Learning to Predict and Improve Build Successes in Package Ecosystems

Harshitha Menon, Todd Gamblin
Lawrence Livermore National Laboratory

BUILD-SI

LLNL-PRES-835342
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC and was supported by LLNL LDRD Program under Project No. 21-SI-005.

Lawrence Livermore National Laboratory

We build codes from hundreds of small, complex pieces

- Component-based software development dates back to the 60's
- M.D. McIlroy, Mass Produced Software Components. NATO SE Conf., 1968
- Pros are well known:
  - Teams can and must reuse each others' work
  - Teams write less code, meet deliverables faster
- Cons:
  - Teams must ensure that components work together
  - Integration burden increases with each additional library
  - Integration must be repeated with each update to components

Developers integrate large software stacks manually

- Simply finding a compatible set of versions for packages is NP-hard (combinatorial)
  - Changing one version may affect all others, those changes may trigger others, and so on
- In HPC, there are many more parameters to adjust
  - Version, compiler, ABI, build options, microarchitecture, GPU capability, etc.
  - Multiple codes in the same environment (workflows), performance
- We solve this problem repeatedly by trial and error
  - Incompatibilities are not known in advance; developers discover them

Transitive dependency requirements can cause cascading errors

```
      umpire                                     umpire
     /       \          upgrade                 /       \
camp@1.0    blt@1.0      --->        camp@1.1    blt@1.1
     \       /                                  \       /
      cmake@3.19                              cmake@3.21.0
```

- blt@1.0 requires cmake >= 3.18, but is incompatible with cmake@3.21.0 due to an unknown bug
- camp@1.0 depends on cmake@3.19 or higher, but camp@1.1 depends on cmake@3.21 or higher
- The umpire developers want to use camp@1.1 for its new features
- Upgrading camp to 1.1 pushes cmake to the latest 3.21.0 will cause the build to fail
- We need to use blt@1.1 to make this work.

Team would have to build several versions of cmake and blt to find a working configuration

BuildCheck

Predict the build outcome of various package configurations with high accuracy.

Graph Neural Networks (GNN) for Build Prediction

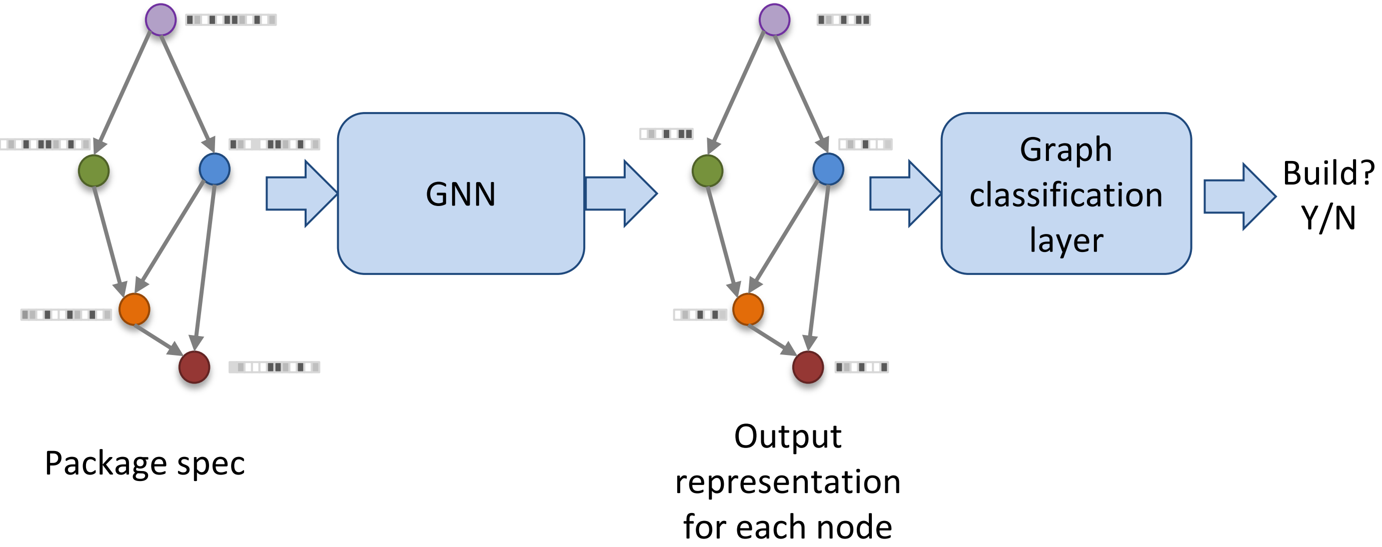
Why GNN?

