

★ Get unlimited access to all of Medium. [Become a member](#)



Graph Neural Networks for Knowledge Tracing



Anirudhan, Jacob, Zach Final Project · [Follow](#)

Published in Stanford CS224W GraphML Tutorials · 16 min read · May 16



3



By *Anirudhan Badrinath, Jacob Smith, and Zachary Chen* as part of the *Stanford CS224W Winter 2023* course project.

Feel free to follow along with our [Colaboratory](#) notebook!

Introduction

In this blog post, we explore the application of graph neural networks (GNNs) in online tutoring systems. In this study, we primarily focus on

knowledge tracing (KT), an educational framework to estimate and model student cognitive mastery in skills over time. Given previous problem solving sequences across various skills, the prediction task associated with KT is future response correctness prediction (i.e., how likely is a student to correctly solve the problem?), given feedback of past interactions with an online tutoring system. KT has been used in many massive open online courses (MOOCs) and computerized tutoring systems, such as Khan Academy and CognitiveTutor. Improvements in KT could lead to better tailoring of skill learning suggestions for students, resulting in more effective and efficient learning.

Problem Statement

Traditional KT methods such as Bayesian Knowledge Tracing [1] have typically assumed a skill independence condition (e.g., between different areas/topics such as Addition and Subtraction), which are unlikely to hold in practical educational settings. Although recent KT methods leverage deep learning without the skill independence assumption, none exploit the existing interdependence between skills by explicitly modeling their structure. For example, state-of-the-art deep KT methods such as DKT [2] and SAKT [3] one-hot encode skills as if they are separate and unrelated. To address this, we propose creating an explicit graph structure describing the interaction between skills in student problem solving sequences, leveraging

a graph neural network to construct skill embeddings that directly model this interdependence.

Modeling Skill Interdependence with Graphs

To motivate the construction of a graph which captures relevant relationships between skills and student interactions, we examine the distribution of skills attempted within student problem solving sequences in the canonical ASSISTments 2009–10 SkillBuilder dataset [4]. Across all sequences, the median number of skills attempted by a student is 4, while the attempted number of skills at the 75th percentile is 9. Importantly, the rate of co-occurrence between skill pairs such as (“Addition and Subtraction Integers”, “Addition and Subtraction Fractions”), (“Addition Whole Numbers”, “Addition and Subtraction Integers”), and (“Addition and Subtraction Integers”, “Multiplication and Division Integers”) within a student’s sequence is approximately 19.2–21.9x higher than random. Clearly, there exists a logical ordering in which students attempt questions between different skills, which can result from either a difficulty-based ordering, chosen by the student, or typical pedagogical ordering, chosen by either student or instructor, between the skills. We propose that the context provided by these similar “neighbouring” skills can be leveraged to more accurately and effectively assess student knowledge.

