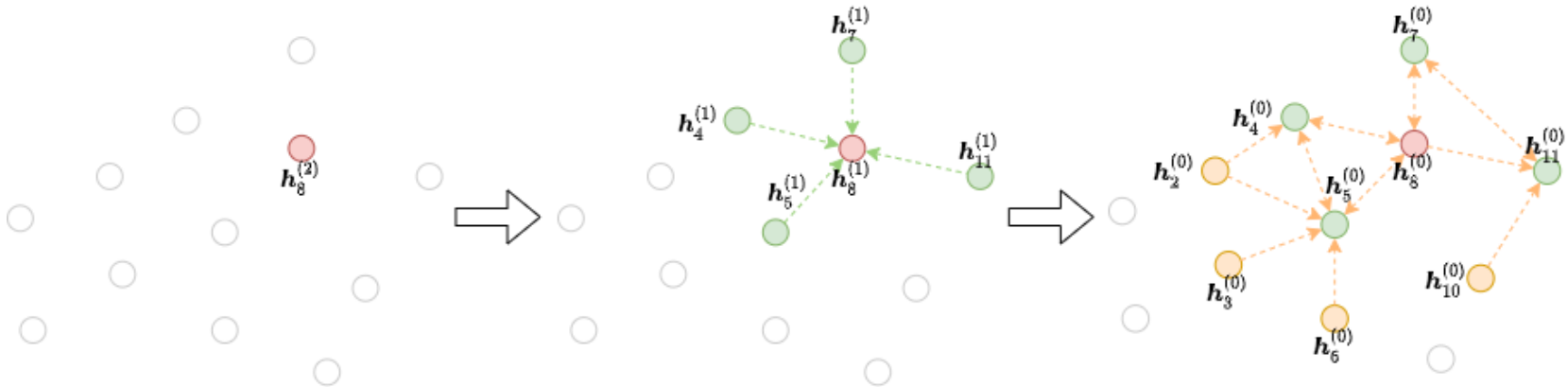
$\square$ - aggregation function (ex. $\sum_j$),
$\mathcal{U}$, $\mathcal{M}$ - update/message differentiable functions (ex. MLP),
$e_{\{j,i\}}$ - edge features from node j to node i.

$$h_i^{(t)} = \mathcal{U}^{(t)}\left(h_i^{(t-1)}, \square_{j \in \mathcal{N}(i)} \mathcal{M}^{(t)}(h_i^{(t-1)}, h_j^{(t-1)}, e_{\{j,i\}})\right)$$



Iterative process for 2-layer message passing (2-hop neighbourhood):

At each layer we obtain a bipartite graph structure



source

Graph Neural Networks: A Review of Methods and Applications arXiv:1812.08434 [cs.LG]

Name	Variant	Aggregator	Updater	
Spectral Methods	ChebNet	$\mathbf{N}_k = \mathbf{T}_k(\tilde{\mathbf{L}})\mathbf{X}$	$\mathbf{H} = \sum_{k=0}^{K} \mathbf{N}_k \mathbf{\Theta}_k$	
	1^{st}-order model	$\mathbf{N}_0 = \mathbf{X}$ $\mathbf{N}_1 = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{X}$	$\mathbf{H} = \mathbf{N}_0\mathbf{\Theta}_0 + \mathbf{N}_1\mathbf{\Theta}_1$	
	Single parameter	$\mathbf{N} = (\mathbf{I}_N + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}})\mathbf{X}$	$\mathbf{H} = \mathbf{N}\mathbf{\Theta}$	
	GCN	$\mathbf{N} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}$	$\mathbf{H} = \mathbf{N}\mathbf{\Theta}$	Graph Convolutions
Non-spectral Methods	Neural FPs	$\mathbf{h}^t_{\mathcal{N}_v} = \mathbf{h}^{t-1}_v + \sum_{k=1}^{\mathcal{N}_v} \mathbf{h}^{t-1}_k$	$\mathbf{h}^t_v = \sigma(\mathbf{h}^t_{\mathcal{N}_v}\mathbf{W}^{\mathcal{N}_v}_L)$	
	DCNN	Node classification: $\mathbf{N} = \mathbf{P}^*\mathbf{X}$ Graph classification: $\mathbf{N} = 1^T_N\mathbf{P}^*\mathbf{X}/N$	$\mathbf{H} = f(\mathbf{W}^c \odot \mathbf{N})$	
	GraphSAGE	$\mathbf{h}^t_{\mathcal{N}_v} = \text{AGGREGATE}_t(\{\mathbf{h}^{t-1}_u, \forall u \in \mathcal{N}_v\})$	$\mathbf{h}^t_v = \sigma(\mathbf{W}^t \cdot [\mathbf{h}^{t-1}_v \| \mathbf{h}^t_{\mathcal{N}_v}])$	
Graph Attention Networks	GAT	$\alpha_{vk} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{h}_v \| \mathbf{W}\mathbf{h}_k]))}{\sum_{j\in\mathcal{N}_v}\exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{h}_v \| \mathbf{W}\mathbf{h}_j]))}$ $\mathbf{h}^t_{\mathcal{N}_v} = \sigma\left(\sum_{k\in\mathcal{N}_v}\alpha_{vk}\mathbf{W}\mathbf{h}_k\right)$ Multi-head concatenation: $\mathbf{h}^t_{\mathcal{N}_v} = \Big\|_{m=1}^{M}\sigma\left(\sum_{k\in\mathcal{N}_v}\alpha^m_{vk}\mathbf{W}^m\mathbf{h}_k\right)$ Multi-head average: $\mathbf{h}^t_{\mathcal{N}_v} = \sigma\left(\frac{1}{M}\sum_{m=1}^{M}\sum_{k\in\mathcal{N}_v}\alpha^m_{vk}\mathbf{W}^m\mathbf{h}_k\right)$	$\mathbf{h}^t_v = \mathbf{h}^t_{\mathcal{N}_v}$	Attention mechanism
Gated Graph Neural Networks	GGNN	$\mathbf{h}^t_{\mathcal{N}_v} = \sum_{k\in\mathcal{N}_v}\mathbf{h}^{t-1}_k + \mathbf{b}$	$\mathbf{z}^t_v = \sigma(\mathbf{W}^z\mathbf{h}^t_{\mathcal{N}_v} + \mathbf{U}^z\mathbf{h}^{t-1}_v)$ $\mathbf{r}^t_v = \sigma(\mathbf{W}^r\mathbf{h}^t_{\mathcal{N}_v} + \mathbf{U}^r\mathbf{h}^{t-1}_v)$ $\widetilde{\mathbf{h}^t_v} = \tanh(\mathbf{W}\mathbf{h}^t_{\mathcal{N}_v} + \mathbf{U}(\mathbf{r}^t_v \odot \mathbf{h}^{t-1}_v))$ $\mathbf{h}^t_v = (1 - \mathbf{z}^t_v) \odot \mathbf{h}^{t-1}_v + \mathbf{z}^t_v \odot \widetilde{\mathbf{h}^t_v}$	

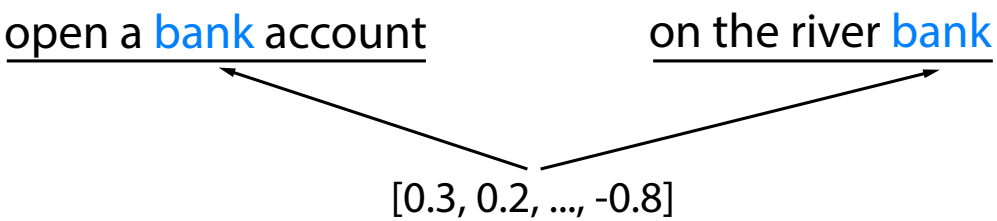
Transformers

Attention Is All You Need

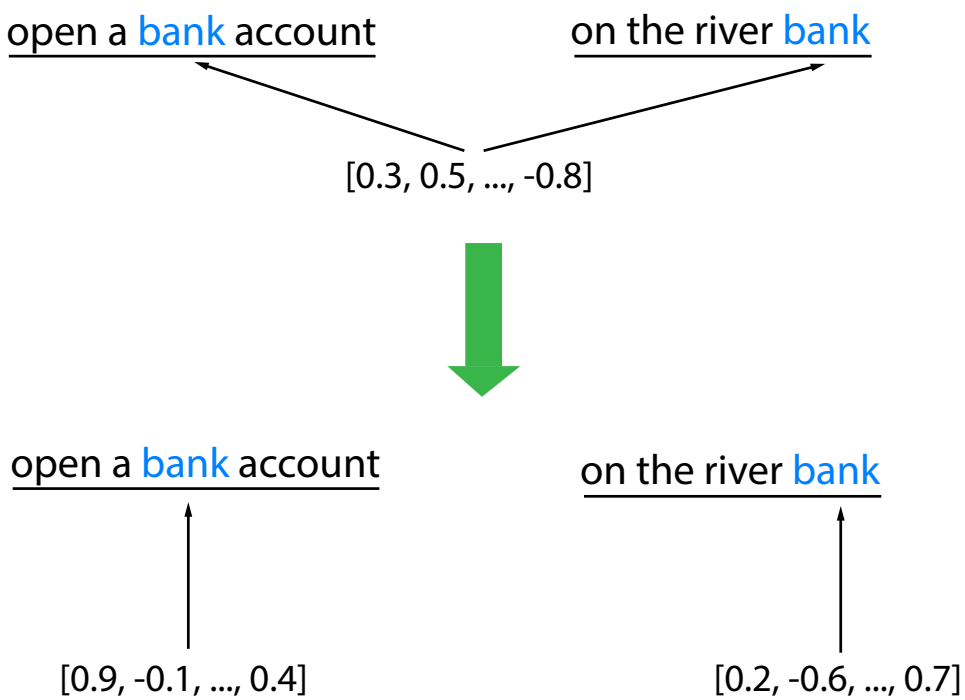
arXiv:1706.03762 [cs.CL]

Contextual representations

- Word embeddings (word2vec, Glove) are *pre-trained* on text corpus based on co-occurence statistics.
- Word embeddings are applied in a **context free** manner.