- GNNs are ideally suited for handling complex dependency relationships in software ecosystems.

Problem Definition

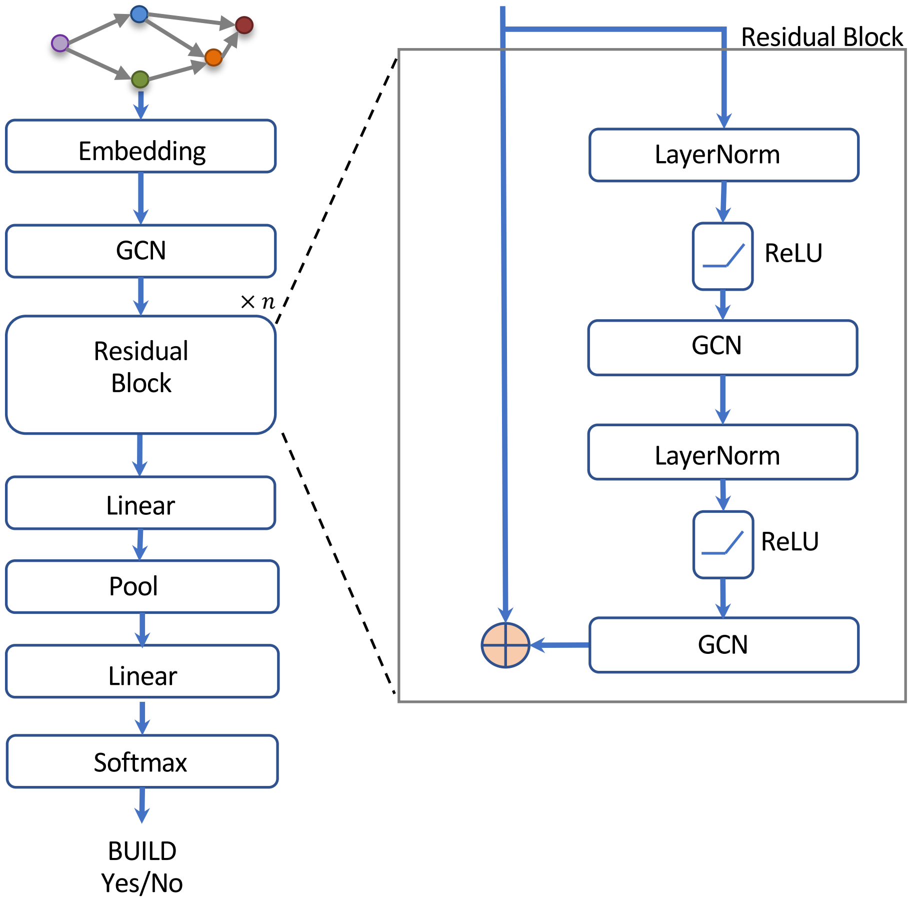
- The package dependency graph is a directed acyclic graph (DAG)
- Graph is represented as $G = (V, E)$, where V is the set of nodes representing the packages and E is the set of edges capturing the dependencies.
- We cast the build success prediction problem as a supervised learning problem
- **Goal**: learn a model to predict the build outcome

Overview of build outcome prediction using GNN



- Each configuration is represented as a graph
- Node features incorporate information about packages (which package and version)
- Layers GCN
- Final layer does a global pooling to predict whether this configuration builds or not.