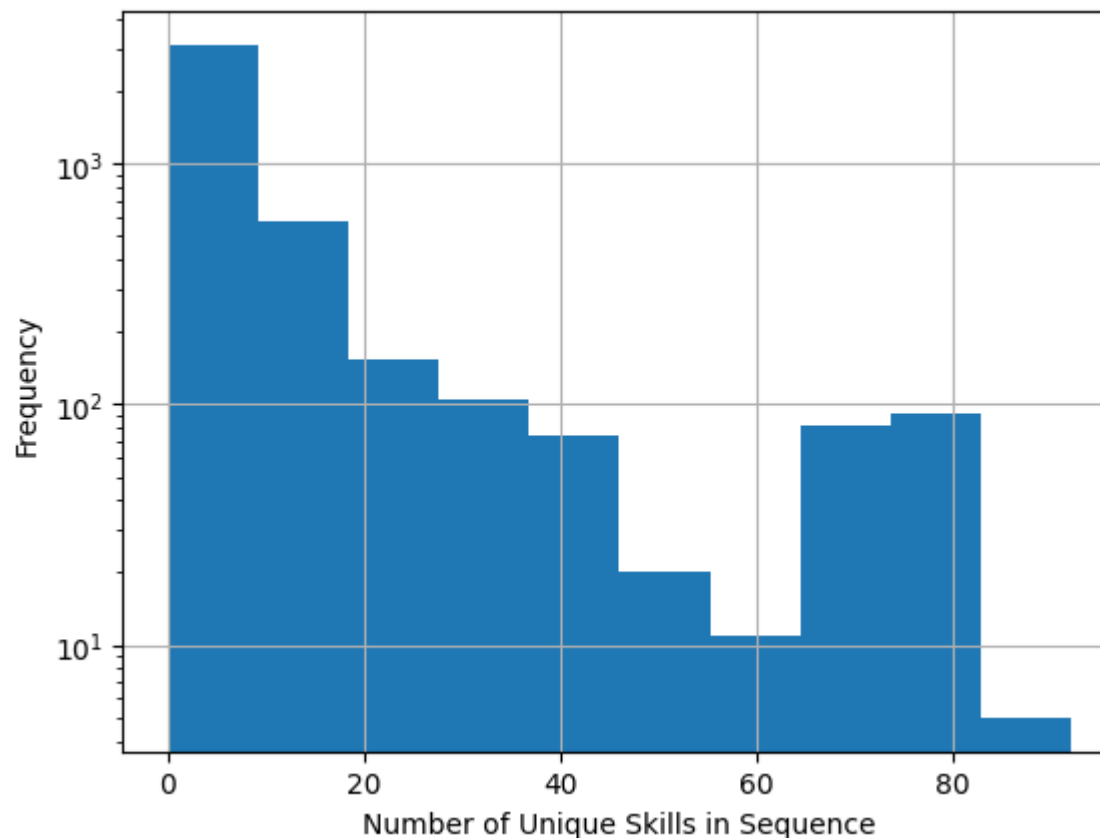


Figure 1: Distribution of number of unique skills attempted in a student sequence in ASSISTments 2009–10 SkillBuilder dataset.

We construct an expressive skill graph that retains both information about (a) skill co-occurrence within student sequences and (b) the order in which skills are typically accessed. We define an ordered co-occurrence as a pair of two skills A and B such that A occurs within the same student sequence and precedes an occurrence of B. The nodes in the skill graph correspond to each skill in the dataset, with the existence of a directed edge between two

nodes if there is a non-trivial rate of ordered co-occurrence between them. Although more complex techniques could be used to define a non-trivial rate of ordered co-occurrence, we set a threshold of 0.1.

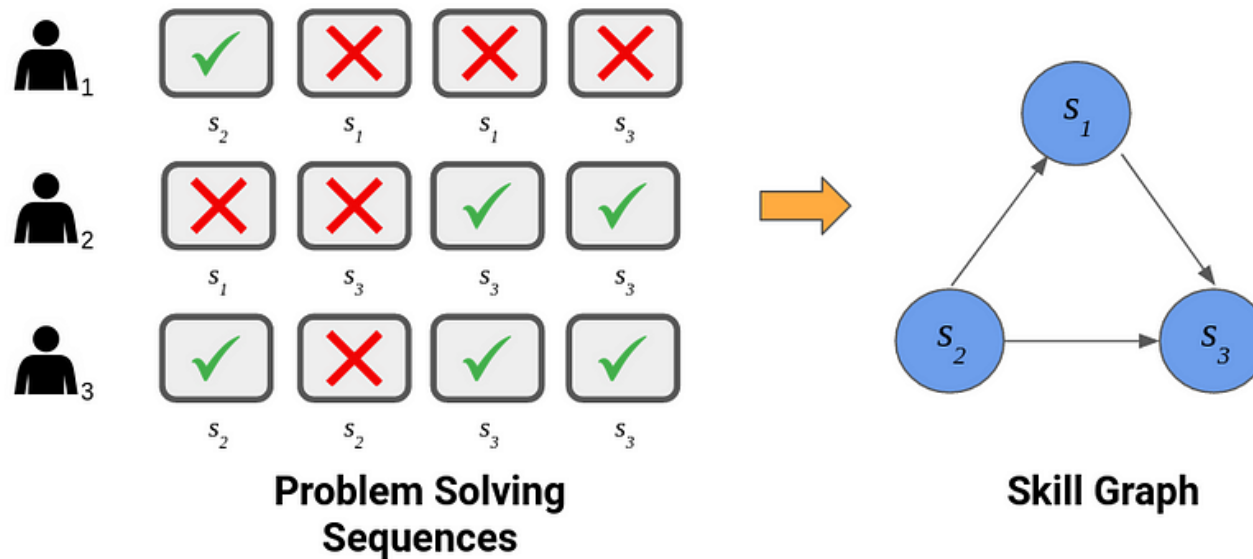


Figure 2: Example of 3 student problem solving sequences with tagged skills s_i , along with the corresponding skill graph.

