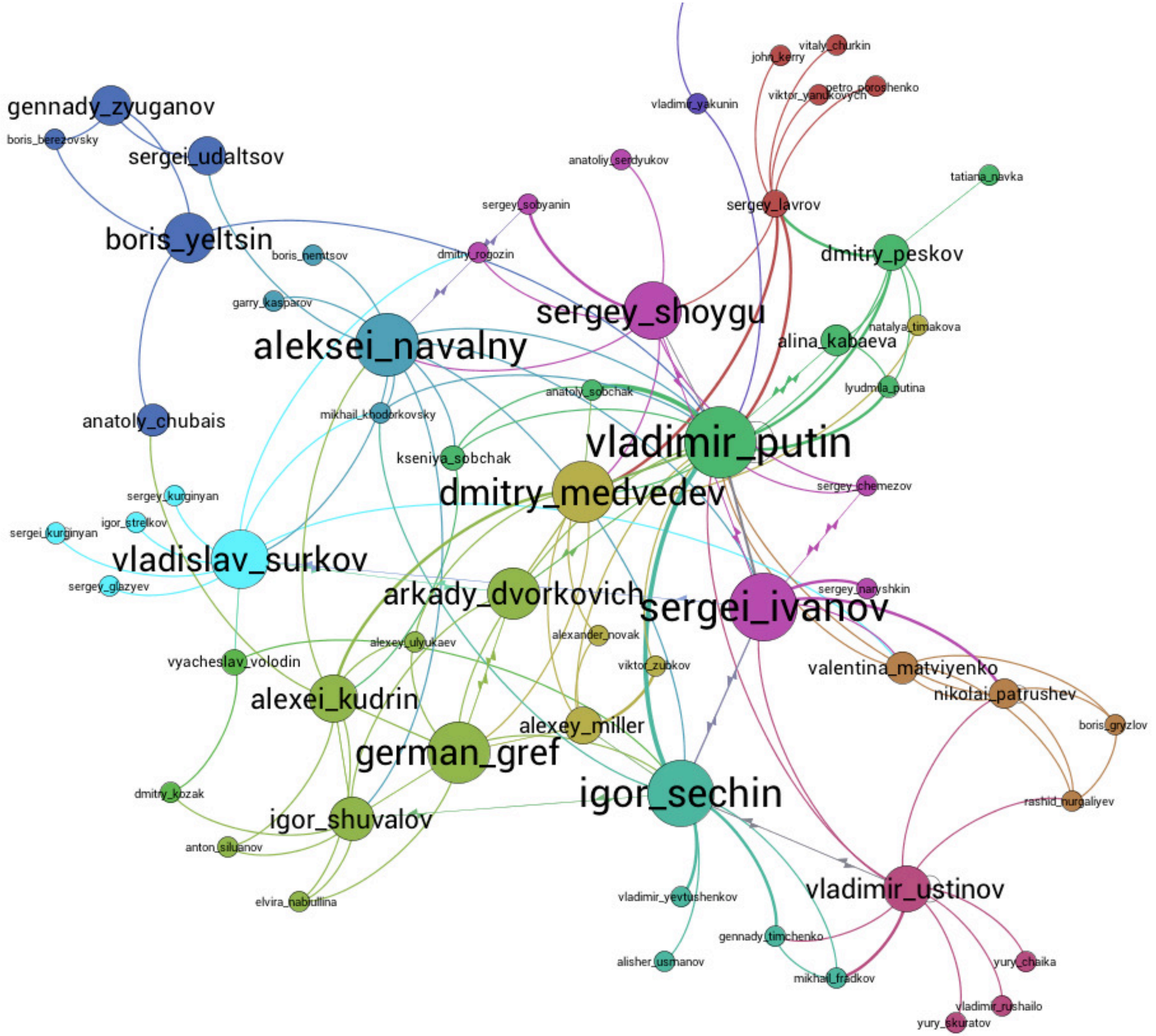
Graphs



gennady_zyuganov
boris_berezovsky
sergei_udaltsov
boris_yeltsin
boris_nemtsov
garry_kasparov
aleksei_navalny
anatoly_chubais
mikhail_khodorkovsky
kseniya_sobchak
sergey_kurginyan
igor_strelkov
sergei_kurginyan
vladislav_surkov
sergey_glazyev
vyacheslav_volodin
alexey_ulyukaev
arkady_dvorkovich
alexei_kudrin
german_gref
dmitry_kozak
igor_shuvalov
anton_siluanov
elvira_nabiullina
vladimir_yakunin
anatoliy_serdyukov
sergey_sobyanin
dmitry_rogozin
sergey_shoygu
john_kerry
vitaly_churkin
viktor_yanukovych
petro_poroshenko
sergey_lavrov
tatiana_navka
dmitry_peskov
natalya_timakova
alina_kabaeva
lyudmila_putina
anatoly_sobchak
