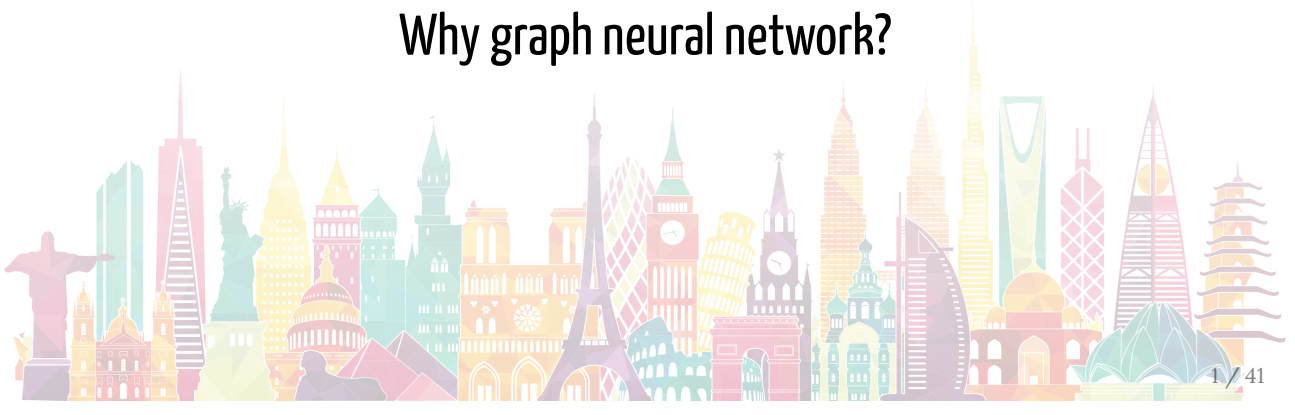
GeometricFlux.jl: Geometric Deep Learning on Flux

Yueh-Hua Tu @ JuliaCon 2020

Why graph neural network?



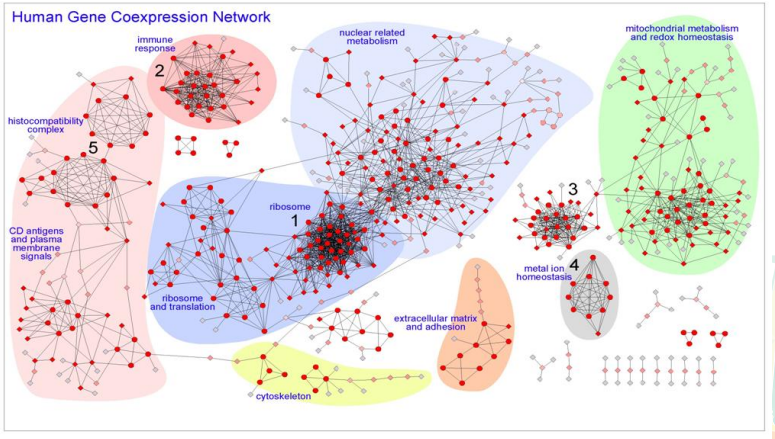
Nowadays AI utilizes vector/matrix to represent data

- Computer vision
- Natural language processing
- Speech
- Biomedical signals

Many scientific data use graph structure

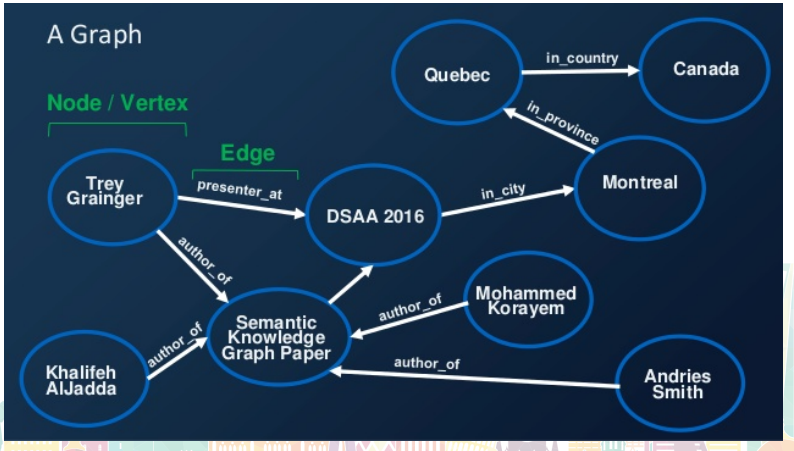
- Social networks in socail science
- Traffic networks
- Gene regulatory networks in biology
- Molecular structure in chemistry
- Knowledge graph
- 3D object surfaces in computer graphics