As an illustrative example, consider the 3 sequences shown in Figure 1, where s_2 always precedes s_1 and s_1 always precedes s_3 . By transitivity, we know that s_2 precedes s_3 . These ordered co-occurrence (or precedence) patterns are reflected in the skill graph displayed in Figure 1 through the edges, where there is a directed edge from s_2 to s_1 and s_3 and from s_1 to s_3 .

We construct the skill graph using the following code snippet that leverages `networkx`, which can be found in our Colab notebook.

```
def create_skill_graph(df):  
    # Cache and load if exists.  
    if os.path.exists('skill_graph.pickle'):  
        print("Using existing skill graph...")  
        return pickle.load(open('skill_graph.pickle', 'rb')), pickle.load(open('skill_graph.pickle', 'rb'))  
  
    # Pre-processing steps to remove unwanted responses and group into buckets.  
    print("Constructing skill graph...")  
    df = df[~df['skill_name'].isna()]  
    grouped = df.groupby('user_id')['skill_name'].agg(list)  
    uniques = list(df['skill_name'].unique())  
  
    # Count ordered co-occurrences in each student sequence.  
    skill_cooccurs = {skill_name: np.zeros(df['skill_name'].nunique())  
                      for skill_name in uniques}  
    for seq in tqdm(grouped.values):  
        cooccur = np.zeros(df['skill_name'].nunique())  
        for s in reversed(seq):  
            cooccur[uniques.index(s)] += 1  
            skill_cooccurs[s] = skill_cooccurs[s] + cooccur  
  
    # Normalize distribution and round to remove noise.  
    skill_cooccurs = {k: (v / sum(v)).round(1)  
                      for k, v in skill_cooccurs.items()}  
  
    dod = {}  
    for i, (skill_name, edges) in enumerate(skill_cooccurs.items()):  
        dod[i] = {}  
        for j, e in enumerate(edges):  
            if e > 0:  
                dod[i][j] = {'weight': e}
```