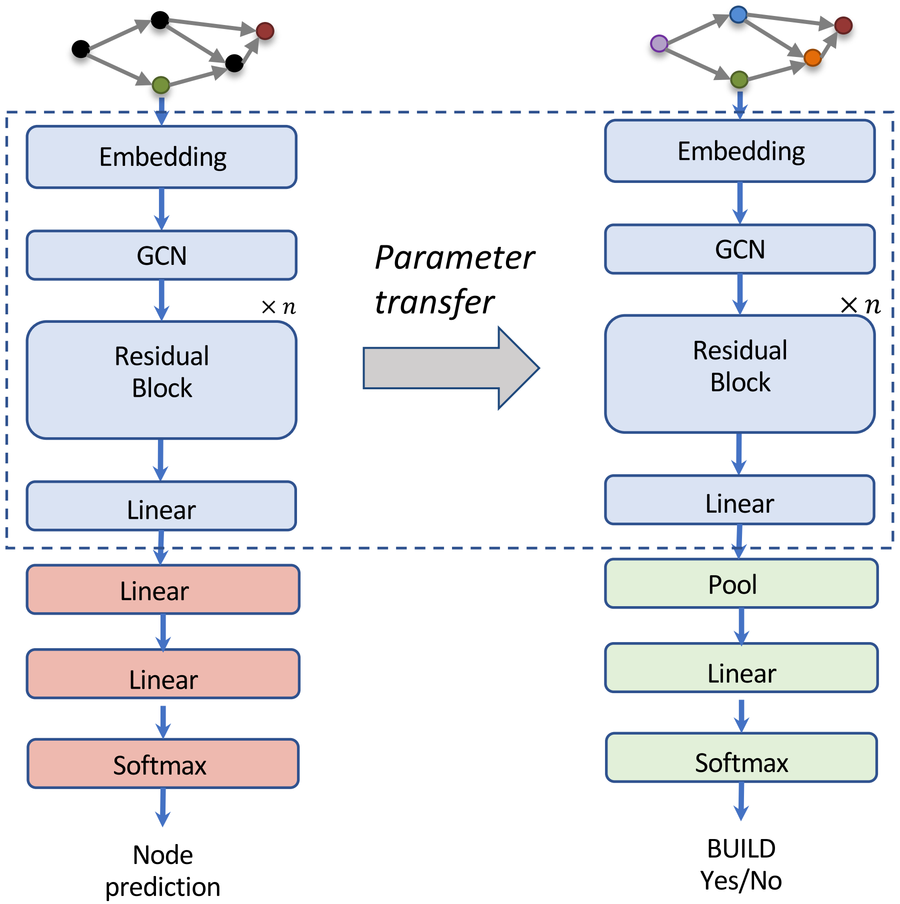
BuildCheck GNN Architecture

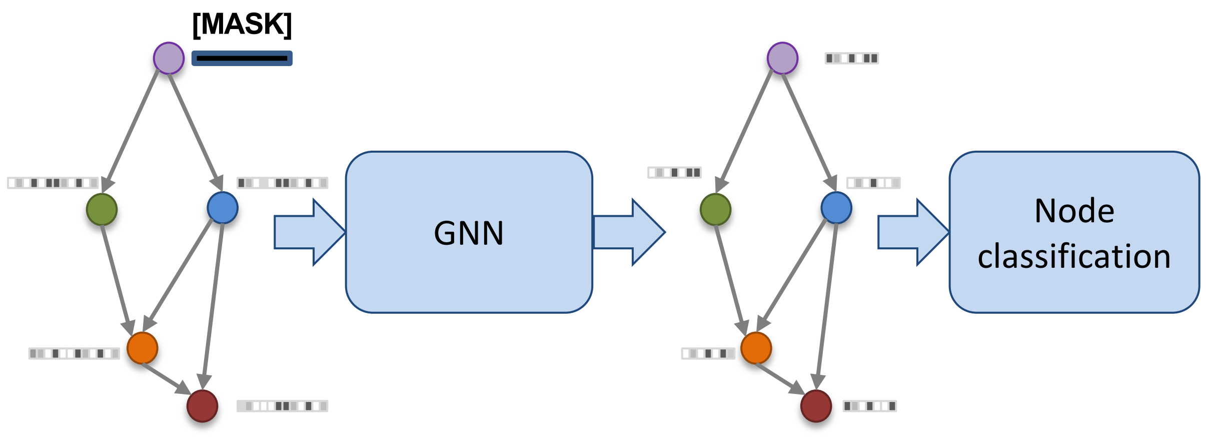


Main Components

- Multiple Graph Convolutional layers.
- Embedding layer: maps package information into a continuous vector space