Human gene coexpression network



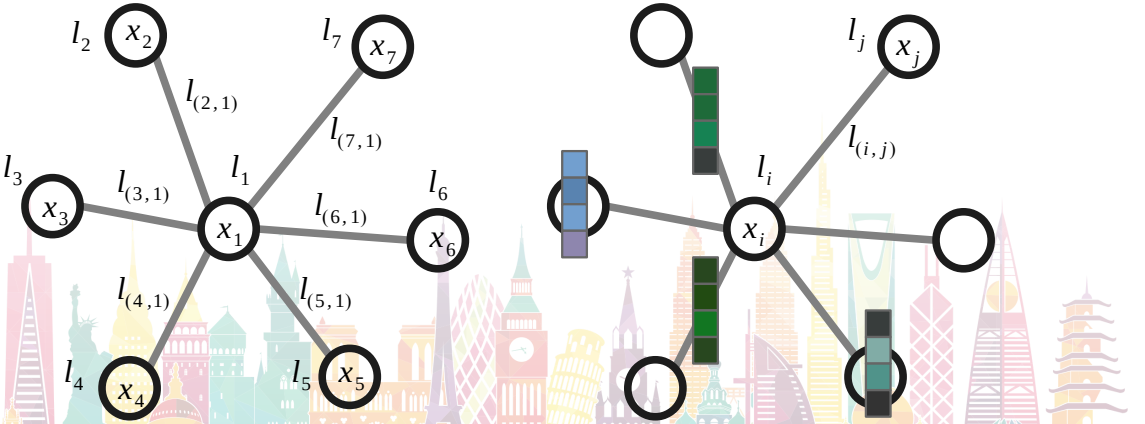
picture source

Knowledge graph

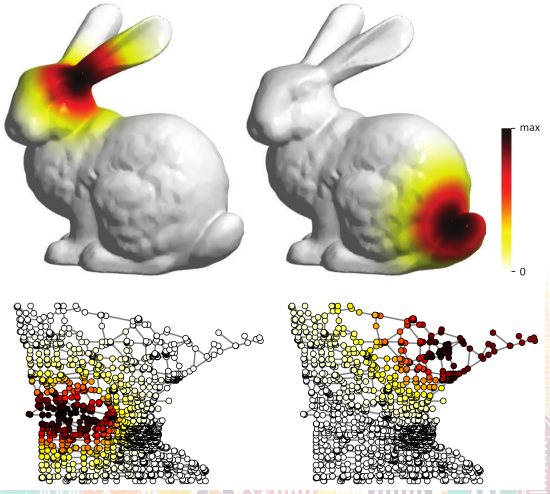


picture source

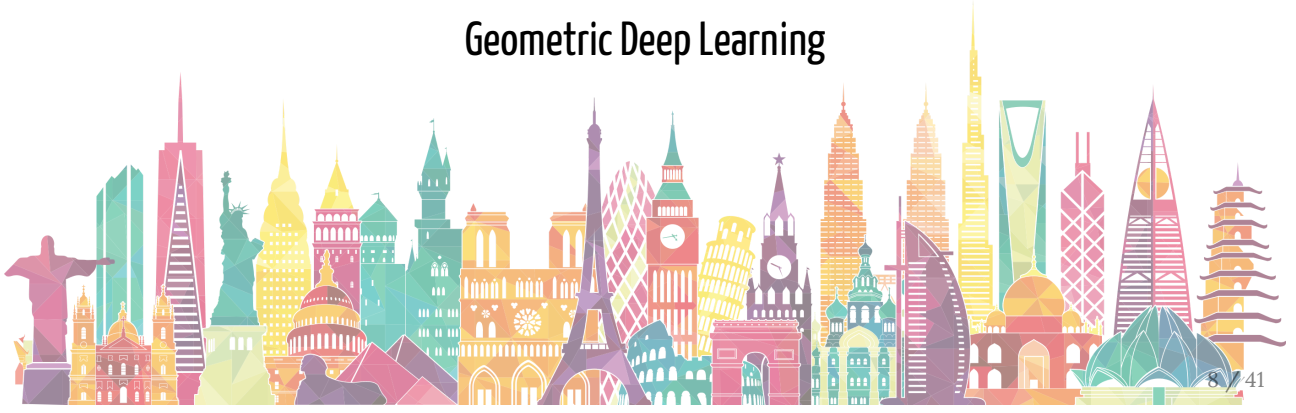
Graphs are more representative than vectors