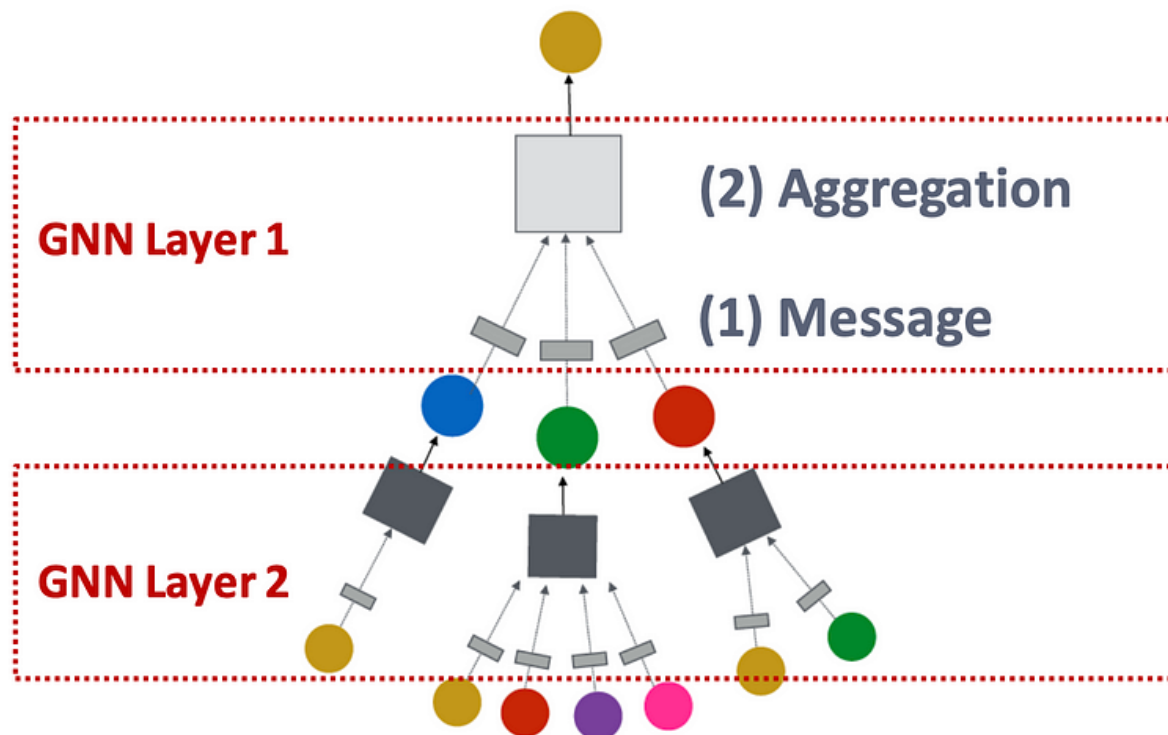
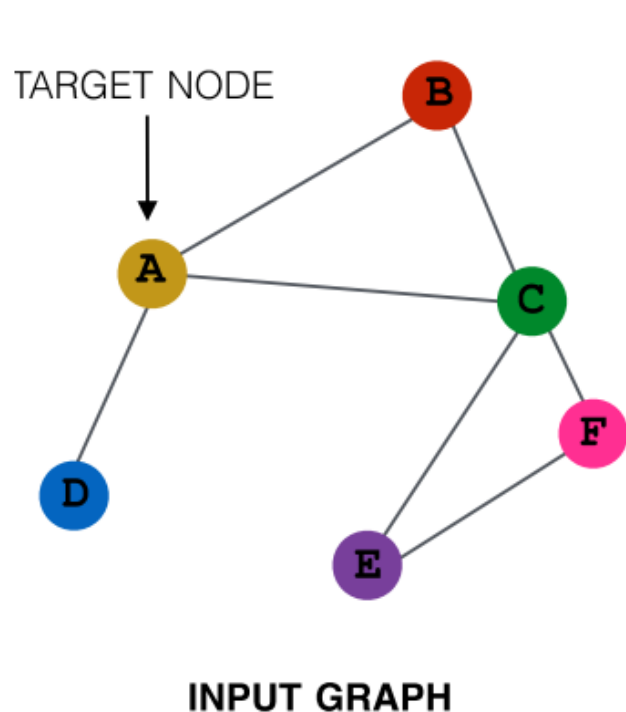
```
# Connect nodes in digraph with forward co-occurrence.
skill_graph = nx.from_dict_of_dicts(dod)
skill_dict = dict(zip(uniques, range(len(uniques))))

# Save and cache graph for future usage.
pickle.dump(skill_graph, open('skill_graph.pickle', 'wb'))
pickle.dump(skill_dict, open('skill_dict.pickle', 'wb'))
return skill_graph, skill_dict
```

Graph Neural Network Models

Figures in this section are from Lecture Slide 5 of the Stanford CS224W Winter 2023 course.

One of the main goals behind graph neural networks is to generate node embeddings that accurately represent the nodes in the graph. To do this, we can define each node's neighborhood as a computation graph, through which neighboring nodes are able to transform and propagate information (i.e., “message passing”) to help construct a given node's node embedding. The computation graph for a single node is pictured below.



At layer 0, the node embeddings for each node are simply each node's initial node features. Then, for each subsequent layer, we can calculate each node's embedding by aggregating its neighbors' previous layer node embeddings. To do so, we first define a message function, with the intuition being that each node will create a message to propagate to other nodes.

$$\mathbf{m}_u^{(l)} = \text{MSG}^{(l)} \left(\mathbf{h}_u^{(l-1)} \right)$$

Here \mathbf{m} in the equation represents the message of a node u at layer l , and \mathbf{h} in the equation represents the node embedding of node u at layer $l - 1$. After computing the messages, we then aggregate them as follows:

$$\mathbf{h}_v^{(l)} = \text{AGG}^{(l)} \left(\left\{ \mathbf{m}_u^{(l)}, u \in N(v) \right\} \right)$$

Note that we include the message generated from the current node's previous layer embedding so as to preserve information about the node itself. It is also common to add activation functions for non-linear expressiveness to the message and/or aggregation step. Putting it all together, to calculate the node embedding at a given layer for a given node, we take the previous layer's embeddings for the node and its neighbors, pass them through a message function, then aggregate them (can use `sum()`, `mean()`, `max()`, etc) and apply an activation function somewhere throughout this process. To construct our graph neural networks, we use the PyG (PyTorch Geometric) library.