vladimir_putin
dmitry_medvedev
sergey_chemezov
sergey_naryshkin
sergei_ivanov
alexander_novak
viktor_zubkov
valentina_matviyenko
nikolai_patrushev
boris_gryzlov
alexey_miller
igor_sechin
rashid_nurgaliyev
vladimir_ustinov
vladimir_yevtushenkov
gennady_timchenko
alisher_usmanov
mikhail_fradkov
yury_chaika
vladimir_rushailo
yury_skuratov

Beyond Sequential Dependency

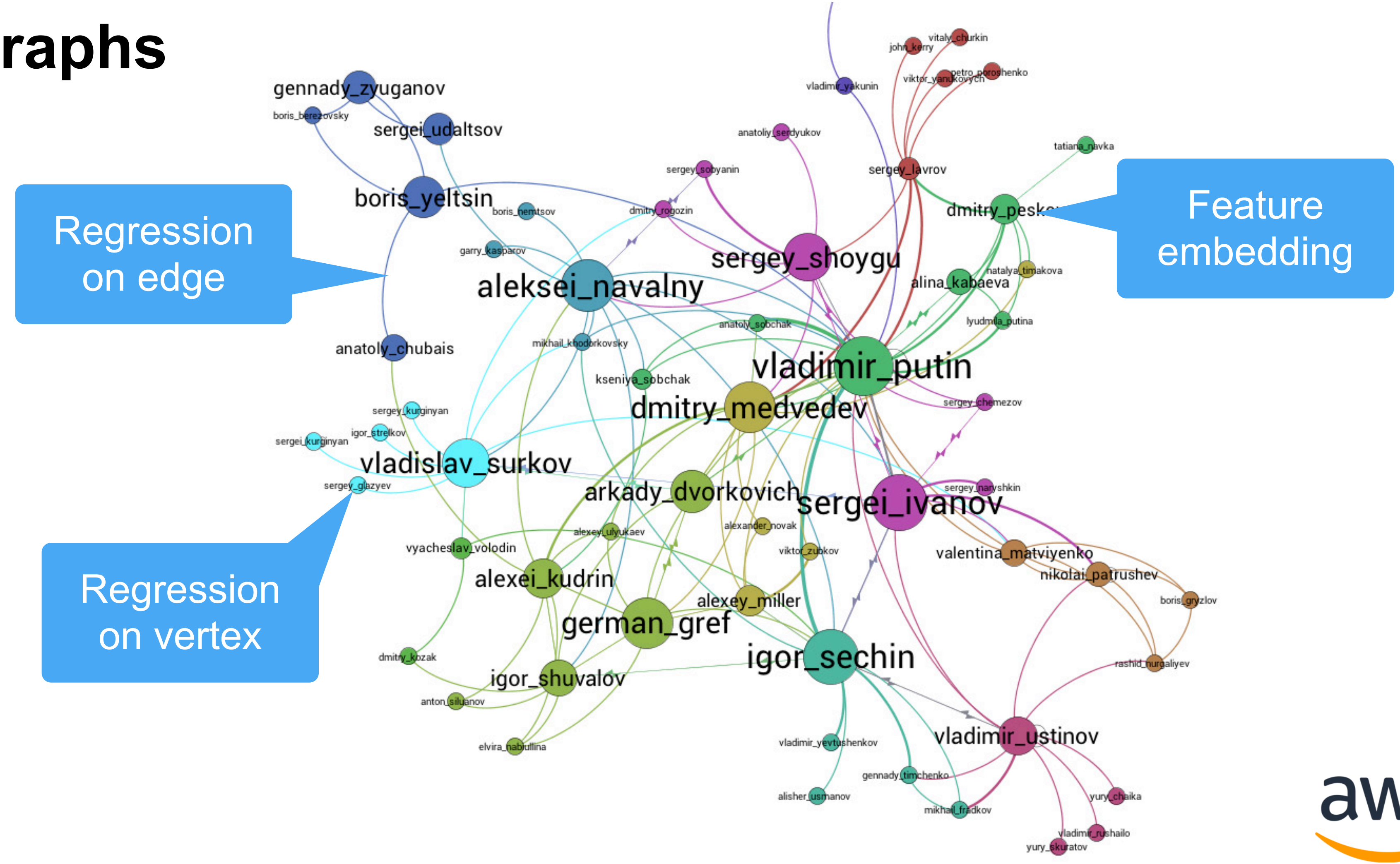
Often difficult to model

- **Spatial dependency** (e.g., road network)
- **General graph dependency**
  - Spread of memes, fake news …
  - Product recommendation (users, items, vendors)
  - Relational databases
- Cannot write as sequence but needs graph. Popular choices:

  Directed graphical model $p(x) = \prod_i p(x_i | x_{\pi(i)})$

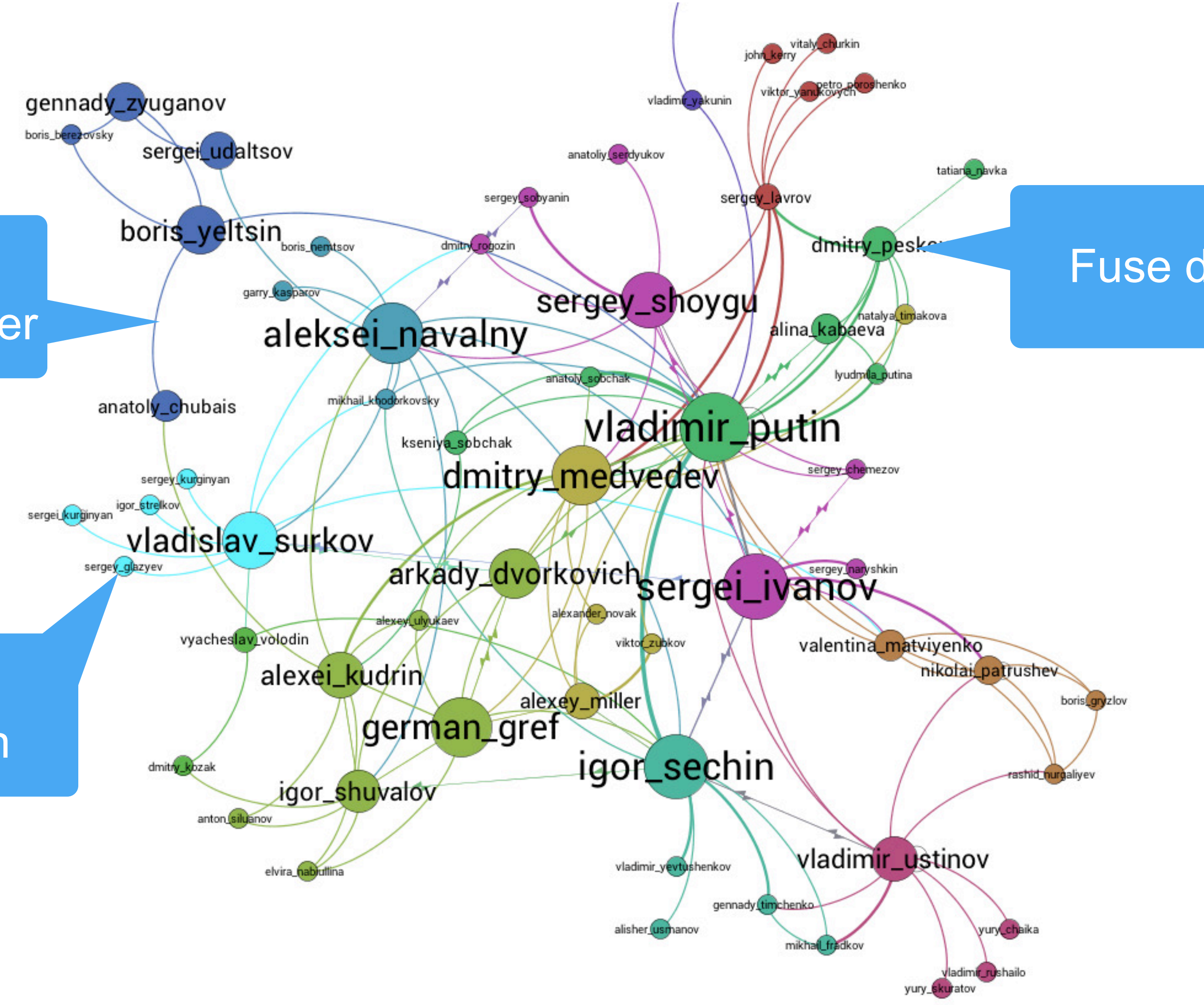
  Undirected graphical model $p(x) = \prod_C \psi_C(x_C)$