Graphs are discrete approximation to manifold

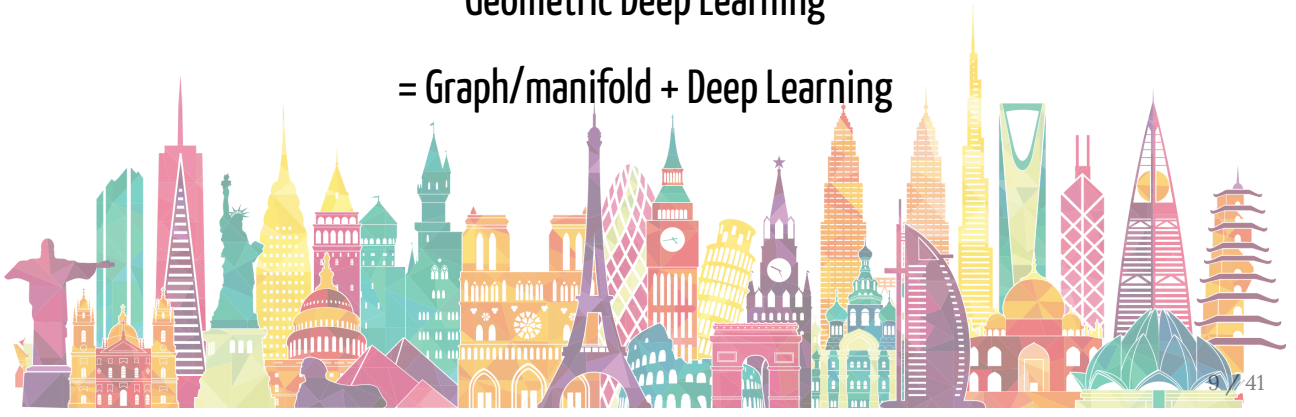


Geometric Deep Learning



Geometric Deep Learning

= Graph/manifold + Deep Learning

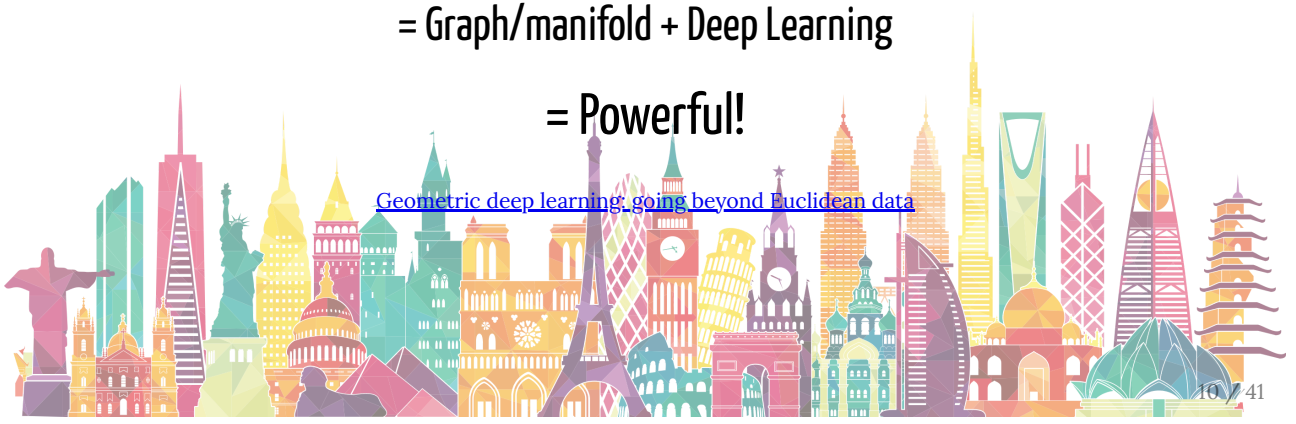


Geometric Deep Learning

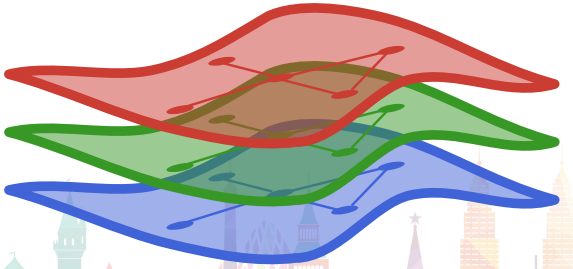
= Graph/manifold + Deep Learning

= Powerful!