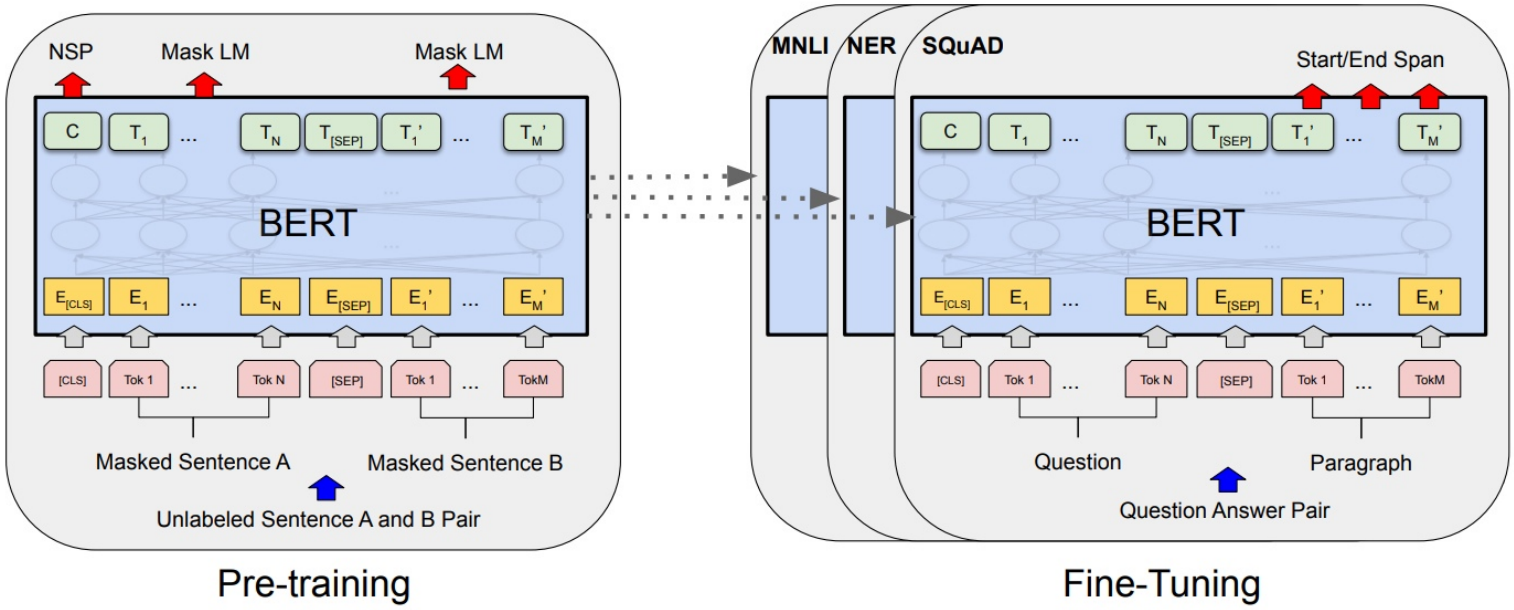


We need contextual embeddings



Bidirectional Encoder Representations from Transformers

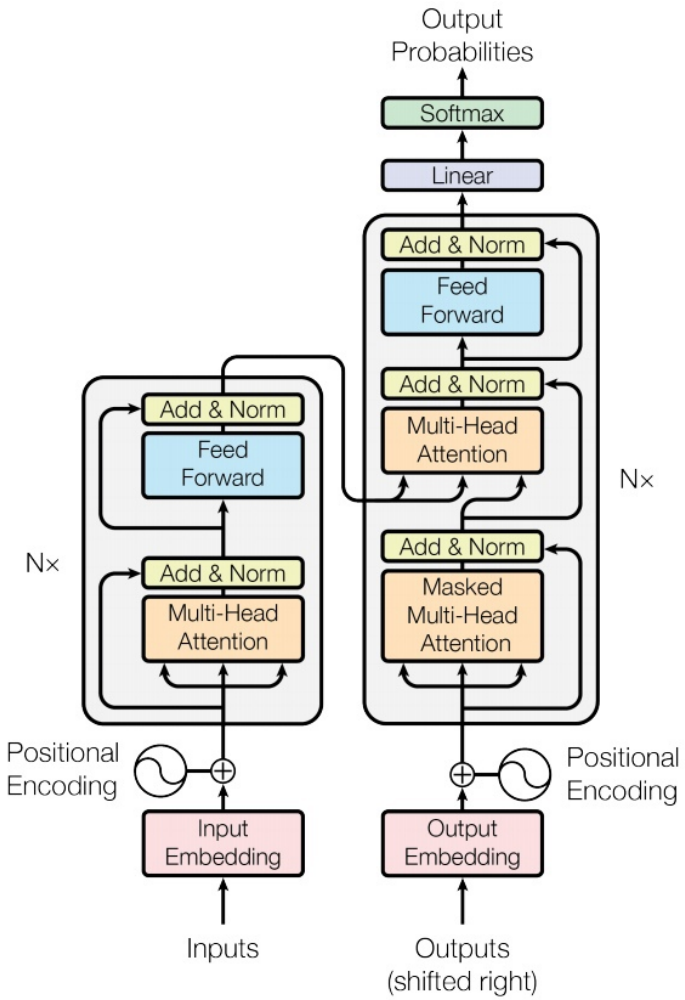
BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
arXiv:1810.04805 [cs.CL]

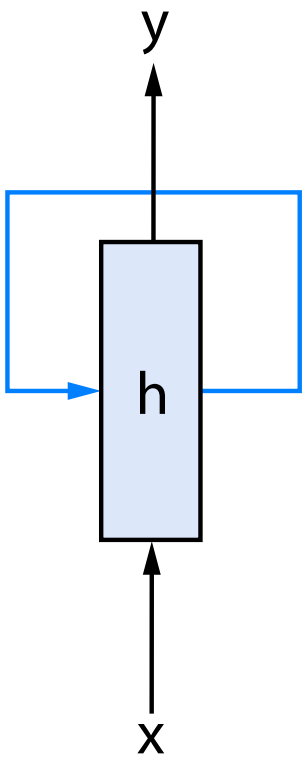


BERT pretraining - two unsupervised tasks:

1. Masked LM: mask some input words and predict them.
2. Next Sentence Prediction: predict whether *Sentence B* is actual sentence that proceeds *Sentence A*.

Transformer architecture





1. Simplest RNN ($\mathbf{y}_t = \mathbf{h}_t$):

$$\mathbf{Y}_{(t)} = \phi\left(\mathbf{X}_{(t)}\mathbf{W} + \mathbf{Y}_{(t-1)}\mathbf{W}_y + \mathbf{b}\right).$$

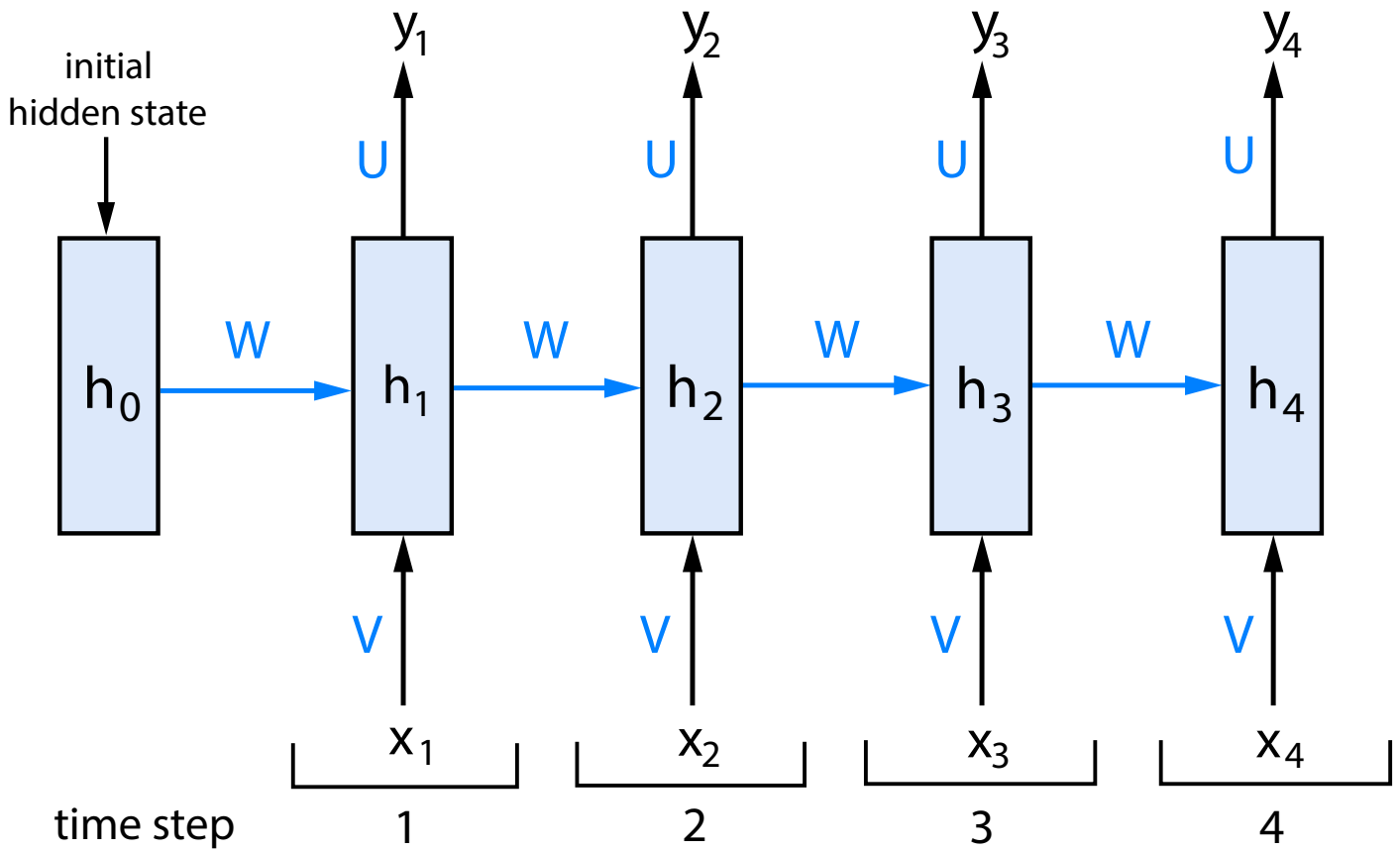
2. Rewrite in concatenated form:

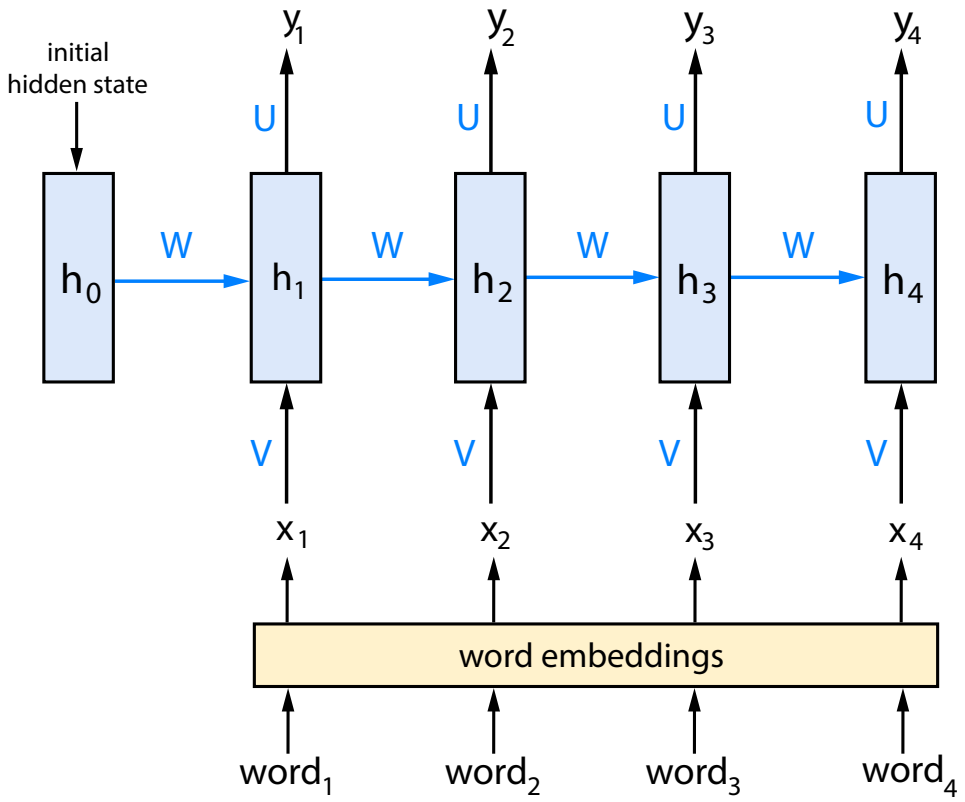
$$\mathbf{Y}_{(t)} = \phi\left(\begin{bmatrix}\mathbf{X}_{(t)} & \mathbf{Y}_{(t-1)}\end{bmatrix}\mathbf{W} + \mathbf{b}\right),$$

$$\mathbf{W} = \begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix}.$$

3. More complex:
   - $\mathbf{y}_t = g\left(\mathbf{h}_t\right)$
   - LSTM
   - GRU
   - Bidirectional
   - Multilayer

Unfold the RNN!





Use cases:

- Tagging (ex: named entity recognition, part-of-speech tagging)
- Sentence classification (ex: sentiment analysis)

Sequence to Sequence Learning with Neural Networks
arXiv:1409.3215 [cs.CL]

- Source sequence: $S = (x_1, \cdots, x_S)$.
- Target sequence: $T = (y_1, \cdots, y_T)$.

Translation
$S \to T$

- Maximize log probability of correct translation T given source S in set $\mathcal{S}$:

$$\mathcal{L} = \frac{1}{|\mathcal{S}|} \sum_{(T,S)\in\mathcal{S}} \log p(T \mid S)$$

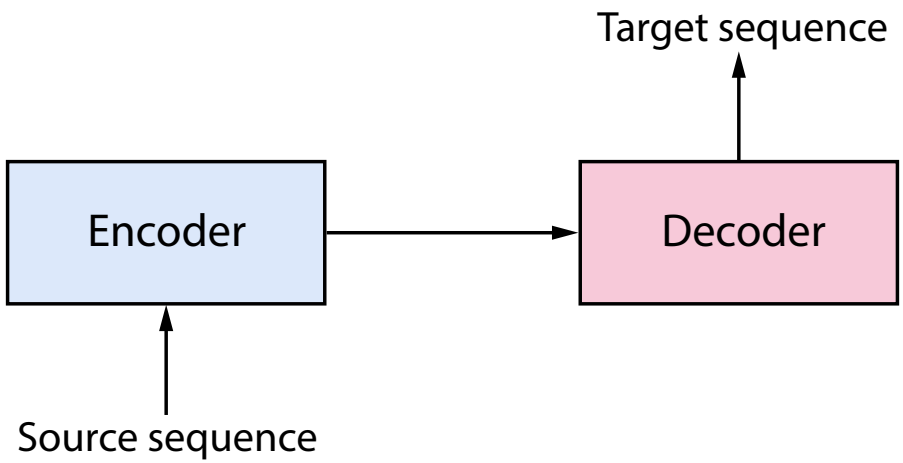
- Neural network defines a probability over translation by decomposing the joint probability into the ordered conditionals:

$$p(y_1, \cdots, y_T \mid x_1, \cdots, x_S) = \prod_{t=1}^{T} p(y_t \mid y_1, \cdots, y_{t-1}, v),$$