Graphs



Regression on edge
Feature embedding
Regression on vertex
gennady_zyuganov
boris_berezovsky
sergei_udaltsov
boris_yeltsin
boris_nemtsov
dmitry_rogozin
garry_kasparov
aleksei_navalny
anatoly_chubais
mikhail_khodorkovsky
kseniya_sobchak
sergey_kurginyan
igor_strelkov
sergei_kurginyan
vladislav_surkov
sergey_glazyev
vyacheslav_volodin
alexei_kudrin
dmitry_kozak
igor_shuvalov
anton_siluanov
elvira_nabiullina
german_gref
arkady_dvorkovich
alexey_ulyukaev
alexander_novak
viktor_zubkov
alexey_miller
igor_sechin
dmitry_medvedev
vladimir_putin
anatoly_sobchak
sergey_shoygu
sergey_sobyanin
anatoliy_serdyukov
vladimir_yakunin
john_kerry
vitaly_churkin
viktor_yanukovych
petro_poroshenko
sergey_lavrov
tatiana_navka
dmitry_peskov
alina_kabaeva
natalya_timakova
lyudmila_putina
sergey_chemezov
sergey_naryshkin
sergei_ivanov
valentina_matviyenko
nikolai_patrushev
boris_gryzlov
rashid_nurgaliyev
vladimir_ustinov
vladimir_yevtushenkov
gennady_timchenko
alisher_usmanov
mikhail_fradkov
yury_chaika
vladimir_rushailo
yury_skuratov

Graphs



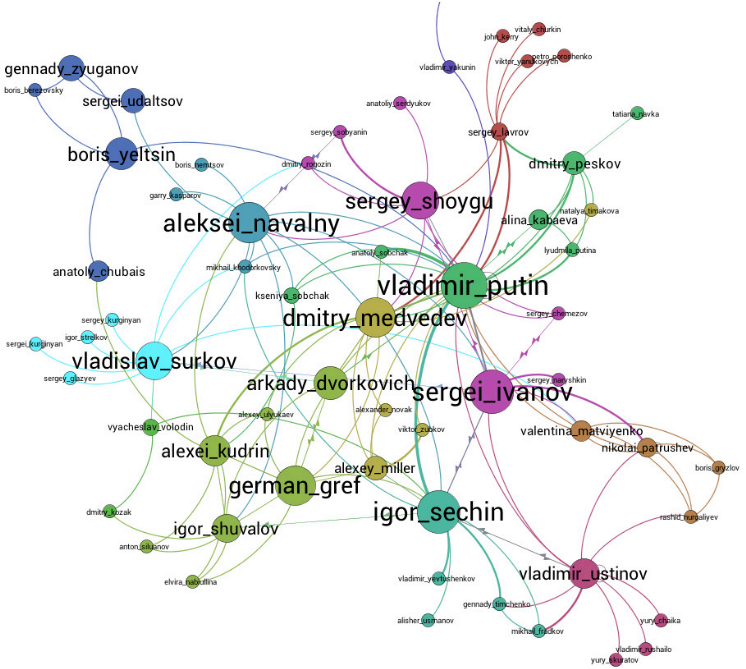
Social Recommender

Fuse data

Fraud detection

Graphs G(V,E)

- Vertices $i \in V$ (with attributes)
- Edges $(i, j) \in E$ (with attributes)
- Estimation problems
  - Given some vertex labels y estimate the remaining ones (e.g. fraud detection)
  - Given some edge attributes e estimate the remaining ones (e.g. link recommendation)



Vertex Updates

Weisfeiler-Lehman algorithm (1976)

- **Key idea**
  - Graph isomorphism … is trivial if vertices are unique
  - Make them unique by repeated hashing

$$v(i) \leftarrow h(v(i), \{v(j) \text{ with } j \sim i\})$$

Local updates

  - Terminate once stationary
- **Machine Learning variant** (Shervashidze & Borgwardt, 2013)
  - The vertex hashes are good features $i \rightarrow \phi(i)$
  - Can prove equivalence to some graph kernels
- Crazy thought … what about vertices with attributes?

aws

Page Rank (Page, Brin, Motwani, Kleinberg, 1990s)

- **Random surfer model with restarts**
  - Start at uniformly random location
  - Follow random link at each vertex
  - Page rank is stationary distribution
- **Self consistency equation**

Random Link following

Random Restart

$$p(i) = \frac{\epsilon}{|V|} + (1 - \epsilon) \sum_{i \sim j} \frac{p(j)}{d(j)} \iff r(i) = \epsilon + (1 - \epsilon) \sum_{i \sim j} \frac{r(j)}{d(j)}$$