Geometric deep learning: going beyond Euclidean data



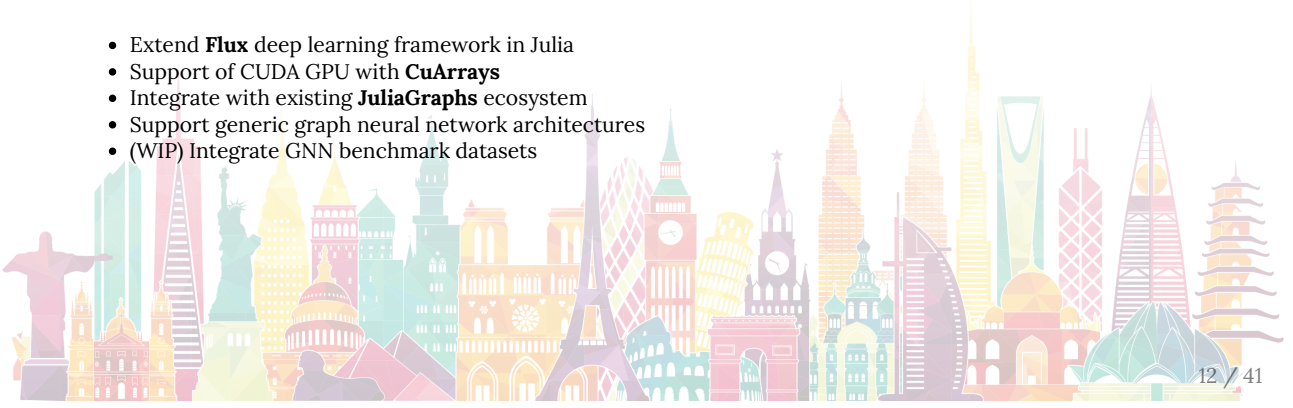
GeometricFlux.jl



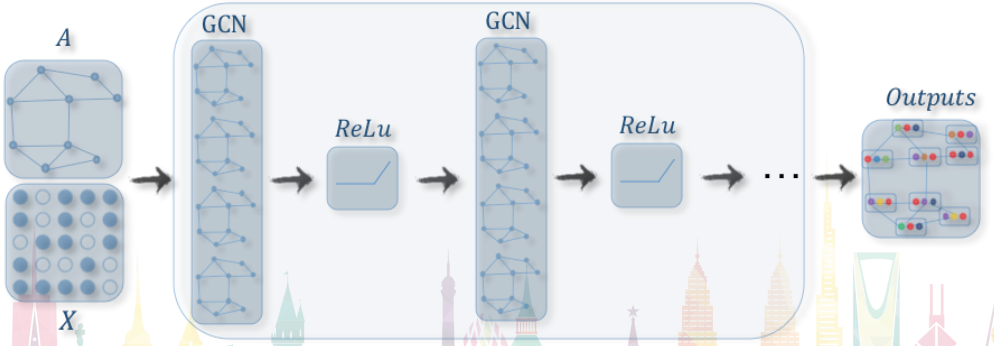
https://github.com/yuehhua/GeometricFlux.jl

Features

- Extend **Flux** deep learning framework in Julia
- Support of CUDA GPU with **CuArrays**
- Integrate with existing **JuliaGraphs** ecosystem
- Support generic graph neural network architectures
- (WIP) Integrate GNN benchmark datasets



Graph convolutional layers




```
GCNConv(adj_mat, num_features=>hidden, relu)
```

A Comprehensive Survey on Graph Neural Networks

Use it as you use Flux


```
model = Chain(GCNConv(g, 5120=>1000, relu),
              GCNConv(g, 1000=>500, relu),
              softmax)

## Loss
loss(x, y) = crossentropy(model(x), y)
accuracy(x, y) = mean(onecold(model(x)) .== onecold(y))

## Training
ps = Flux.params(model)
train_data = [(train_X, train_y)]
opt = ADAM(0.01)
evalcb() = @show(accuracy(train_X, train_y))

Flux.train!(loss, ps, train_data, opt, cb=throttle(evalcb, 10))
```

Compatible with layers in Flux


```
## Model
model = Chain(GCNConv(g, 5120=>1000, relu),
              Dropout(0.5),
              GCNConv(g, 1000=>500, relu),
              Dense(500, 7),
              softmax)
```

Layers

Convolution layers

- GCNConv
- GraphConv
- ChebConv
- GatedGraphConv
- GATConv
- EdgeConv

Layers

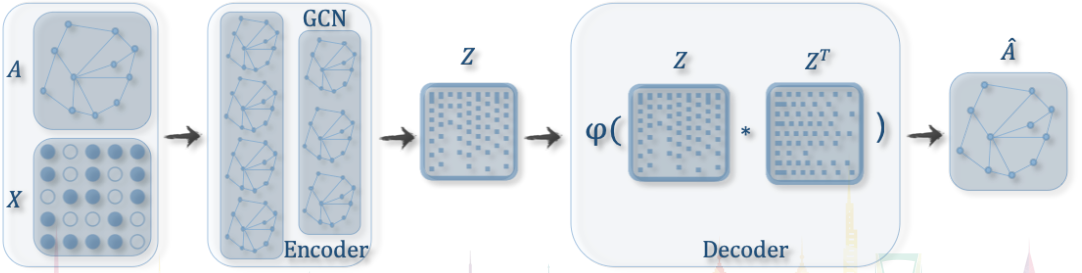
Convolution layers

- GCNConv
- GraphConv
- ChebConv
- GatedGraphConv
- GATConv
- EdgeConv

Pooling layers

- GlobalPool
- TopKPool
- MaxPool
- MeanPool
- sum/sub/prod/div/max/min/mean pool