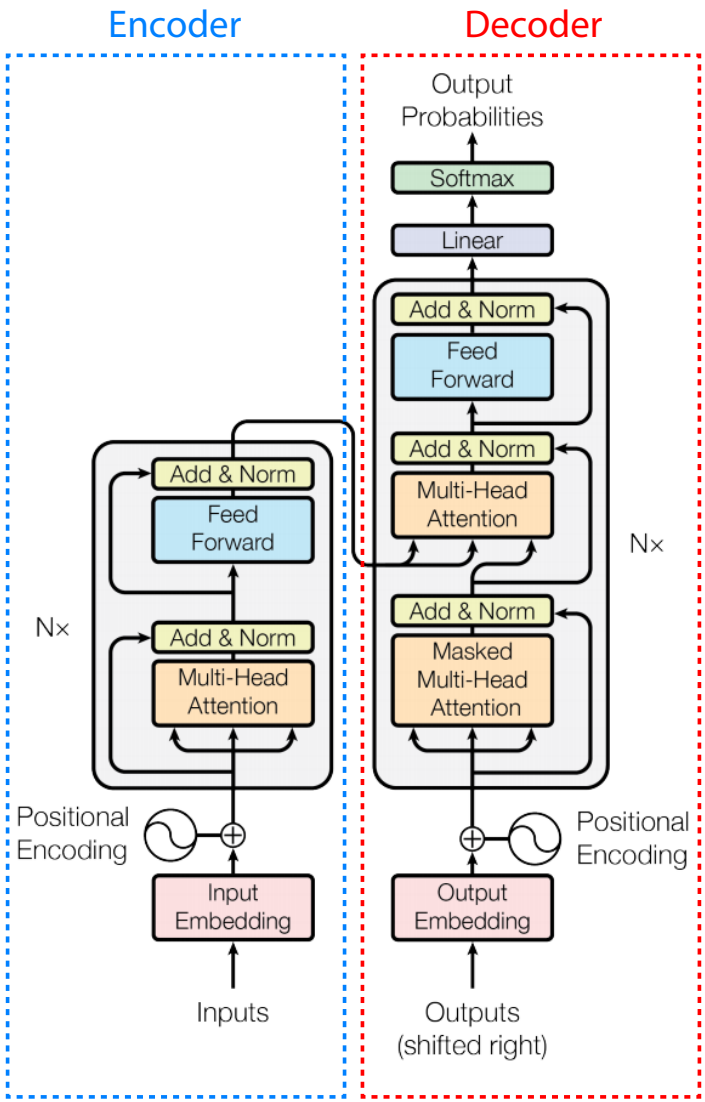
v - representation of $(x_1, \cdots, x_S)$.

Encoder-Decoder

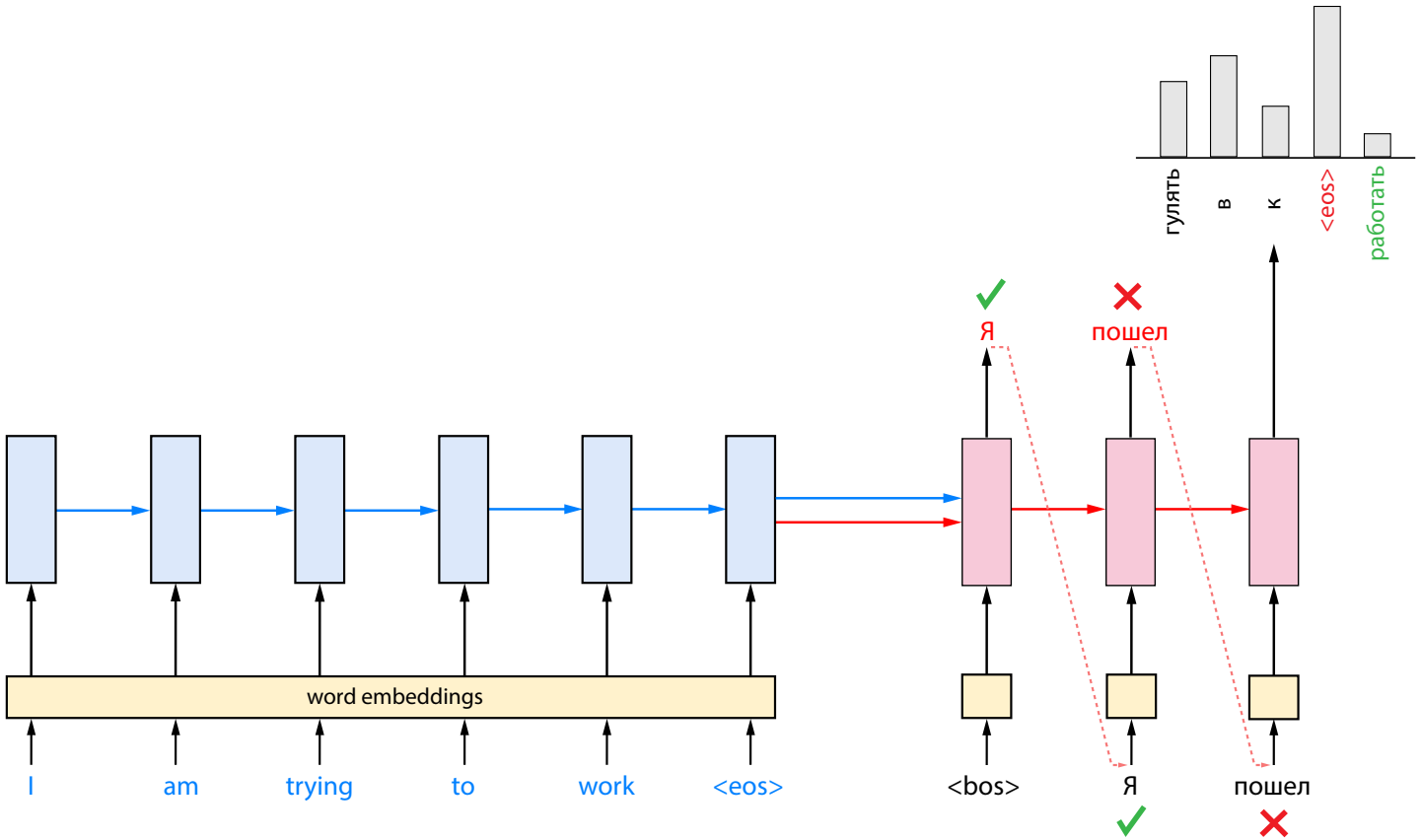
- Encoder produces a representation of source sequence and passes it to decoder.
- Decoder generates target sequence based on encoding.



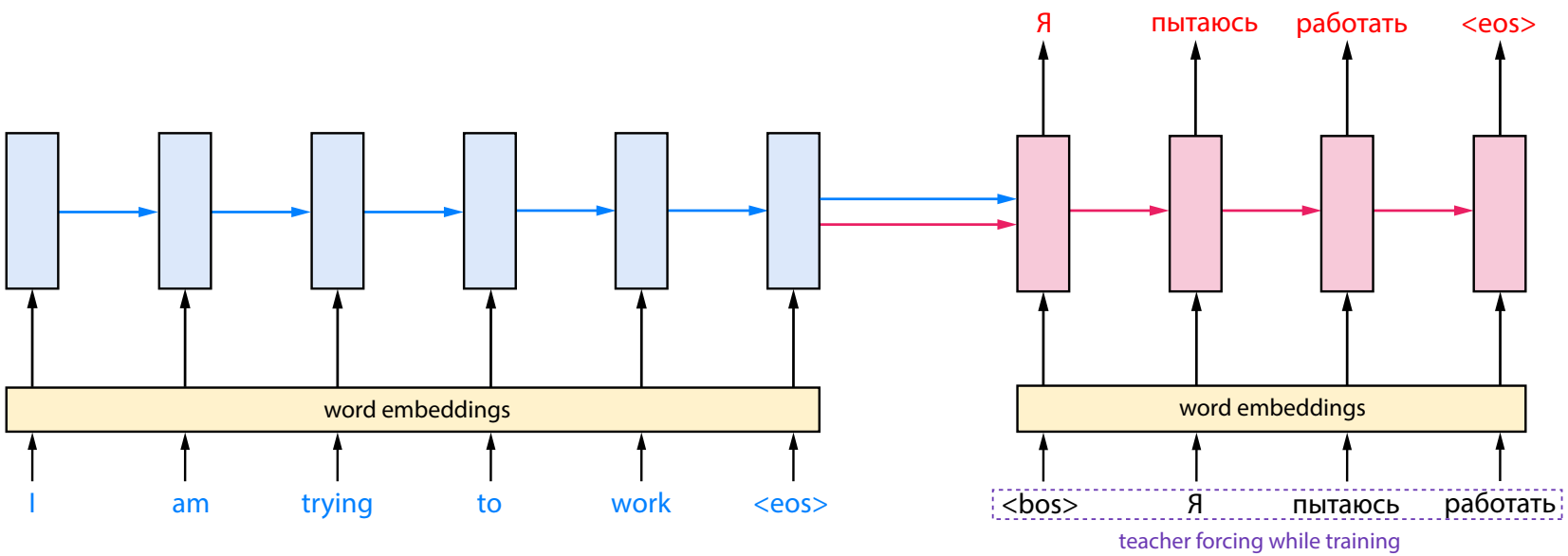
Transformer: Encoder-Decoder



- Simple case: two RNNs.
- Encoder's final state $\mathbf{h}_S$ - representation for the entire source sentence - serves as initial hidden state for Decoder RNN.



- Teacher forcing (training): replace the output $y(t)$ generated by network with the actual (expected) value in the training dataset.
- Bottleneck: all source information is encoded in single vector.



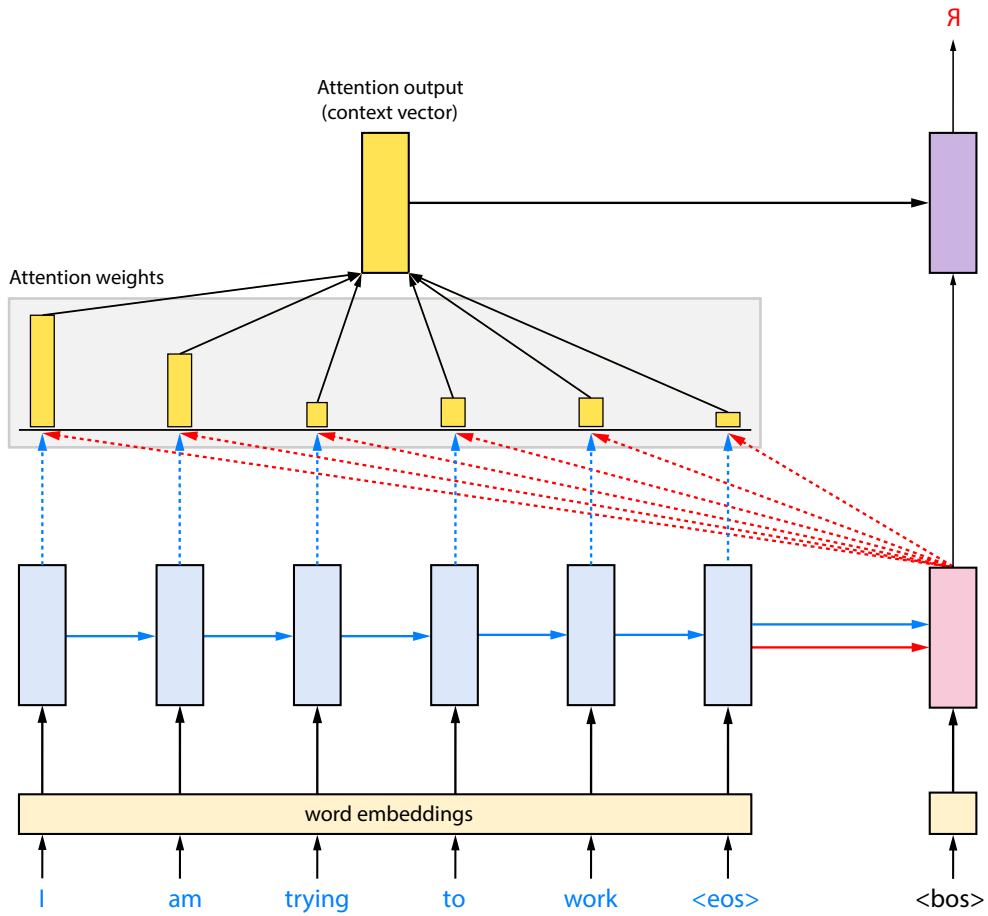
Attention

Neural Machine Translation by Jointly Learning to Align and Translate
arXiv:1409.0473 [cs.CL]