- Solve by iterating fixed number of times or until convergence

Local updates

(naive) PageRank Implementation in DGL

```
import dgl.function as fn

def pagerank_builtin(g):
    g.ndata['pv'] = g.ndata['pv'] / g.ndata['deg']
    g.update_all(message_func=fn.copy_src(src='pv', out='m'),
                 reduce_func=fn.sum(msg='m',out='m_sum'))
    g.ndata['pv'] = (1 - DAMP) / N + DAMP * g.ndata['m_sum']
```

Incoming PageRank

Random surfer

Random surfer

Belief Propagation / Message Passing

- **Graphical models** (we operate on clique graph)
- Vertex potential and directed messages

$$\phi_C = \psi_C \prod_{D \sim C} \mu_{D \to C}$$

$$\mu_{D \to C}(x_{C \cap D}) = \sum_{x_{C \setminus D}} \phi_C(x_C)\, \mu^{-1}_{D \to C}(x_{C \cap D})$$

- Finite time convergence if clique graph is a junction tree
- In practice ignore and run *for some time* on clique graph (loopy belief propagation)

Belief Propagation / Message Passing

- **Graphical models** (we operate on clique graph)
- Vertex potential and directed messages

$$\phi_C = \psi_C \prod_{D \sim C} \mu_{D \to C}$$

$$\mu_{D \to C}(x_{C \cap D}) = \sum_{x_{C \setminus D}} \phi_C(x_C)\, \mu^{-1}_{D \to C}(x_{C \cap D})$$

- Crazy thought … what if we didn’t care about graphical models (they might not converge anyway)?

This looks like vertex updates
Can we learn them?
Can we get vertex features?

Graph Convolutions (e.g. Kipf & Welling, 2016)

- **Basic idea**
  - Vertex (and edge) features x_i and x_{ij}
  - Compute new vertex (and edge) features using local update function

$$v_i \leftarrow f(v_i, \{v_j, v_{ij} \text{ with } i \sim j\})$$

$$v_{ij} \leftarrow g(v_{ij}, v_i, v_j)$$

Optional (e.g. not in K&W'16)

- **Variants**
  - Run for a fixed number of steps (like graph kernel)
  - Run to convergence (like page rank)

Forward model

```
G.ndata['feat'] = torch.eye(34)

def gcn_message(edges):
    # The argument is a batch of edges using source node’s feature ‘h’
    return {'msg' : edges.src['h']}

def gcn_reduce(nodes):
    # The argument is a batch of nodes with features summed over ‘msg’
    return {'h' : torch.sum(nodes.mailbox['msg'], dim=1)}

class GCNLayer(nn.Module):
    def __init__(self, in_feats, out_feats):
        super(GCNLayer, self).__init__()
        self.linear = nn.Linear(in_feats, out_feats)

    def forward(self, g, inputs):
        g.ndata['h'] = inputs
        g.send(g.edges(), gcn_message) # trigger message passing
        g.recv(g.nodes(), gcn_reduce)  # trigger aggregation at all nodes
        h = g.ndata.pop(‘h')           # get the result node features
        return self.linear(h)          # perform linear transformation
```

Forward Model

```
class GCN(nn.Module):
    def __init__(self, in_feats, hidden_size, num_classes):
        super(GCN, self).__init__()
        self.gcn1 = GCNLayer(in_feats, hidden_size)
        self.gcn2 = GCNLayer(hidden_size, num_classes)

    def forward(self, g, inputs):
        h = self.gcn1(g, inputs)
        h = torch.relu(h)
        h = self.gcn2(g, h)
        return h
# First layer  - 34 inputs to 5 dimensions
# Second layer - 2 dimensional embedding for 2 groups
net = GCN(34, 5, 2)
```