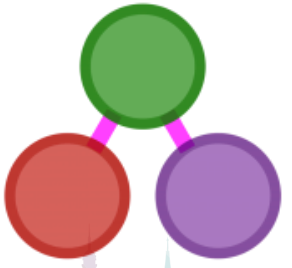
Special layers for autoencoders




```
GAE(enc, InnerProductDecoder(σ))
VGAE(VariationalEncoder(enc, h_dim, z_dim), InnerProductDecoder(σ))
```

- VariationalEncoder (WIP)
- InnerProductDecoder

Construct layers from SimpleGraph/SimpleWeightedGraph



It allows you to cache a *static* graph in layer for computational efficiency.

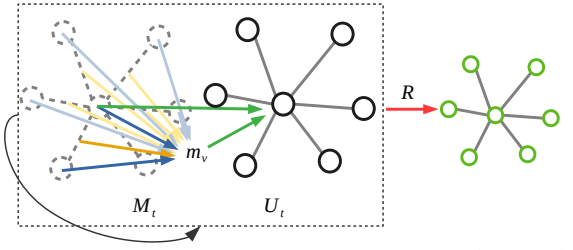
```
g = SimpleGraph(5)
add_edge!(1, 2); add_edge!(3, 4)

GCNConv(g, num_features=>1000, relu)
```

AD-ready with Zygote

GeometricFlux support AD with Zygote on CPU and GPU!

Use message passing scheme



$$m_v^{t+1} = \sum_{u \in N(v)} M_t(h_v^t, h_u^t, e_{uv})$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1})$$

- h_v^t: vertex features
- e_{uv}: edge features
- $N(v)$: neighbors of vertex v

Message passing scheme API

Define your message-passing layer

```
struct MyMessagePassingLayer <: MessagePassing
    ...
end
```

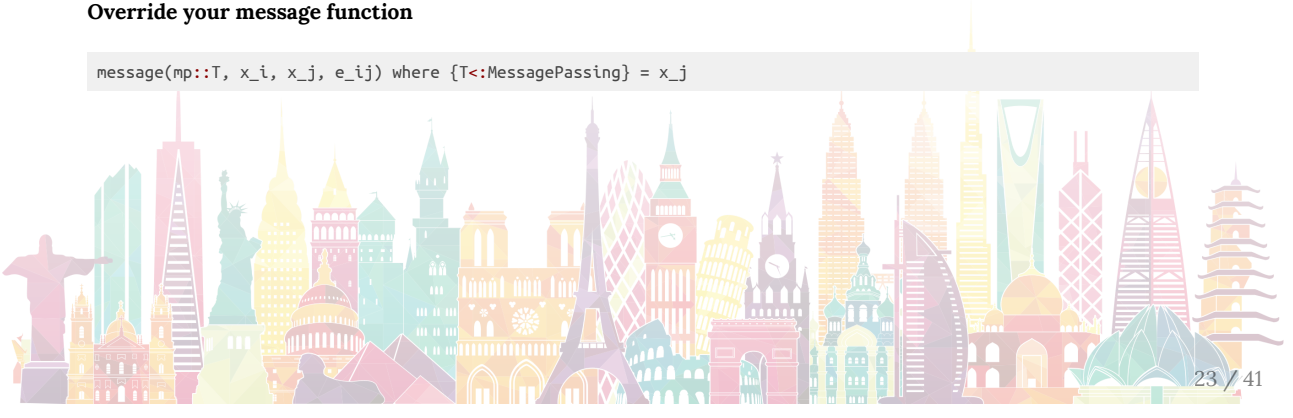
Message passing scheme API

Define your message-passing layer

```
struct MyMessagePassingLayer <: MessagePassing
    ...
end
```

Override your message function

```
message(mp::T, x_i, x_j, e_ij) where {T<:MessagePassing} = x_j
```



Message passing scheme API

Define your message-passing layer

```
struct MyMessagePassingLayer <: MessagePassing
    ...
end
```

Override your message function

```
message(mp::T, x_i, x_j, e_ij) where {T<:MessagePassing} = x_j
```

Choose your aggregation function

```
aggregate_neighbors(mp::T, aggr::Symbol, ...) where {T<:MessagePassing}
```

Message passing scheme API

Define your message-passing layer

```
struct MyMessagePassingLayer <: MessagePassing
    ...
end
```

Override your message function

```
message(mp::T, x_i, x_j, e_ij) where {T<:MessagePassing} = x_j
```