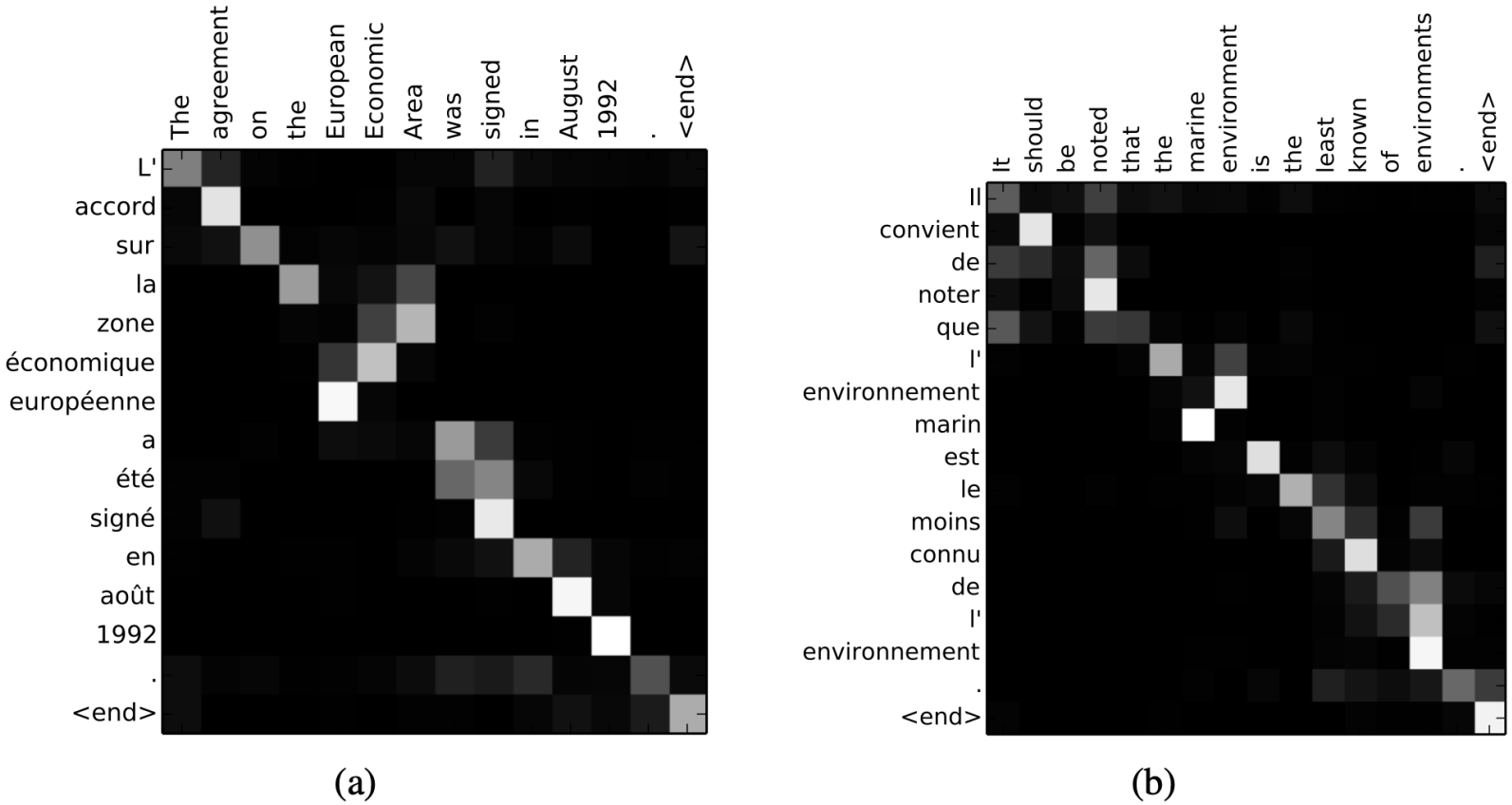
Effective Approaches to Attention-based Neural Machine Translation
arXiv:1508.04025 [cs.CL]

Attention Is All You Need
arXiv:1706.03762 [cs.CL]

- At different decoder steps, Attention lets a model to focus on different parts of the input (according to their relevance for the current decoder state).



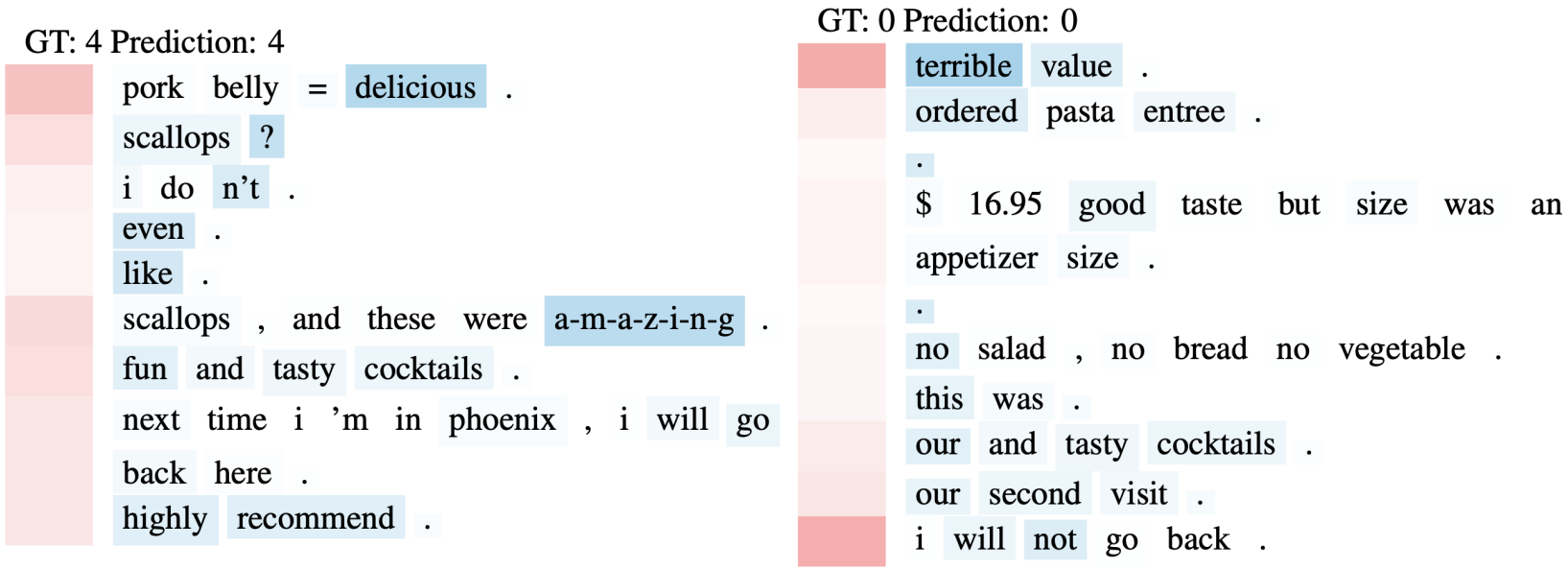
English ⟶ French



source

Document Classification

$4 \longleftrightarrow$ star $5, 0 \longleftrightarrow$ star 1.



Hierarchical Attention Networks for Document Classification

Calculation for 1 pair $(S \leftrightarrow T)$ in batch

- d - hidden dimension of the model, m, n - seq lens.
- A set of m query vectors (come from the decoder):

$$\mathbf{Q} \in \mathbb{R}^{m \times d}$$

- A set of n key and n value vectors (come from the output of encoder):

$$\mathbf{K} \in \mathbb{R}^{n \times d}, \quad \mathbf{V} \in \mathbb{R}^{n \times d}.$$

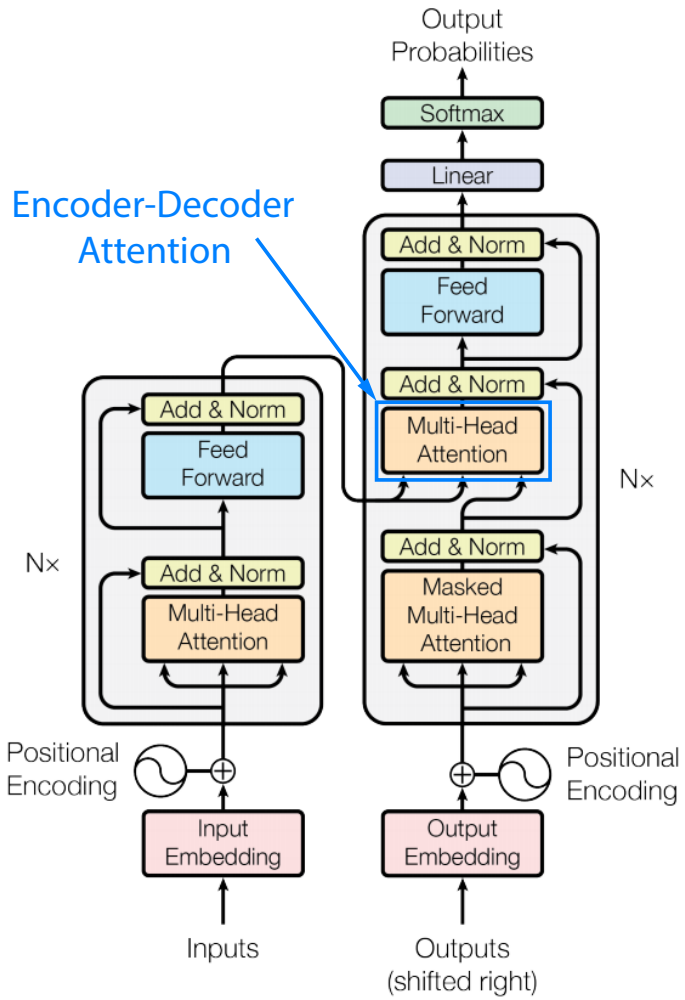
- Alignment scores and Compatibility Matrix $\mathbf{A}$:

$$\mathbf{R} = \text{score}\left(\mathbf{Q}, \mathbf{K}\right) \in \mathbb{R}^{m \times n},$$

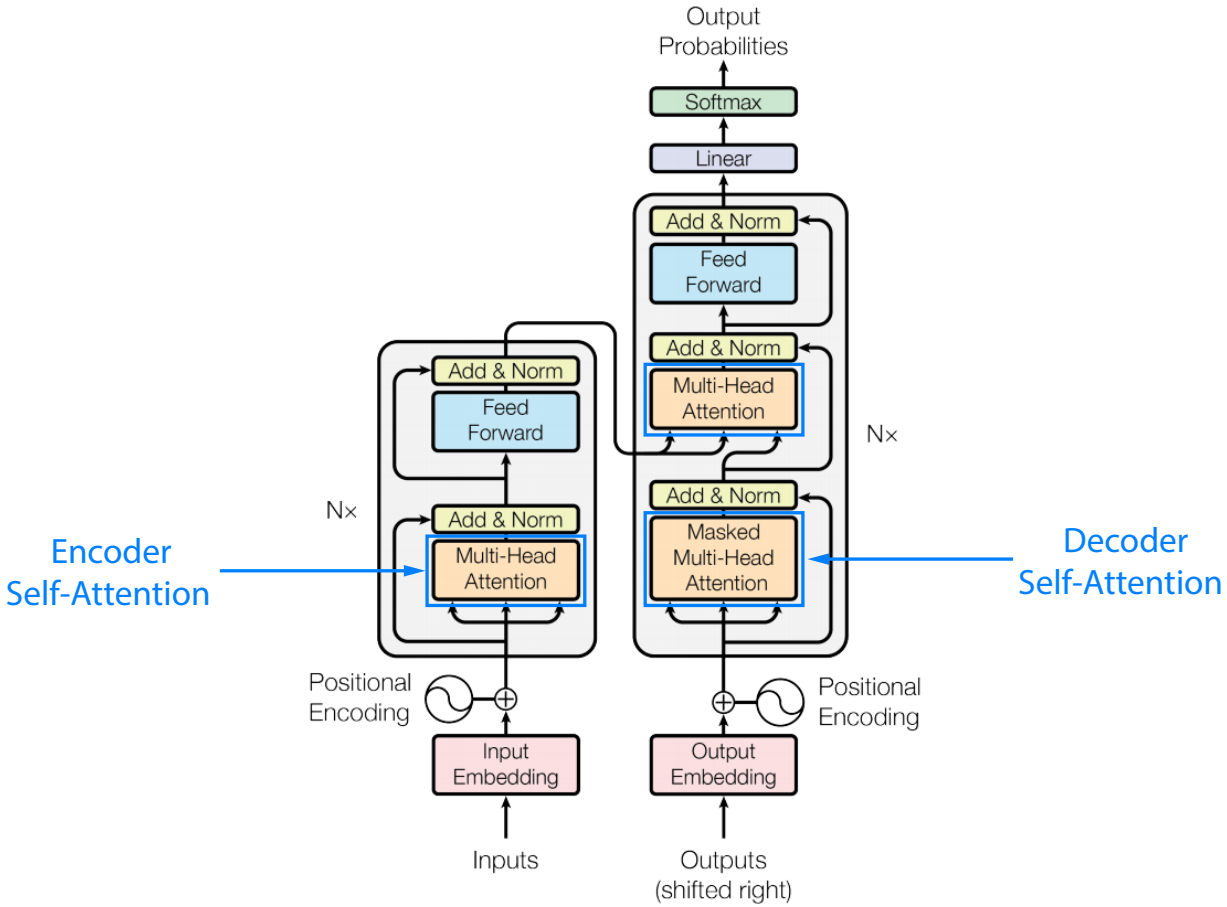
$$\mathbf{A} = \text{Softmax}\left(\mathbf{R}\right).$$

- Attention - weighted sum of value vectors:

$$\text{Attention}\left(\mathbf{Q}, \mathbf{K}, \mathbf{V}\right) = \mathbf{A}\mathbf{V} \in \mathbf{R}^{m \times d}.$$


Output Probabilities
Softmax
Linear
Encoder-Decoder Attention
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

- Self-attention layer: keys, values and queries come from the **same place**.