Graph Convolutional Network (GCN) [6]

A GCN is an example of a classical graph neural network in which the computation for the node embedding \mathbf{h} of a node v at a given layer l is defined as follows:

$$\mathbf{h}_v^{(l)} = \sigma \left(\underbrace{\sum_{u \in N(v)} \left(\underbrace{\mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}}_{\text{Message}} \right)}_{\text{Aggregation}} \right)$$

Here, the message layer consists of multiplying by the layer l learnable weight matrix \mathbf{W} , then normalizing by node degree of the node v . Then we use a sum aggregation to aggregate the messages, and finally apply an activation function.

We implemented this architecture using the following code snippet:

```
class GCN(torch.nn.Module):
    def __init__(self, num_skills, hidden_dim = 128):
        """
        Represents a simple 3-layer Graph Convolutional Network (GCN)
        with embedding and hidden dimension of hidden_dim.
        """
        super().__init__()
        self.tag = 'GCN'
        self.pre_embs = nn.Embedding(num_skills, hidden_dim)
        self.conv1 = GCNConv(hidden_dim, hidden_dim)
        self.prelu1 = nn.PReLU()
        self.conv2 = GCNConv(hidden_dim, hidden_dim)
        self.prelu2 = nn.PReLU()
        self.conv3 = GCNConv(hidden_dim, hidden_dim)
        self.prelu3 = nn.PReLU()
        self.out = nn.Linear(hidden_dim, hidden_dim)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x, edge_index, edge_weight):
        """
        Runs a forward pass through the GCN with given initial skill IDs and
        edge_index and edge_weights.

        Arguments:
        - x: skill IDs (torch.Tensor)
        - edge_index: edges in skill graph (torch.Tensor)
        - edge_weight: edge weights of skill graph (torch.Tensor)

        Returns:
        - final node embedding for skill
        """
        h0 = self.pre_embs(x)
```

```

h1 = self.dropout(self.prelu1(self.conv1(h0, edge_index, edge_weight = e
h2 = self.dropout(self.prelu2(self.conv2(h1, edge_index, edge_weight = e
h3 = self.prelu3(self.conv3(h2, edge_index, edge_weight = edge_weight))
return self.out(h3)

```

GraphSAGE [7]

GraphSAGE is another example of a commonly used GNN layer. The SAGEConv layer for GraphSAGE, as it is implemented in torch_geometric, sums the previous layer's node embedding multiplied by a learned weight matrix with the aggregation of messages from its neighbors multiplied by another learned weight matrix. This vector is passed through a linear layer and a non-linearity, such as PReLU, then passed through a final linear layer to arrive at the next layer's embedding at the desired embedding dimension. The equation for a single SAGEConv layer is as follows:

$$\mathbf{h}_v^{(l)} = \mathbf{W}_1 \cdot \mathbf{h}_v^{(l-1)} + \mathbf{W}_2 \cdot \text{mean}_{u \in N(v)} \mathbf{h}_u^{(l-1)}$$

The message passing is simply an embedding lookup of neighboring nodes. Aggregation then occurs at two levels. First, an aggregation function such as mean pooling is applied over the set of messages. We use mean pooling in

this work. The mean aggregation function takes a weighted element wise average of the neighbors. Finally, we use dropout in the forward pass for each of the message passing layers except the last.