- Residual block: aids in training deeper networks.
- Pool layer: computes the average of all node features and creates a representation of the entire graph

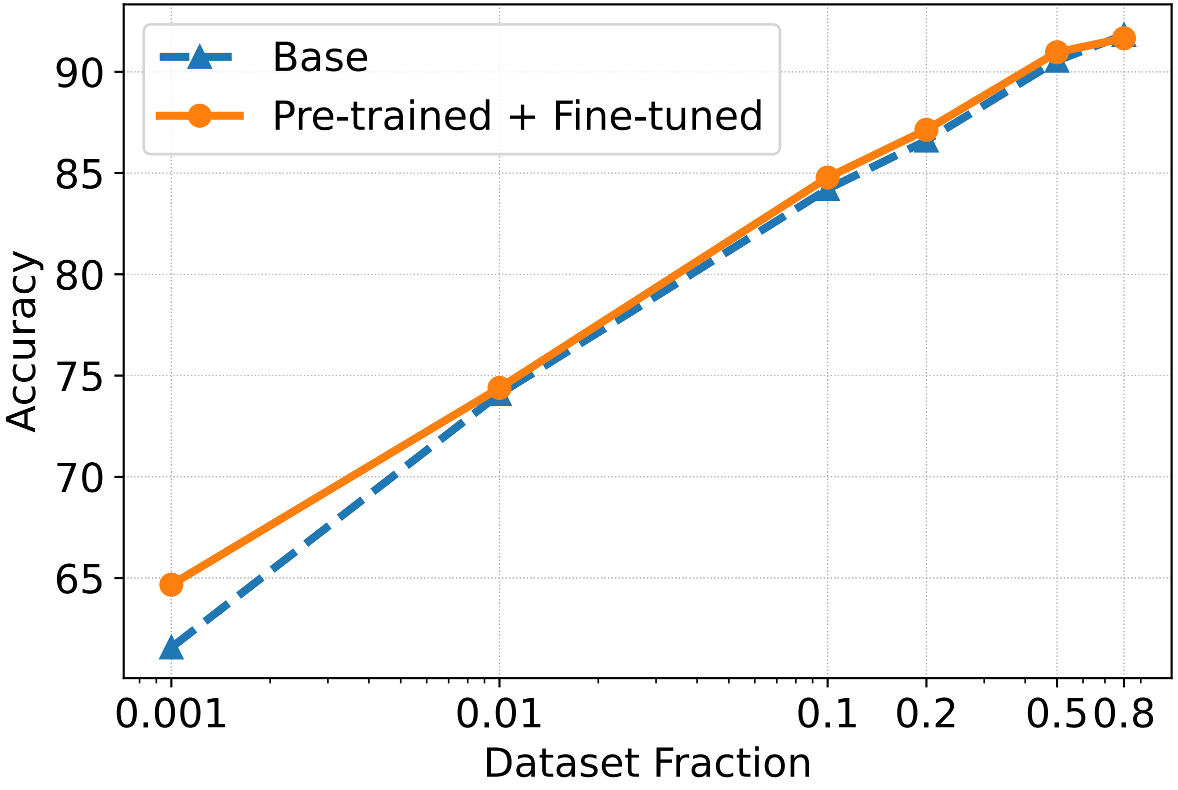
Self-Supervised Pretraining Task for Learning Node Embeddings for Downstream tasks