Positional Encoding

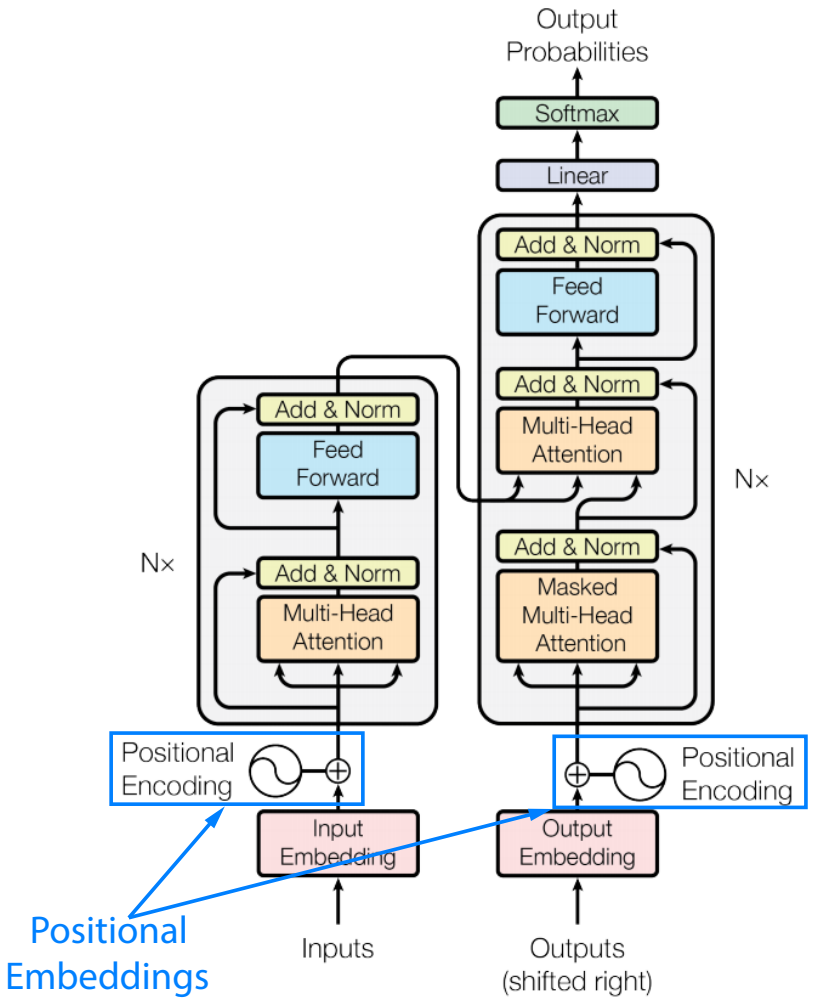
- Positional embeddings **PE**: vectors of the same dimension d as embeddings.
- Encode the positions of words within a sequence.
- Fixed positional embeddings:

$$PE_{(p,2i)} = \sin(p/10000^{2i/d}),$$

$$PE_{(p,2i+1)} = \cos(p/10000^{2i/d}).$$

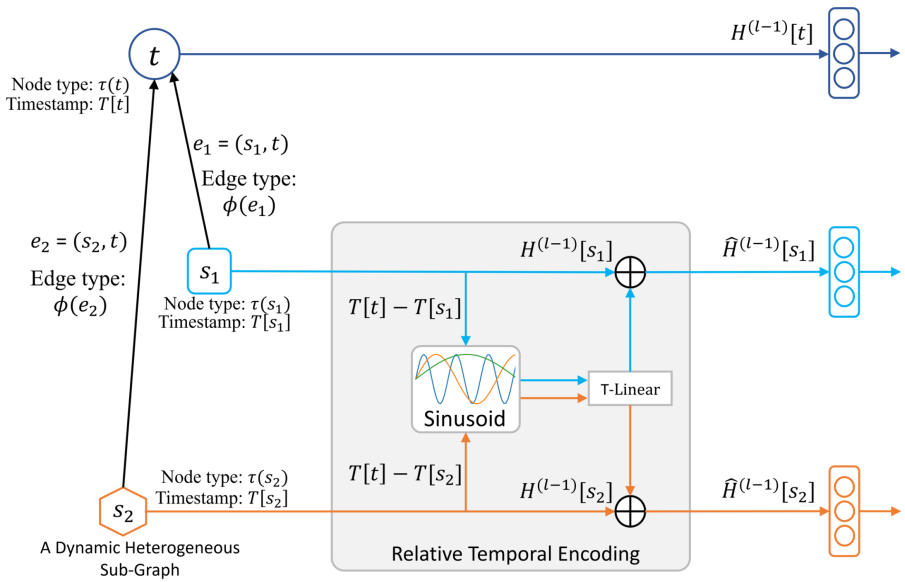
- We don't need it in graphs: invariance to the order of nodes unless we are encoding some (time) dynamic.

Simply add the positional embedding to the word embedding.



Relative Temporal Encoding

- Handles graph dynamic



$$Base(\Delta T(t,s), 2i) = sin\left(\Delta T_{t,s} / 10000^{\frac{2i}{d}}\right) \quad (6)$$

$$Base(\Delta T(t,s), 2i+1) = cos\left(\Delta T_{t,s} / 10000^{\frac{2i+1}{d}}\right) \quad (7)$$

$$RTE(\Delta T(t,s)) = \text{T-Linear}\left(Base(\Delta T_{t,s})\right) \quad (8)$$

Finally, the temporal encoding relative to the target node t is added to the source node s' representation as follows:

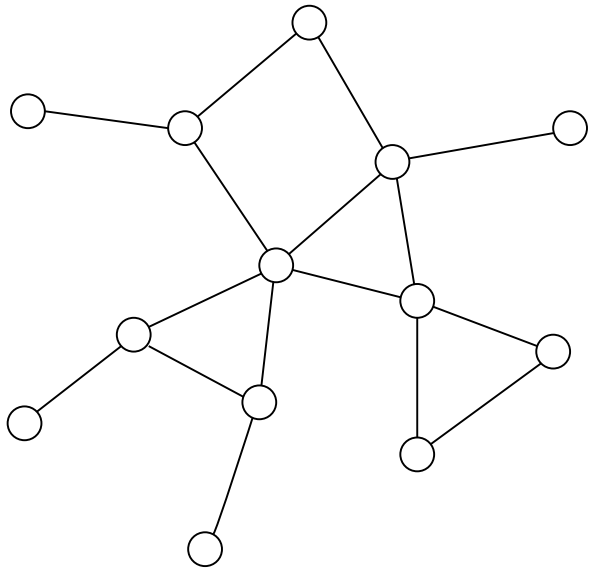
$$\widehat{H}^{(l-1)}[s] = H^{(l-1)}[s] + RTE(\Delta T(t,s)) \quad (9)$$

Heterogeneous Graph Transformer

Graph Attention Networks

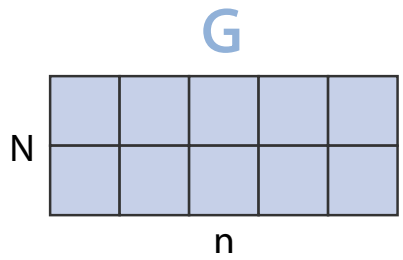
Graph Attention Networks
arXiv:1710.10903 [stat.ML]

- G - graph with N nodes and n features.



- Represent the graph in the tensor form:

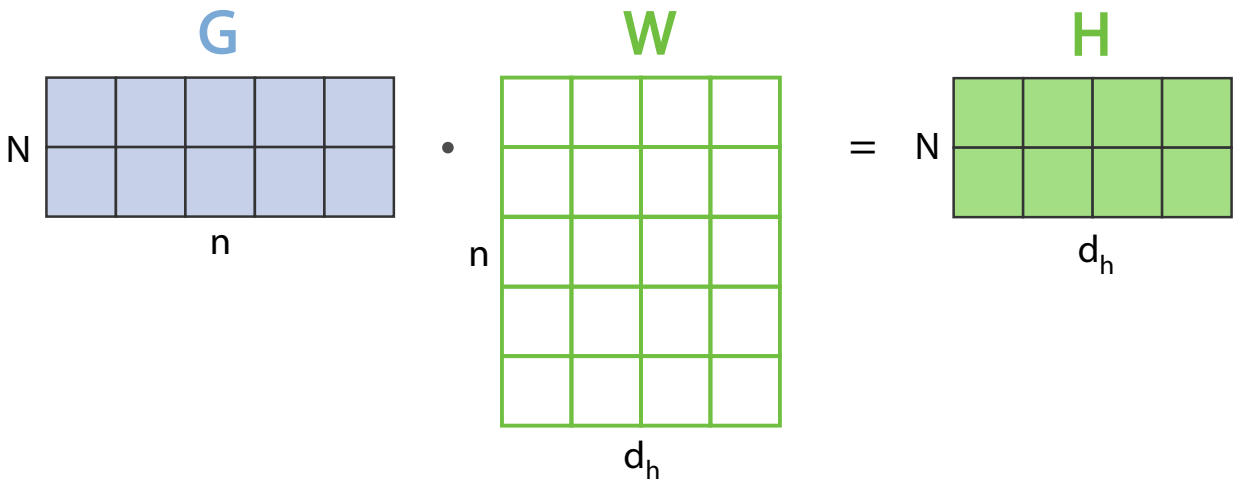
$$\mathbf{G} \in \mathbb{R}^{N \times n}$$



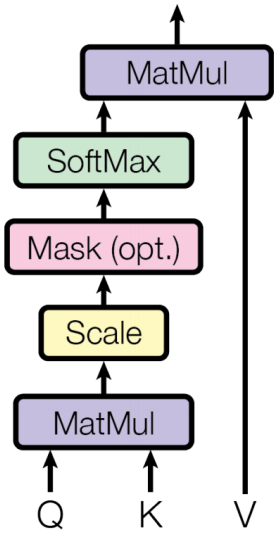
- Compute initial node embeddings through a linear projection $\mathbf{W} \in \mathbb{R}^{n \times d_h}$:

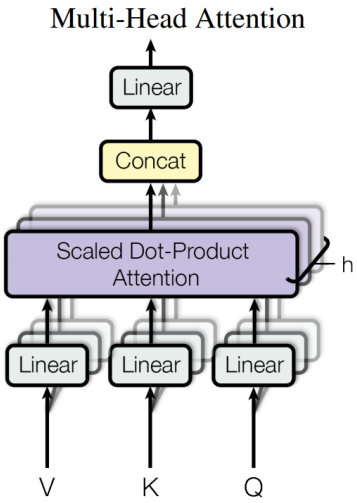
$$\mathbf{G} \in \mathbb{R}^{N \times n} \longrightarrow \mathbf{H} \in \mathbb{R}^{N \times d_h}$$

$$\mathbf{H} = \mathbf{G}\mathbf{W}$$



Multihead Attention (Vaswani et al., 2017)





- Linearly project initial node embeddings $\mathbf{H} \in \mathbb{R}^{N \times d_h}$ (query, key, value):

$$\mathbf{Q} = \mathbf{H}\mathbf{W}^Q, \mathbf{K} = \mathbf{H}\mathbf{W}^K, \mathbf{V} = \mathbf{H}\mathbf{W}^V,$$

$$\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V \in \mathbb{R}^{d_h \times d}.$$

- Split into M heads and compute compatibility matrix $\mathbf{A} \in \mathbb{R}^{M \times N \times N}$ for graph nodes:

$$\mathbf{A} = \text{Softmax}\left(\frac{\mathbf{R}}{\sqrt{d/M}}\right) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d/M}}\right).$$

- Compute attention messages for each head:

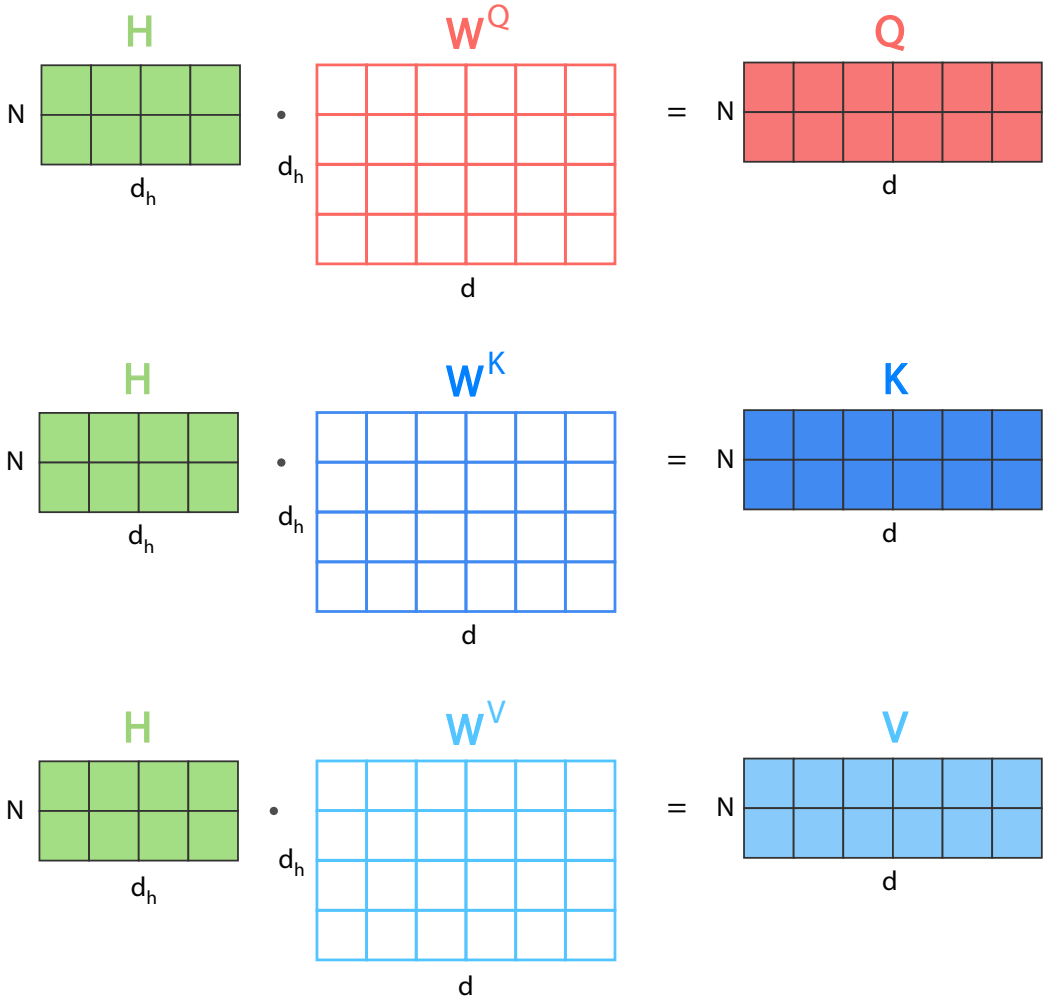
$$\mathbf{H}' = \mathbf{A}\mathbf{V} \in \mathbb{R}^{M \times N \times d/M}$$

- Concatenate heads and project out with $\mathbf{W}^O \in \mathbb{R}^{d \times d}$:

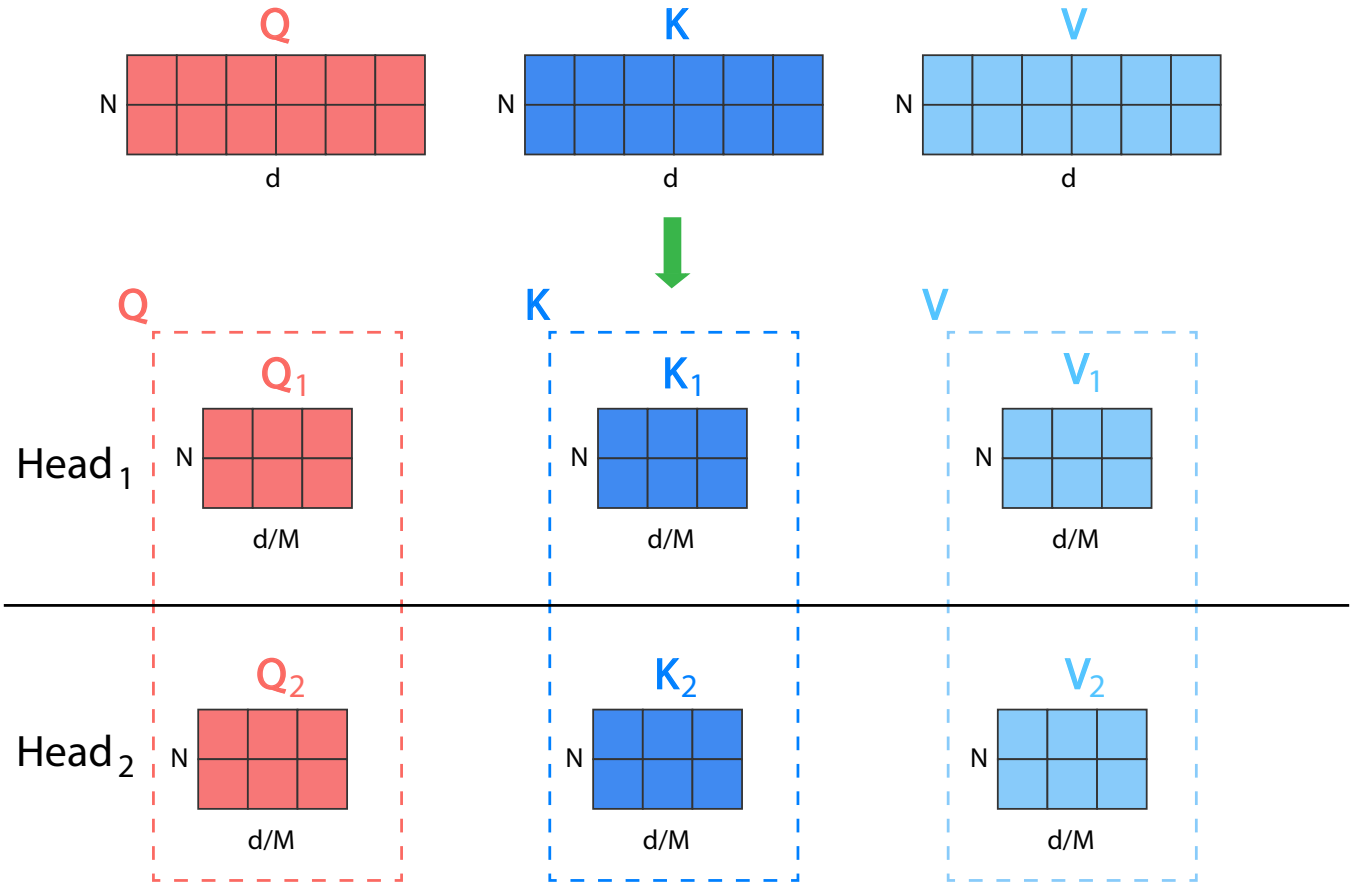
$$MHA = \text{Concat}\left(\mathbf{H}'_1, \ldots, \mathbf{H}'_M\right)\mathbf{W}^O \in \mathbb{R}^{N \times d}.$$

Multihead self-attention: step 1

Create query, key and value: $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$.

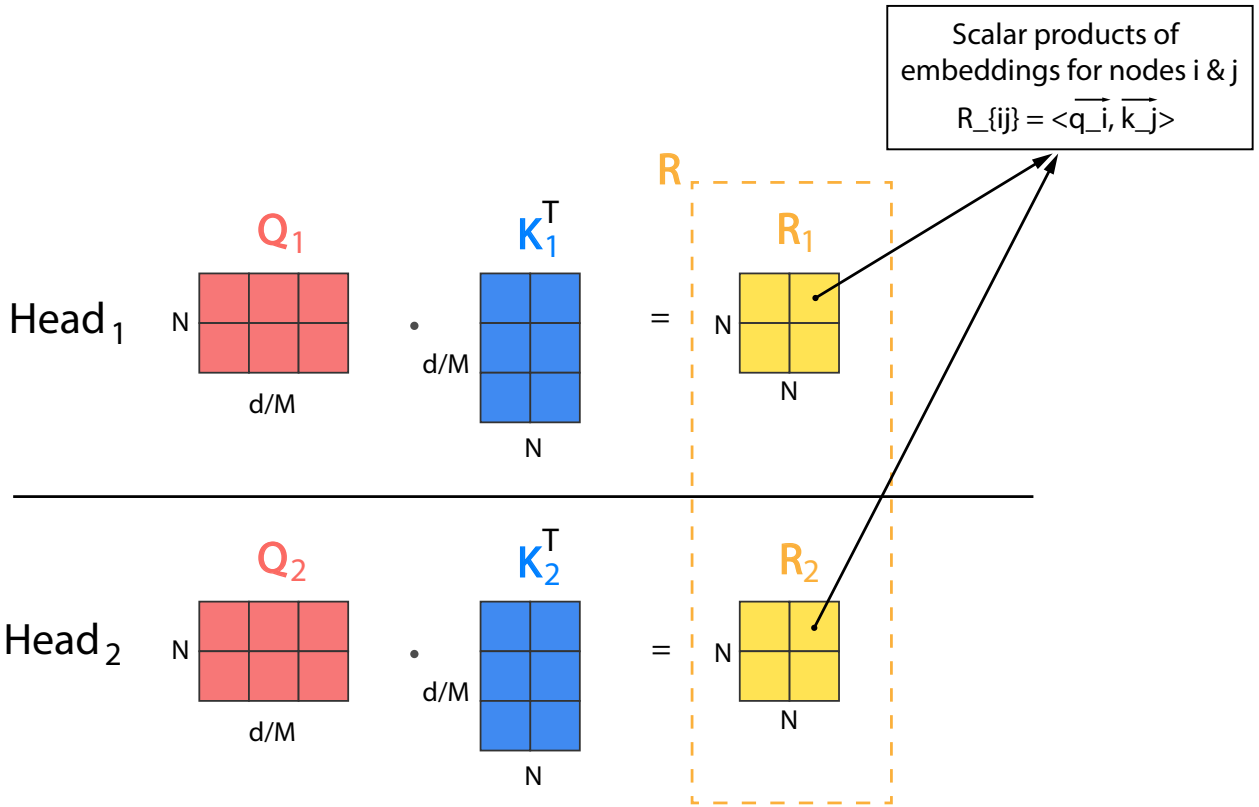


Split computations into $M = 2$ heads.
This adds extra dimension: $[batch\_size, N, d] \to [batch\_size, M, N, d/M]$.



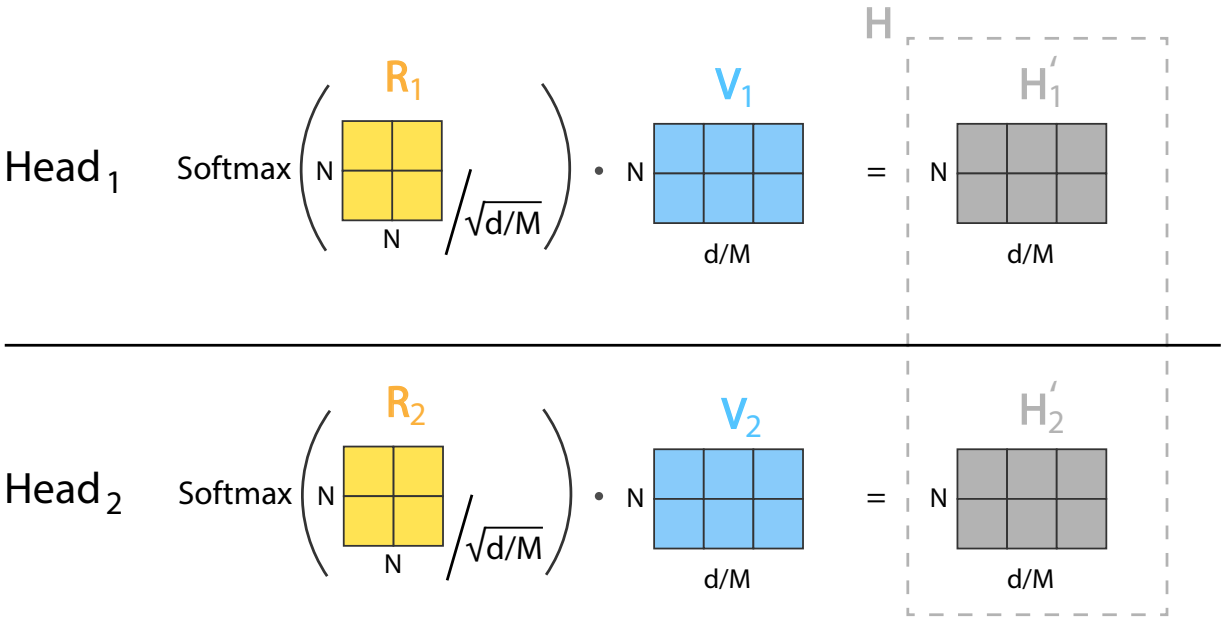
Multihead self-attention: step 3

- Compute compatibility matrix for nodes: $\mathbf{A} \in \mathbb{R}^{M \times N \times N}$.
- Note how each head learns its own representations (*compare with convolutions*)