Training it

```
optimizer = torch.optim.Adam(net.parameters(), lr=0.01)
all_logits = []
for epoch in range(30):
    logits = net(G, inputs)
    # save the logits for visualization later
    all_logits.append(logits.detach())
    logp = F.log_softmax(logits, 1)
    # only compute loss for labeled nodes
    loss = F.nll_loss(logp[labeled_nodes], labels)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Graph Convolutions with state (GeniePath - Liu et al., 2018, Platanios & S, 2018)

- **Basic idea**
  - Vertex (and edge) features x_i and x_{ij}
  - Vertex has internal state, LSTM or similar (GeniePath)
  - Compute new vertex (and edge) features using local update function

$$(v_i^{t+1}, h_i^{t+1}, c_i^{t+1}) \leftarrow \mathrm{LSTM}(v_i^t, h_i^t, c_i^t, \{v_j, v_{ij} \text{ with } i \sim j\})$$

$$v_{ij} \leftarrow g(v_{ij}, v_i, v_j)$$

Optional

  - This works better on some datasets

Graph Convolutions with state (GeniePath - Liu et al., 2018, Platanios & S, 2018)

- **Basic idea**
  - Vertex (and edge) features x_i and x_{ij}
  - Vertex has internal state, LSTM or similar (GeniePath)
- **Minor twist (Deep Sets - Zaheer et al., 2017)**
  All functions on sets are nonlinear sums

$$(v_i^{t+1}, h_i^{t+1}, c_i^{t+1}) \leftarrow \mathrm{LSTM}\left(v_i^t, h_i^t, c_i^t, g\Big(\sum_{j \sim i} f(v_j, v_{ij})\Big)\right)$$

$$v_{ij} \leftarrow g(v_{ij}, v_i, v_j)$$

Graph Attention Networks (Velickovic et al., 2017)

- **Basic idea**
  - Aggregation from neighboring vertices should be attention gated (self attention strategy)
  - Without attention

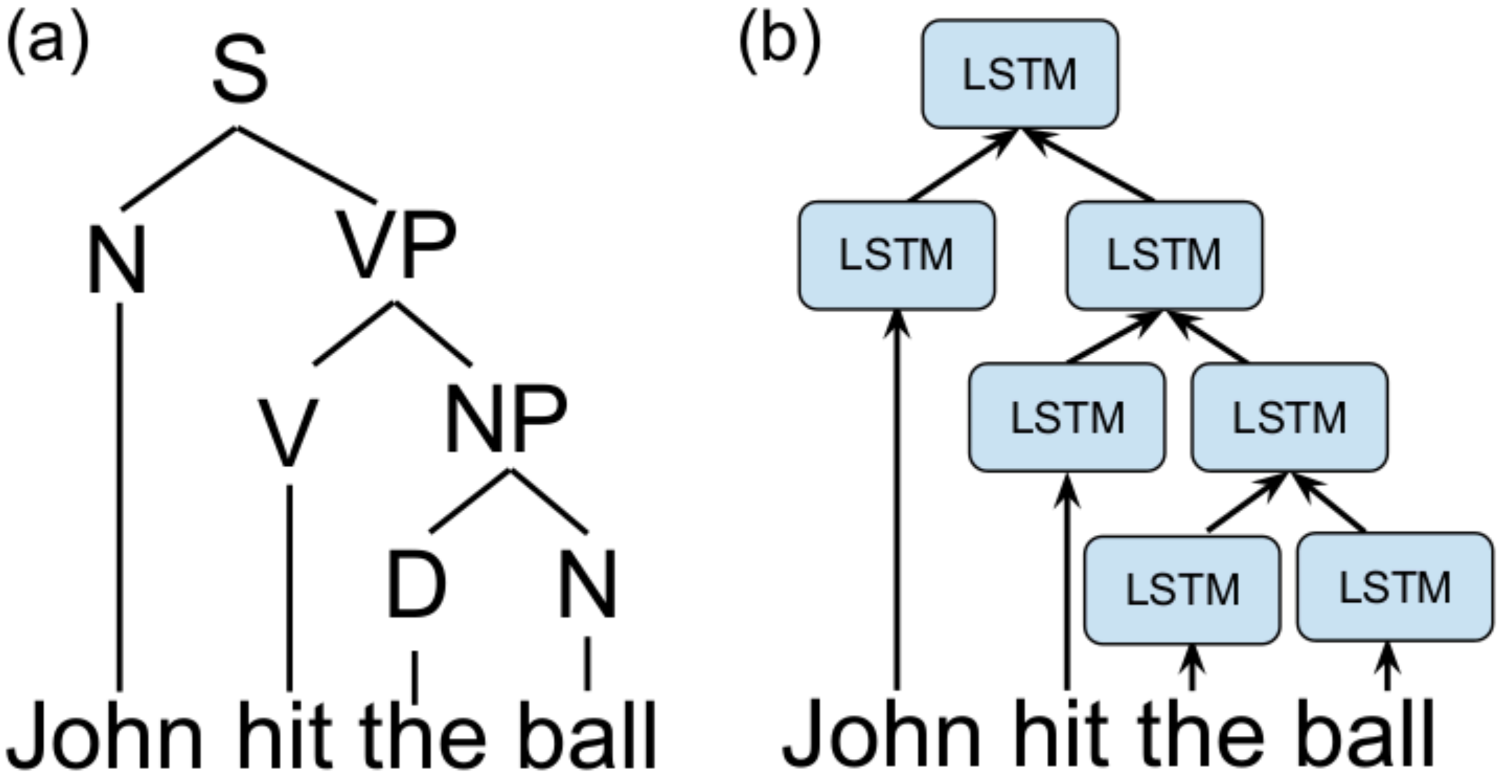
$$v_i \leftarrow f\left(v_i, \sum_{i \sim j} h(v_j)\right)$$

  - With attention

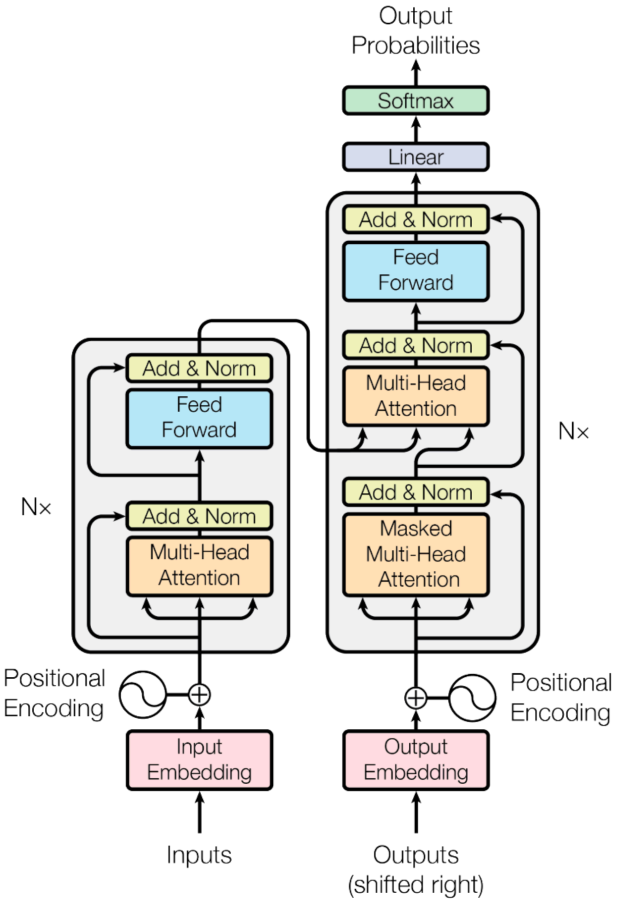
$$v_i \leftarrow f\left(v_i, \sum_{i \sim j} \alpha_{ij} h(v_j)\right) \text{ where } \alpha_{ij} = \mathrm{softmax}\left(\{v_i^\top M v_j\}\right)$$

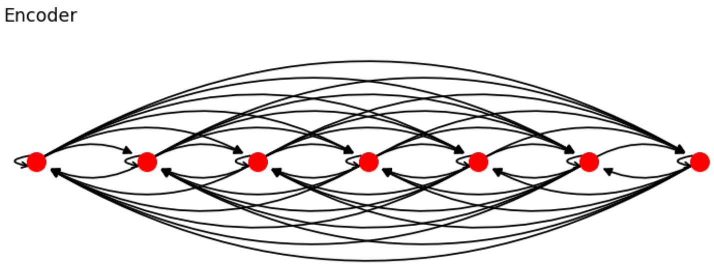
  - This works better on some datasets

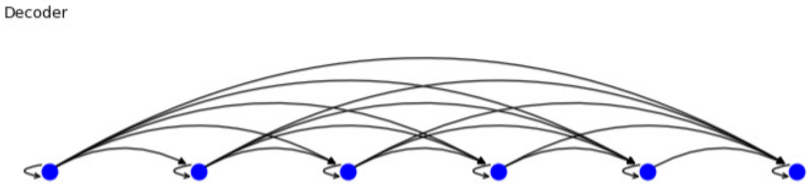
Tree-LSTM as message-passing

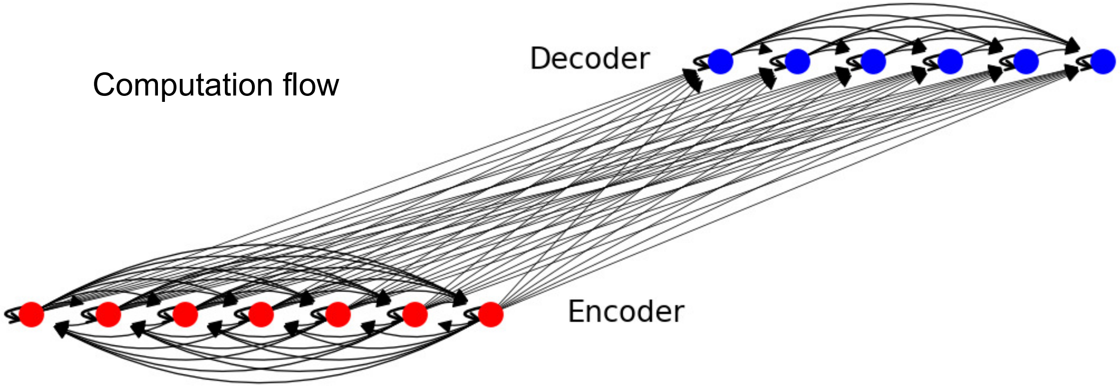