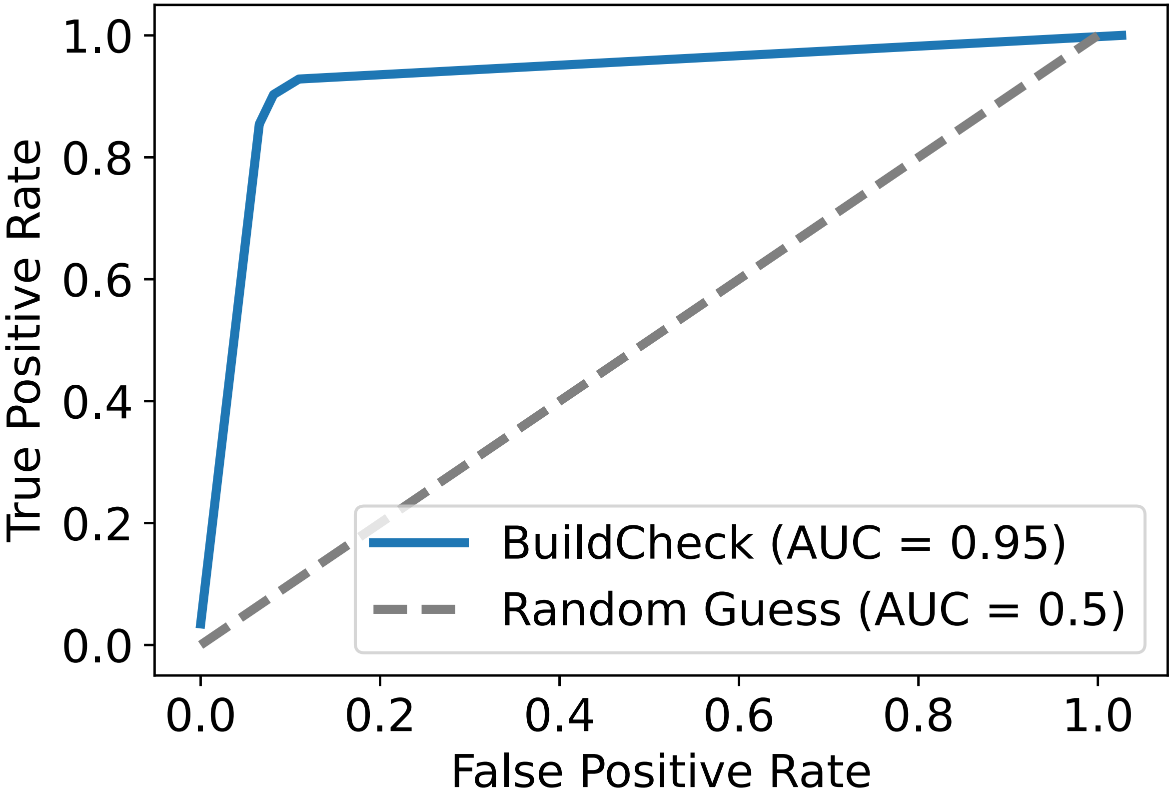




Pre-trained package dependency model can be used for build prediction with fine-tuning

Evaluation





Our model achieves an accuracy of 91% on E4S build dataset

Conclusion and Ongoing Work

- Demonstrated how to combine the capabilities of Graph Neural Networks and advanced package management technologies to offer practical solutions for managing package dependencies
- BuildCheck can eliminate very expensive trial-and-error exercise to find working builds
- Use the outputs of BuildCheck in Spack's concretizer solver
- Integrate the model with the Spack's Continuous Integration system