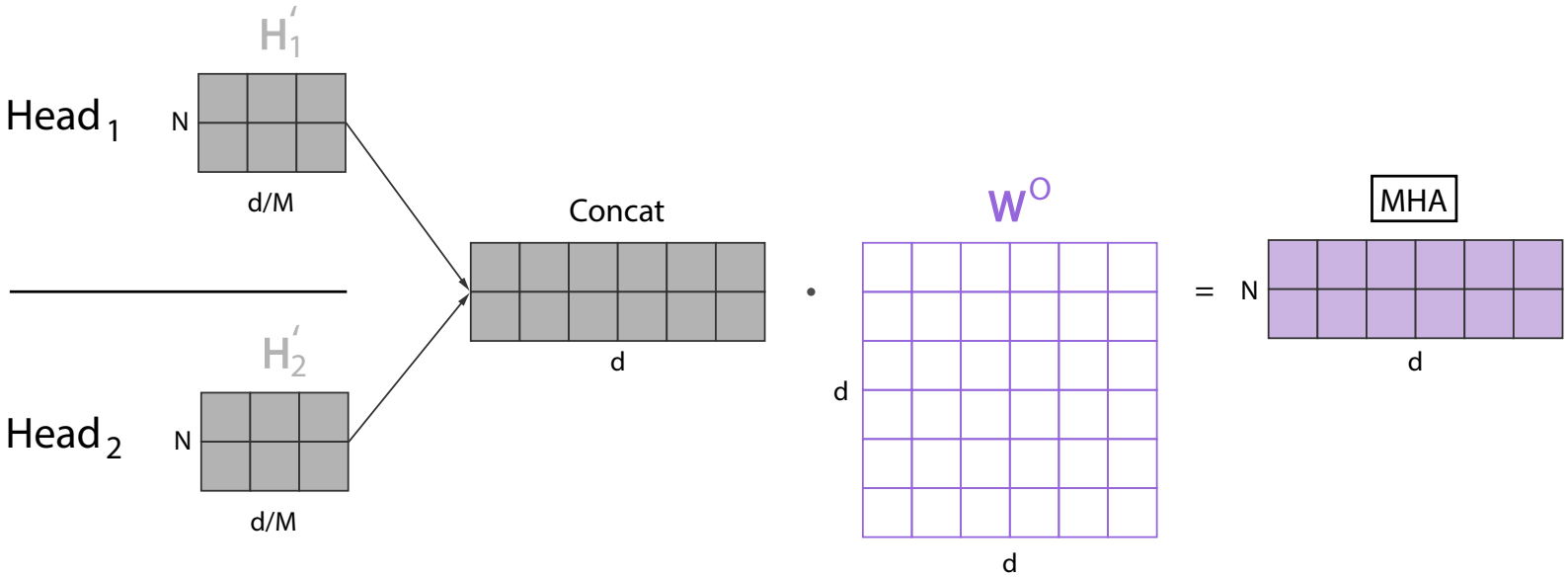
Multihead self-attention: step 4

- Compute messages (attention outputs) for each head in parallel.
- Note how different heads update embeddings for each of N nodes using different subspaces of dimension d/M.



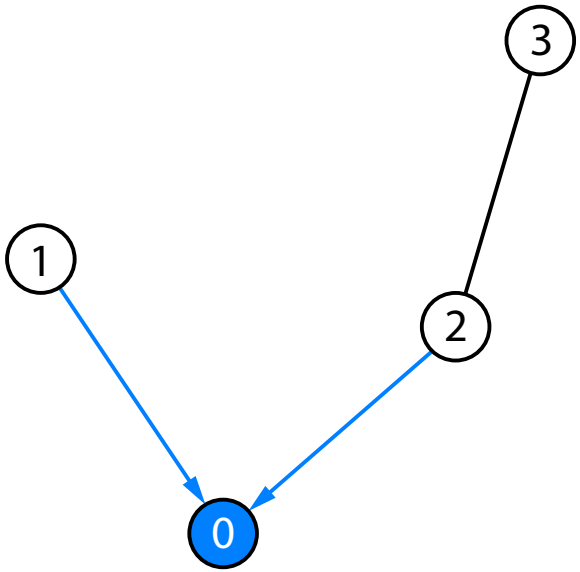
Multihead self-attention: step 5

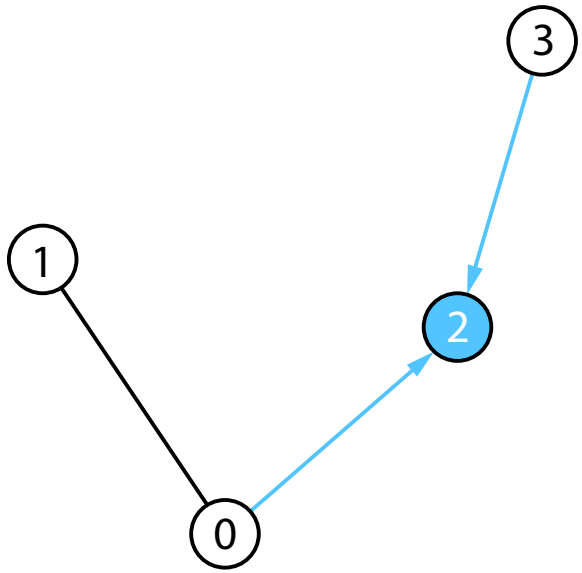
- Concatenate outputs and project them to get the final values.



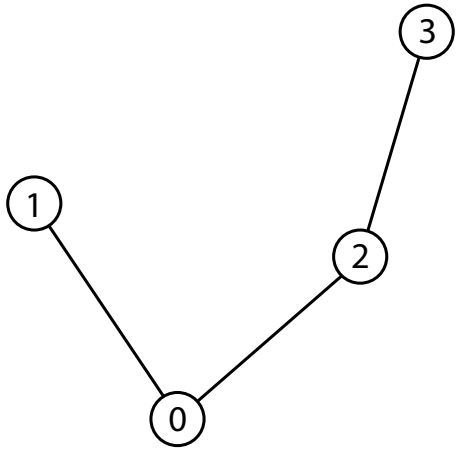
Message Passing Mask

- How can we incorporate graph structure?
- We need to take into account only nodes in some neighbourhood (say, 1st order neighbours)





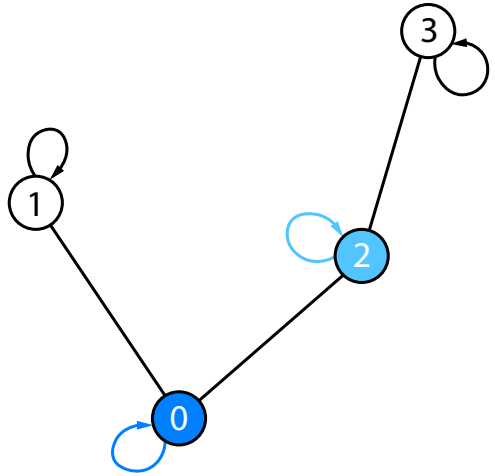
Message Passing Mask



- Compute adjacency matrix:

$$\mathbf{Adj} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Message Passing Mask



- Add self-loops $\Longleftrightarrow$ add identity matrix:

$$\mathbf{Adj} \longrightarrow \mathbf{Adj} + \mathbf{I} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \mathbf{1} & 1 & 1 & 0 \\ 1 & \mathbf{1} & 0 & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 1 & \mathbf{1} \end{bmatrix} \end{array}$$

Message Passing Mask

- Add mask:

$$\textbf{Mask} = -10^9 \cdot (1 - \textbf{Adj})$$

.

- Mask is applied before **Softmax**. Multiplied by (almost) negative infinity to zero out nodes after applying softmax.

$$\textbf{Adj} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

$$\textbf{Mask} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & \text{-np.inf} \\ 0 & 0 & \text{-np.inf} & \text{-np.inf} \\ 0 & \text{-np.inf} & 0 & 0 \\ \text{-np.inf} & \text{-np.inf} & 0 & 0 \end{bmatrix} \end{array}$$

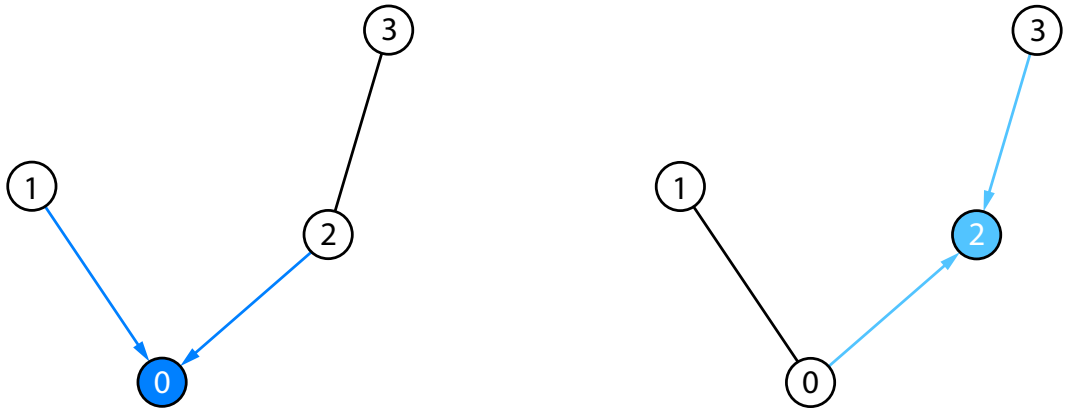
- Suppose we have score matrix: $\mathbf{R} = \text{score}(\mathbf{Q}, \mathbf{K})$.
- Apply mask before computing attention messages:

$$\mathbf{R}' = \mathbf{R} + \mathbf{Mask}$$

$$R' = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} r_{00} & r_{01} & r_{02} & r_{03} \\ r_{10} & r_{11} & r_{12} & r_{13} \\ r_{20} & r_{21} & r_{22} & r_{23} \\ r_{30} & r_{31} & r_{32} & r_{33} \end{bmatrix} \end{array} + \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & \text{-np.inf} \\ 0 & 0 & \text{-np.inf} & \text{-np.inf} \\ 0 & \text{-np.inf} & 0 & 0 \\ \text{-np.inf} & \text{-np.inf} & 0 & 0 \end{bmatrix} \end{array}$$

$$\mathbf{A} = \mathbf{Softmax}(R') = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} r_{00} & r_{01} & r_{02} & 0 \\ r_{10} & r_{11} & 0 & 0 \\ r_{20} & 0 & r_{22} & r_{23} \\ 0 & 0 & r_{32} & r_{33} \end{bmatrix} \end{array}$$

Message Passing Mask



$$\mathbf{A} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} r_{00} & r_{01} & r_{02} & 0 \\ r_{10} & r_{11} & 0 & 0 \\ r_{20} & 0 & r_{22} & r_{23} \\ 0 & 0 & r_{32} & r_{33} \end{bmatrix} \end{array}$$

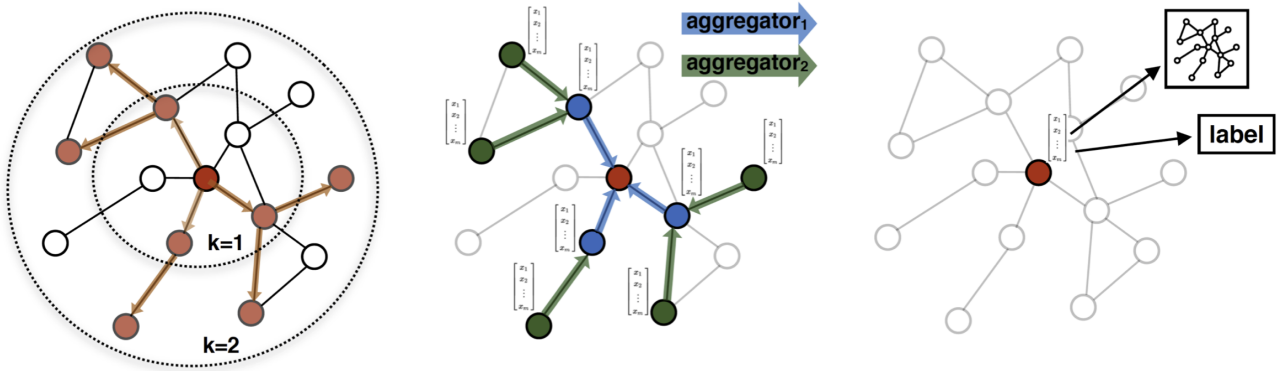
Only 1st order neighbours will send messages.