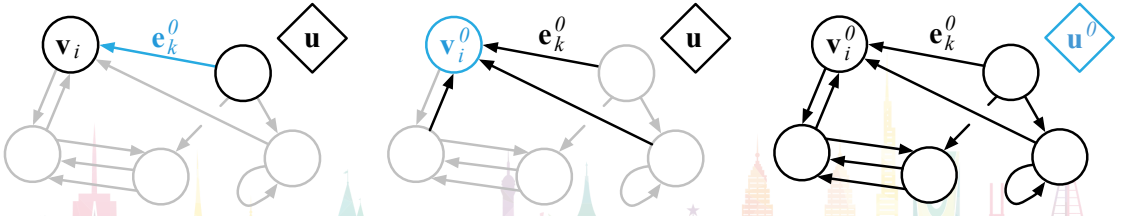
Choose your aggregation function

```
aggregate_neighbors(mp::T, aggr::Symbol, ...) where {T<:MessagePassing}
```

Override your update function

```
update(mp::T, m, x) where {T<:MessagePassing} = m
```

Graph network



(a) Edge update (b) Node update (c) Global update

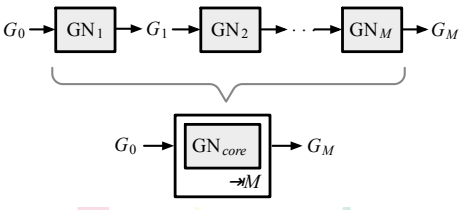
Relational inductive biases, deep learning, and graph networks

Define your own graph network block as layer

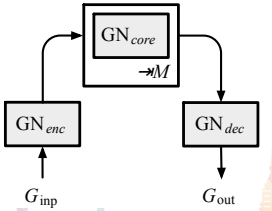
```
struct MyGNBlock <: GraphNet
    ...
end
```

Define your own graph network block as layer

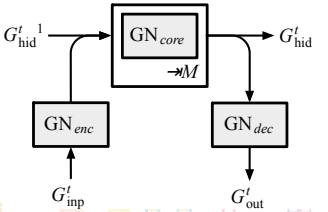
```
struct MyGNBlock <: GraphNet
    ...
end
```



(a) Composition of GN blocks



(b) Encode-process-decode



(c) Recurrent GN architecture

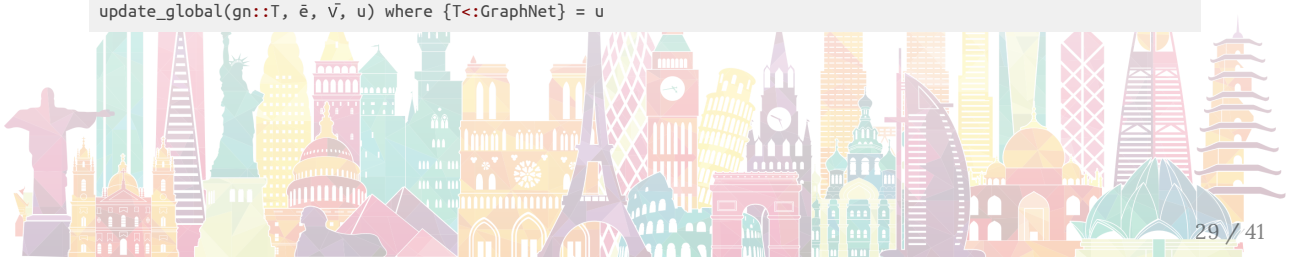
```
Chain(MyGNBlock(...),
      MyGNBlock(...),
      MyGNBlock(...))
```

[Relational inductive biases, deep learning, and graph networks]

Graph network API

Update functions

```
update_edge(gn::T, e, vi, vj, u) where {T<:GraphNet} = e
update_vertex(gn::T, ē, vi, u) where {T<:GraphNet} = vi
update_global(gn::T, ē, v̄, u) where {T<:GraphNet} = u
```



Graph network API

Update functions

```
update_edge(gn::T, e, vi, vj, u) where {T<:GraphNet} = e
update_vertex(gn::T, ē, vi, u) where {T<:GraphNet} = vi
update_global(gn::T, ē, v̄, u) where {T<:GraphNet} = u
```