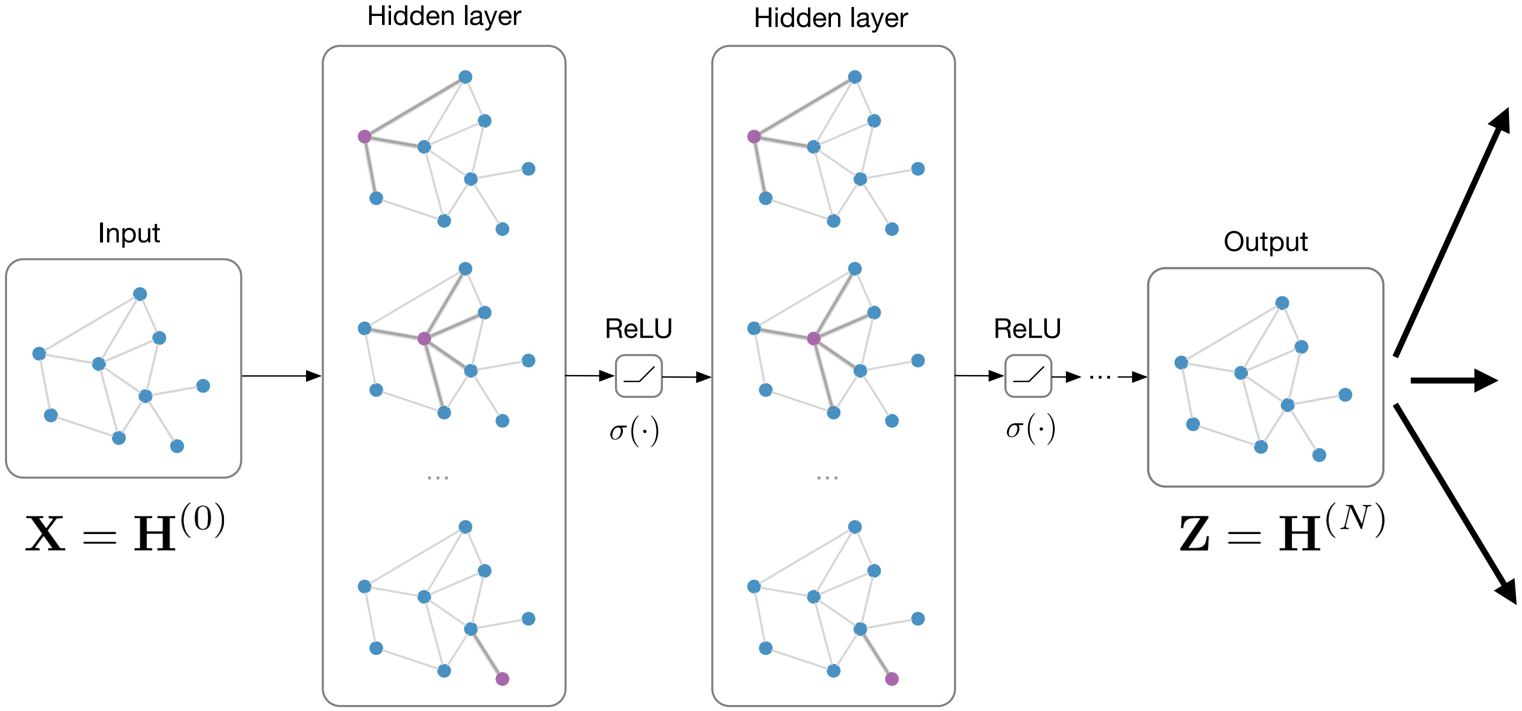
CASC

Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

This work was prepared by LLNL under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 21-SI-005.

Graph Neural Network



$\mathbf{X} = \mathbf{H}^{(0)}$

$\mathbf{Z} = \mathbf{H}^{(N)}$

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right)$$

Node classification:

$$\mathrm{softmax}(\mathbf{z_n})$$

e.g. Kipf & Welling (ICLR 2017)

Graph classification:

$$\mathrm{softmax}(\textstyle\sum_n \mathbf{z_n})$$

e.g. Duvenaud et al. (NIPS 2015)