GraphSAGE

Inductive Representation Learning on Large Graphs

arXiv:1706.02216 [cs.SI]

GraphSAGE: SAmple and aggreGatE



1. Sample neighborhood

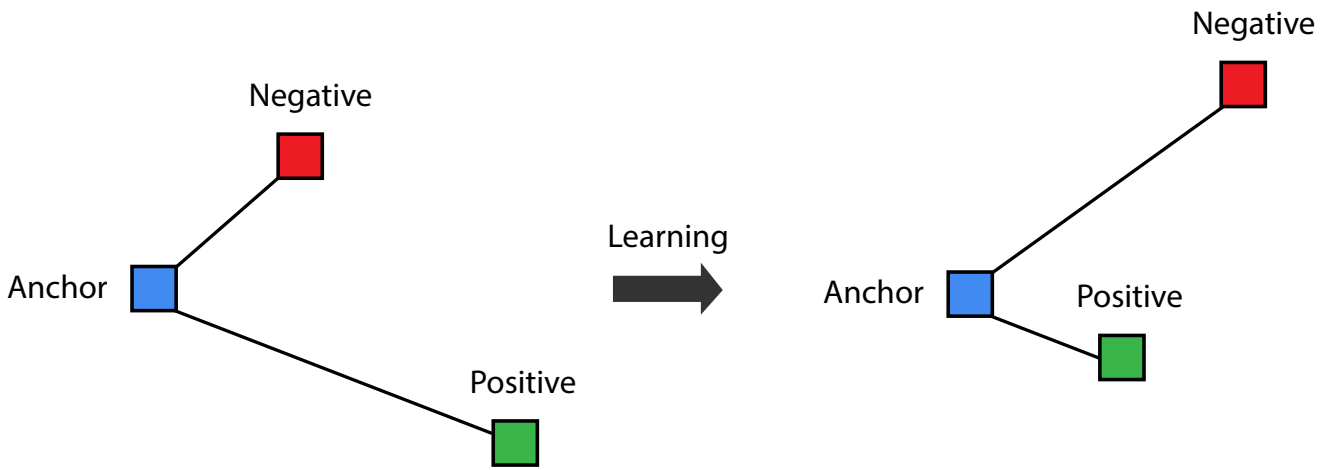
2. Aggregate feature information from neighbors

3. Predict graph context and label using aggregated information

- Neighborhood sampling allows *mini-batch* training on large-scale graphs.
- Pytorch Geometric: NeighborSampler class.

Triplet Loss

- FaceNet: A Unified Embedding for Face Recognition and Clustering
  arXiv:1503.03832 [cs.CV]
- Minimize the distance between an anchor and a positive.
- Maximize the distance between the anchor and a negative.

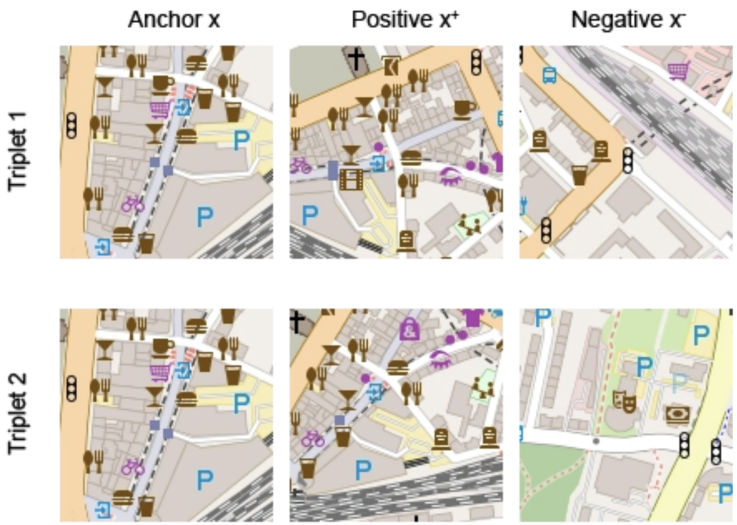


- Example: loc2vec

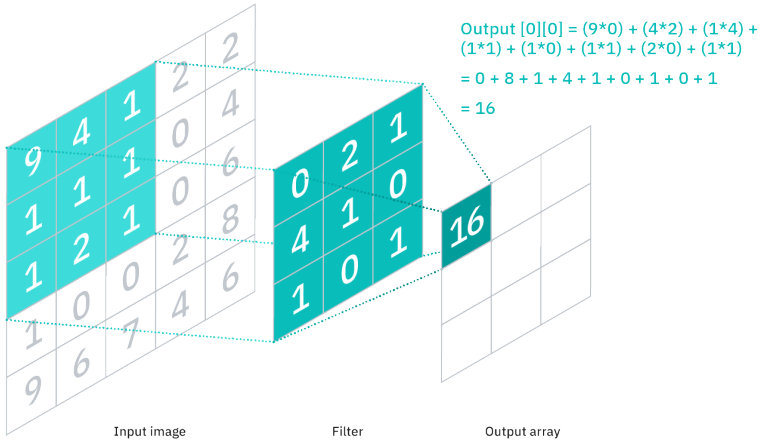
Location Embeddings

loc2vec

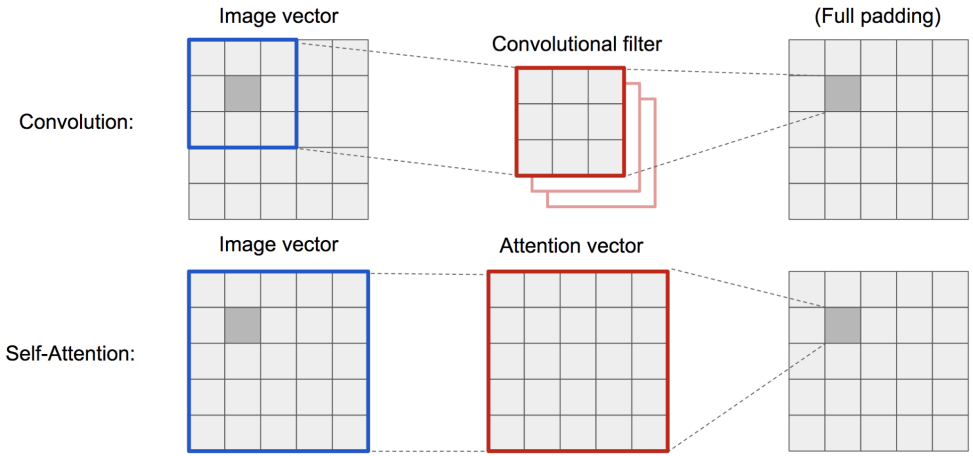
- N-channel tensor represents the area. Channels contain road network, points of interest, etc.
- Feature extractor: Convolutional neural network CNN



Convolutions



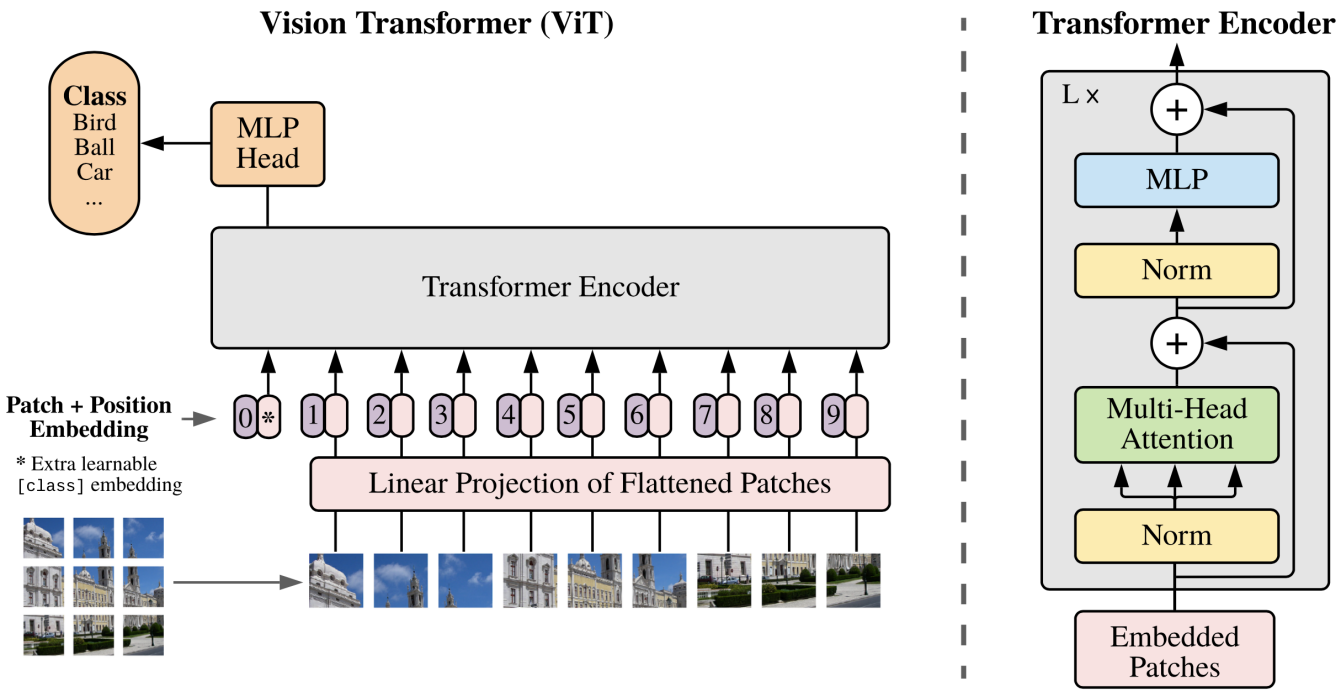
source

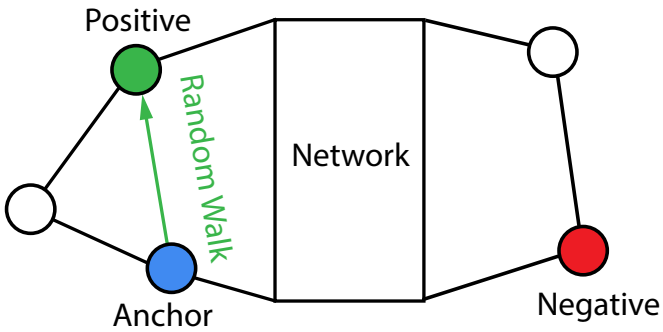


source

Vision Transformer

An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale
arXiv:2010.11929 [cs.CV]

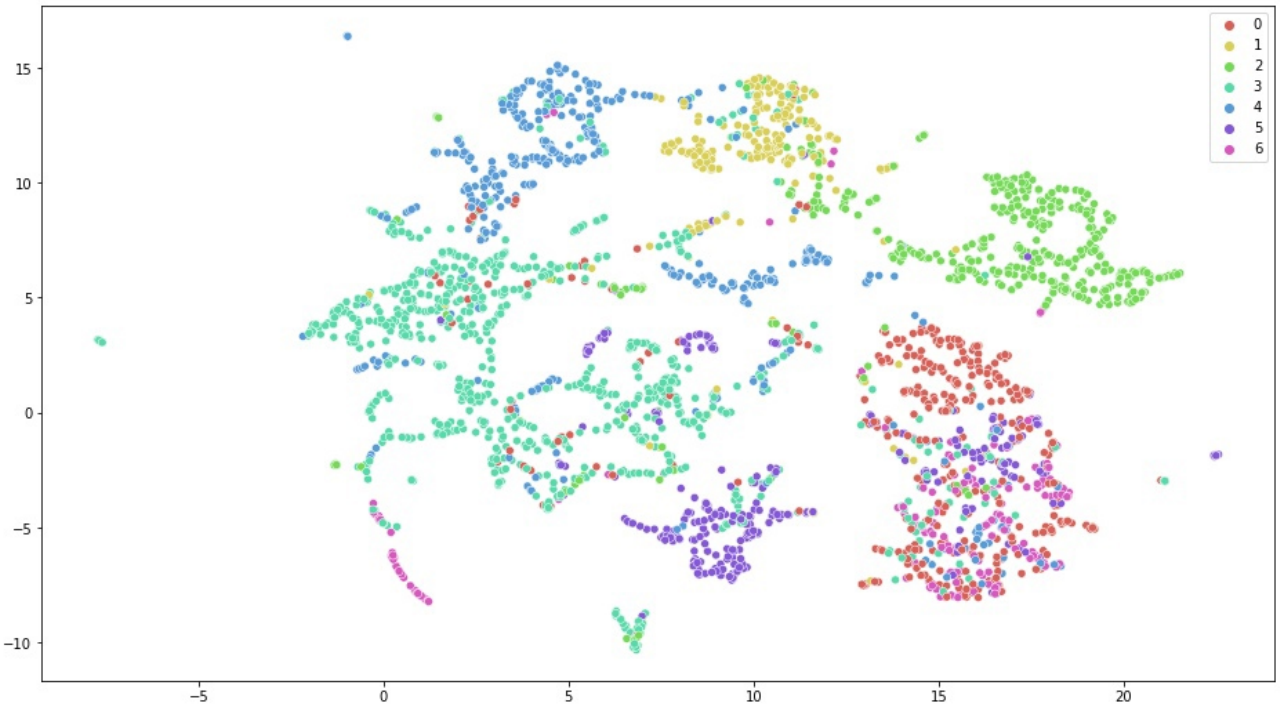




- Nearby nodes - similar representations
- Disparate nodes - distinct representations.

$$\mathcal{J}_G(\mathbf{z}_u) = -\log(\sigma(\mathbf{z}_u{}^T \mathbf{z}_v)) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log(\sigma(-\mathbf{z}_u{}^T \mathbf{z}_{v_n}))$$

2D visualization of CORA dataset unsupervised embeddings (GraphSAGE). Dimensional reduction was obtained by using UMAP.



The End.