We implemented this architecture using the following code snippet:

```
class GraphSAGE(torch.nn.Module):
    def __init__(self, num_skills, hidden_dim = 128):
        """
        Represents a 3-layer GraphSAGE GNN model
        with embedding and hidden dimension of hidden_dim.
        """
        super().__init__()
        self.tag = 'GraphSAGE'
        self.pre_embs = nn.Embedding(num_skills, hidden_dim)
        self.conv1 = SAGEConv(hidden_dim, hidden_dim)
        self.prelu1 = nn.PReLU()
        self.conv2 = SAGEConv(hidden_dim, hidden_dim)
        self.prelu2 = nn.PReLU()
        self.conv3 = SAGEConv(hidden_dim, hidden_dim)
        self.prelu3 = nn.PReLU()
        self.out = nn.Linear(hidden_dim, hidden_dim)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x, edge_index, edge_weight):
        """
        Runs a forward pass through GraphSAGE with given initial skill IDs and
        edge_index and edge_weights.

        Arguments:
        - x: skill IDs (torch.Tensor)
```

```

- edge_index: edges in skill graph (torch.Tensor)
- edge_weight: edge weights of skill graph (torch.Tensor)

Returns:
- final node embedding for skill
"""
h0 = self.pre_embs(x)
h1 = self.dropout(self.prelu1(self.conv1(h0, edge_index)))
h2 = self.dropout(self.prelu2(self.conv2(h1, edge_index)))
h3 = self.prelu3(self.conv3(h2, edge_index))
return self.out(h3)

```

Graph Attention Network (GAT) [8]

GATs are quite similar to GCNs but with the key addition of having learned attention weights to focus on the more important neighbors when calculating node embeddings.

$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)} \right)$$

Attention weights

Note that this is the same equation as for GCN but instead of normalizing by node degree we instead apply the attention weights. To calculate these

attention weights, we first use an attention mechanism a , to compute the attention coefficients e across pairs of nodes.

$$e_{vu} = a(\mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}, \mathbf{W}^{(l)} \mathbf{h}_v^{(l-1)})$$

e_{vu} indicates the importance of u 's message to node v

Here a can simply be a concatenation followed by a linear layer, where we can train the parameters of a jointly with the weight matrices. Once we have e , we can then normalize using the softmax function so that for a given node v , the sum of the attention weights for its neighbors towards v is equal to 1.

$$\sum_{u \in N(v)} \alpha_{vu} = 1: \quad \alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

To stabilize the learning process of the attention mechanism, we can do multi-headed attention, where we create multiple attention scores each with their own parameters, then aggregate the outputs, as follows:

$$\mathbf{h}_v^{(l)}[1] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^1 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

$$\mathbf{h}_v^{(l)}[2] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^2 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

$$\mathbf{h}_v^{(l)}[3] = \sigma(\sum_{u \in N(v)} \alpha_{vu}^3 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

$$\mathbf{h}_v^{(l)} = \text{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$$

We implemented this architecture using the following code snippet:

```
class GAT(torch.nn.Module):
    def __init__(self, num_skills, hidden_dim = 128):
        """
        Represents a 3-layer Graph Attention Network (GAT)
        with embedding and hidden dimension of hidden_dim.
        """
        super().__init__()
        self.tag = 'GAT'
        self.pre_embs = nn.Embedding(num_skills, hidden_dim)
        self.conv1 = GATConv(hidden_dim, hidden_dim)
```



```

self.prelu1 = nn.PReLU()
self.conv2 = GATConv(hidden_dim, hidden_dim)
self.prelu2 = nn.PReLU()
self.conv3 = GATConv(hidden_dim, hidden_dim)
self.prelu3 = nn.PReLU()
self.out = nn.Linear(hidden_dim, hidden_dim)
self.dropout = nn.Dropout(0.3)

def forward(self, x, edge_index, edge_weight):
    """
    Runs a forward pass through GAT with given initial skill IDs and
    edge_index and edge_weights.

    Arguments:
        - x: skill IDs (torch.Tensor)
        - edge_index: edges in skill graph (torch.Tensor)
        - edge_weight: edge weights of skill graph (torch.Tensor)

    Returns:
        - final node embedding for skill
    """
    h1 = self.dropout(self.prelu1(self.conv1(self.pre_embs(x), edge_index)))
    h2 = self.dropout(self.prelu2(self.conv2(h1, edge_index)))
    h3 = self.prelu3(self.conv3(h2, edge_index))
    return self.out(h3)

```

Methodology

We construct GraphKT, a graph-based response correctness prediction model for student problem solving sequences through sequence modeling (leveraging a transformer architecture as in [3]). Unlike prior KT methods that use one-hot encoding to represent skill values, GraphKT leverages node

embeddings for each skill constructed by a graph neural network (GNN) applied to the skill graph. In this study, we focus on GCN, GraphSAGE, and GAT as the chosen methods for constructing node embeddings.