Deconstruct Transformer as a graph

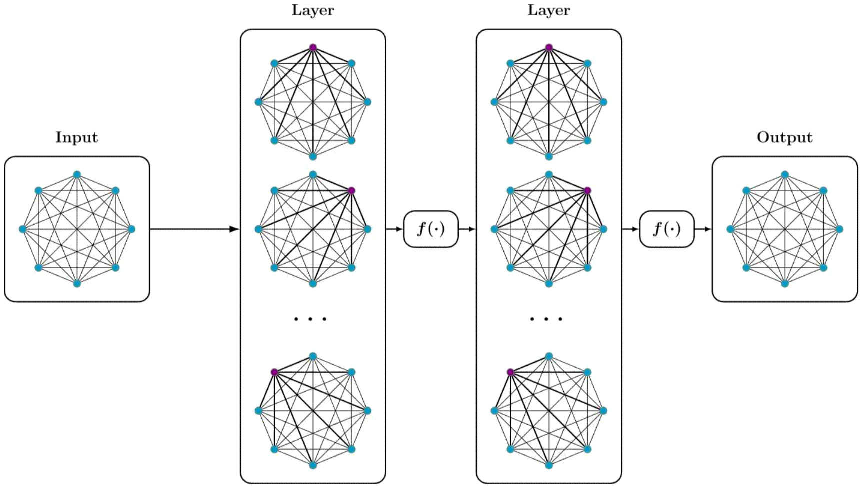


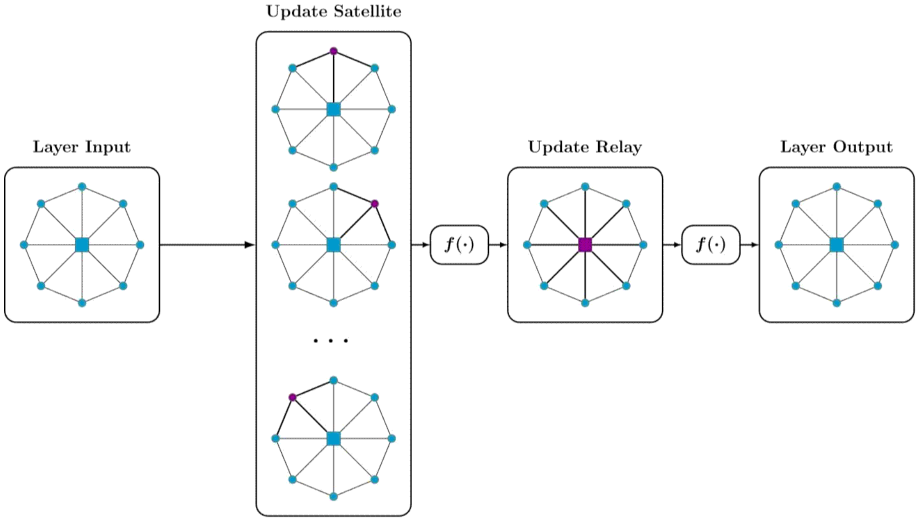






Deconstruct Transformer as a graph





Transformer:
- data & compute hungry

Star-Transformer (in NAACL’19)
- much less data hungry
- leverages ngram prior,
- has issues with long dependency

SegTree-Transformer (in ICLR’19 RLGM)
- less data hungry
- A good compromise in between

Making it Practical (Dai et al., 2018)

- Learning the vertex update function is **expensive**
  - Backprop over graph has to deal with entire graph quickly (6 degrees of separation kill BPTT)
  - Not much benefit in finite iterations
- Replace with fixed point iteration
  - Can learn it directly
  - Local convergence
  - **Sample vertex updates** (much smaller subset)
- This works better on some datasets

Making it Practical (Dai et al., 2018)

- Initialize
- Compute update

  $$v_i \leftarrow f(v_i, \{v_j \text{ with } i \sim j\})$$

- Compute (regression) loss from embedding
- Backprop to change
  - f via loss
  - f via self consistency