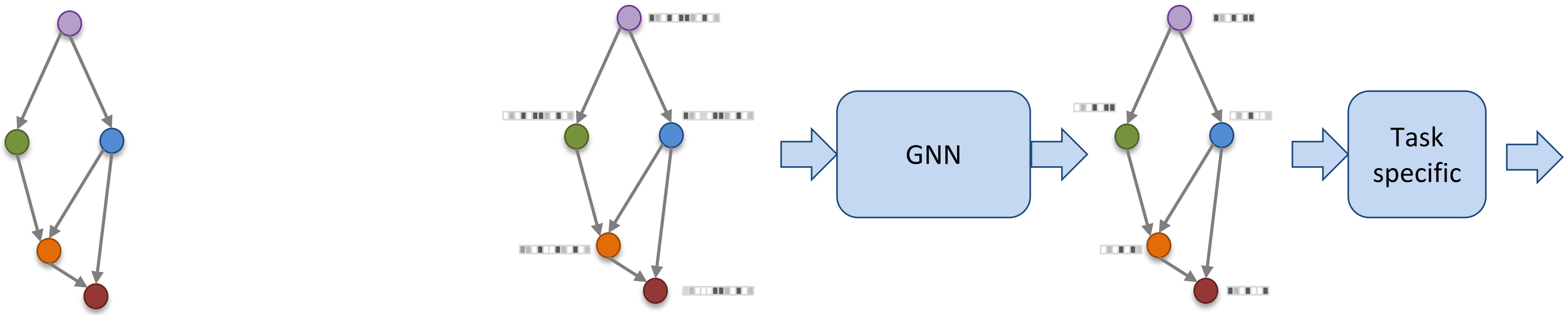
Link prediction:

$$p(A_{ij}) = \sigma(\mathbf{z_i^T z_j})$$

Kipf & Welling (NIPS BDL 2016)
"Graph Auto-Encoders"