Aggregate functions

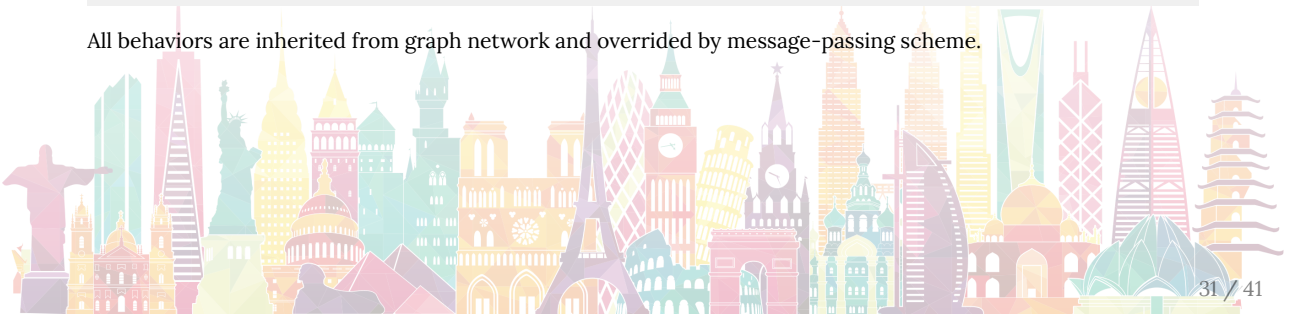
```
aggregate_neighbors(gn::T, aggr::Symbol, ...) where {T<:GraphNet}
aggregate_edges(gn::T, aggr::Symbol, ...) where {T<:GraphNet}
aggregate_vertices(gn::T, aggr::Symbol, ...) where {T<:GraphNet}
```

Feel free to override!

Graph network is compatable with message passing scheme

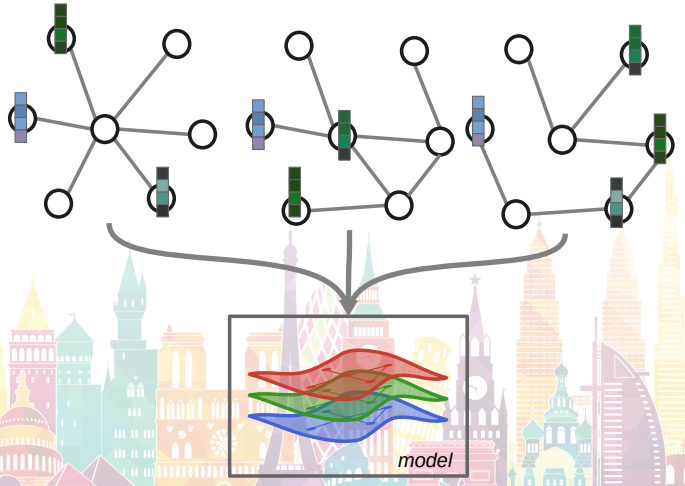
```
abstract type MessagePassing <: GraphNet end
```

All behaviors are inherited from graph network and overrided by message-passing scheme.

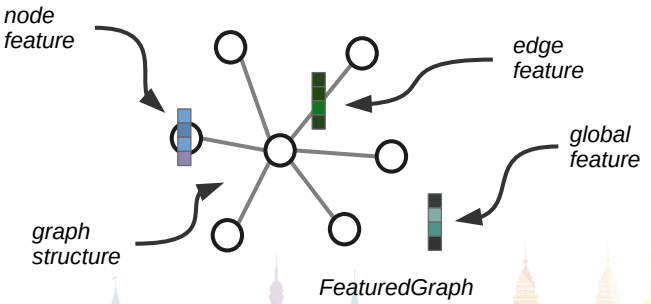


Variable graph input is supported

What if I want to train my model with different graph structure?

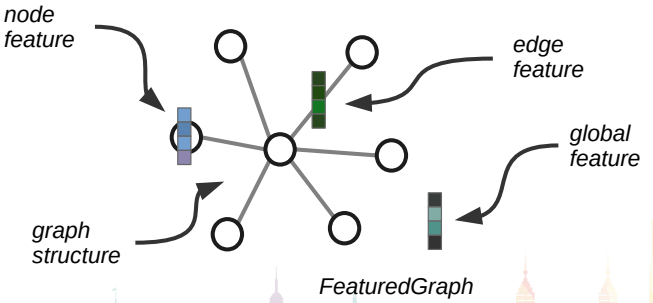


FeaturedGraph goes through the whole network



- Support `GraphNetwork` and `MessagePassing`
- Support linear algebra related calculations
- Integrates well with other GeometricFlux layers

FeaturedGraph goes through the whole network



- Support `GraphNetwork` and `MessagePassing`
- Support linear algebra related calculations
- Integrates well with other GeometricFlux layers

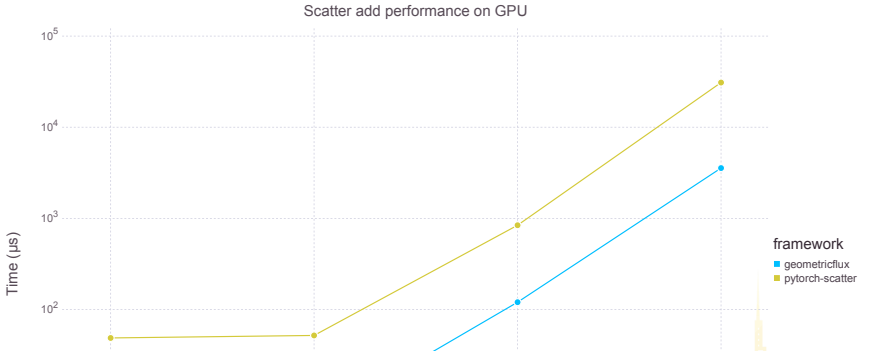
Graph-free layer can be used with FeaturedGraph input

```
GCNConv(num_features=>1000, relu)
```

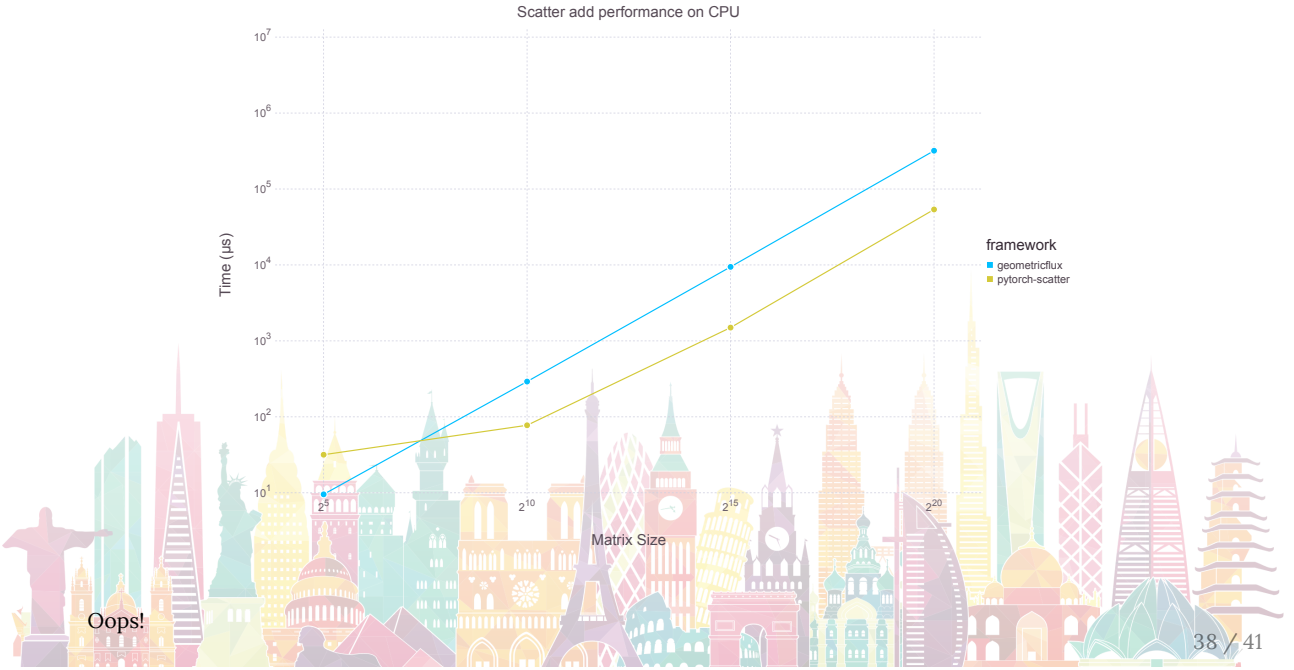
Working examples

- Graph convolution network (GCN)
- Graph autoencoder (GAE)

Performance



Thank you, Tim!


Scatter add performance on CPU
Time (µs)
10^7
10^6
10^5
10^4
10^3
10^2
10^1
2^5
2^{10}
2^{15}
2^{20}
Matrix Size
framework
geometricflux
pytorch-scatter

Oops!

Future work

- Support sparse array computation
  - on CPU and performance optimization
  - on cuda (CUDA/CUSPARSE)
- More layers/models
- Datasets inetegration

Future work

- Support sparse array computation
  - on CPU and performance optimization
  - on cuda (CUDA/CUSPARSE)
- More layers/models
- Datasets inetegration

Near future

- Example models:
  - Graph attention network (GAT)
  - Variational graph autoencoder (VGAE)
  - Neural graph differential equations (GDE)
- Documentation and tutorials
- Increase test cases
- Improve scatter performance on CPU
- Move to FluxML

Thank you for attention

Pull request and issues are welcome.

slides: https://yuehhua.github.io/juliacon2020/geometricflux/

This slides are generated by Remark.jl