* figure from Thomas Kipf, University of Amsterdam

Graph Neural Network

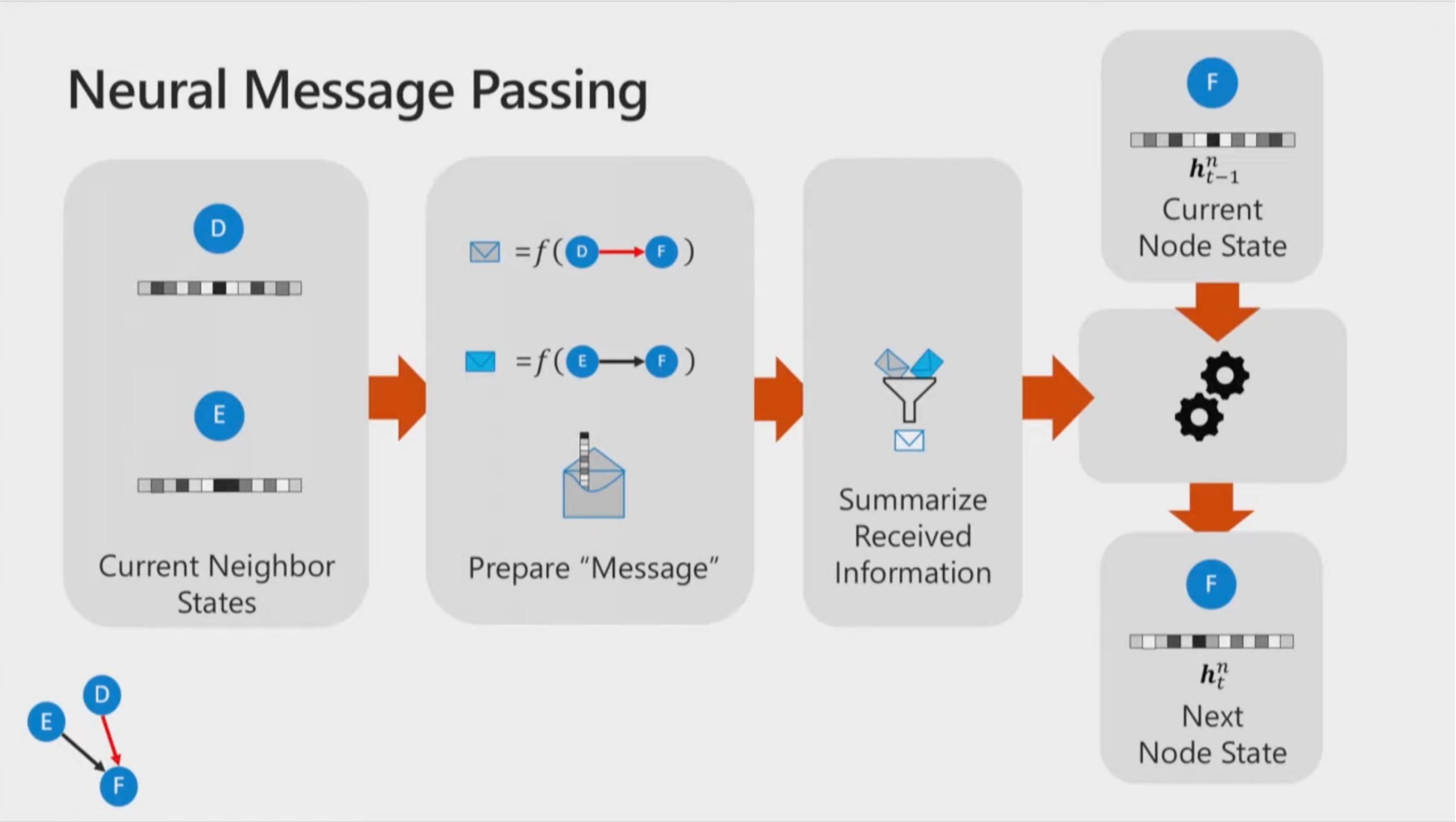


Graph representation of the problem

Initial representation of each node

Output representation of each node

Graph Neural Network

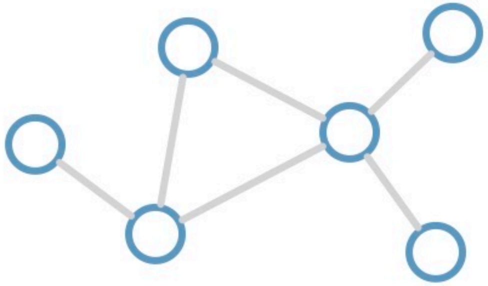


* fiigure from Miltos Allamanis, MSR lecture series

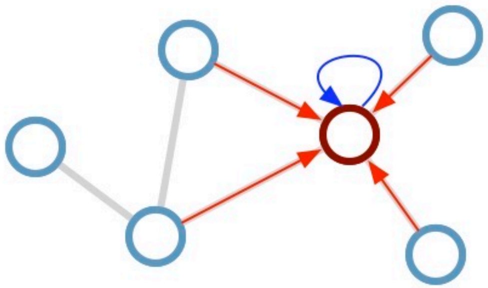
Graph Convolutional Networks (GCNs)

Kipf & Welling (ICLR 2017), related previous works by Duvenaud et al. (NIPS 2015) and Li et al. (ICLR 2016)

Consider this undirected graph:



Calculate update for node in red:



Update rule:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{h}_i^{(l)}\mathbf{W}_0^{(l)} + \sum_{j\in\mathcal{N}_i} \frac{1}{c_{ij}}\mathbf{h}_j^{(l)}\mathbf{W}_1^{(l)}\right)$$

$\mathcal{N}_i$: neighbor indices

c_{ij} : norm. constant (fixed/trainable)

Graph Neural Networks (GNN) are ideally suited for this application

The package dependency graph is a directed acyclic graph (DAG)

Graph is represented as $G = (V, E)$, where V is the set of nodes representing the packages and E is the set of edges capturing the dependencies.

We cast the build success prediction problem as a supervised learning problem.

- Given a dataset $D = \{(G_i, y_i)\}^M$ of M graphs Gi and their corresponding build outcomes $y \in$ {SUCCESS, FAILURE}, i