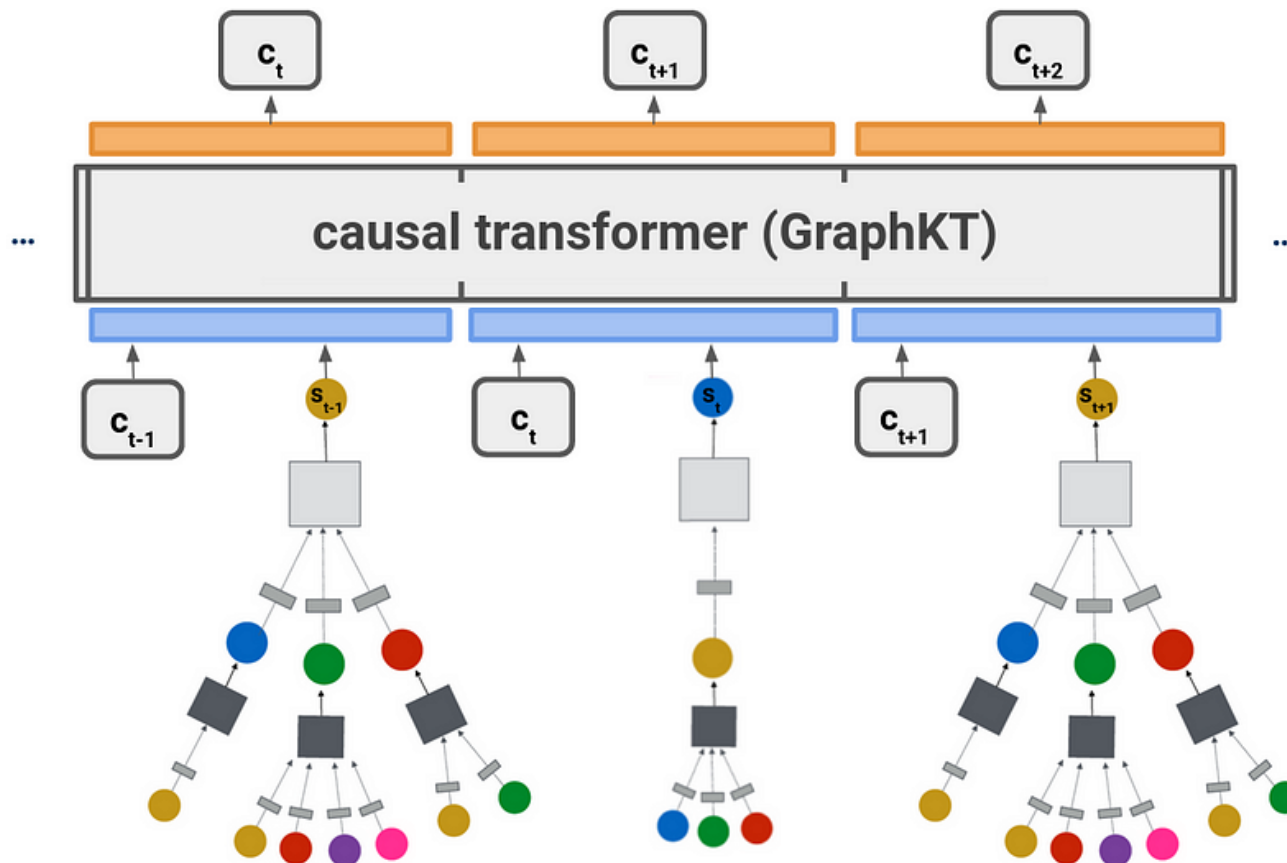


Figure 3: Architecture of GraphKT, where c_t represents the correctness values at time t and s_t represents node embeddings outputted by a GNN applied to the skill graph. Node embedding graphic taken from [Lecture Slide 5 of the Stanford CS224W Winter 2023 course](#).

As shown in Figure 3, we pass in the skill index s_t at time t within a student sequence as input to the GNN, which is transformed through a fixed-size dictionary of initial skill embeddings. Although this process is inherently transductive since it relies on the skill index, we believe this does not detract from the practical applications of the method as KT is by nature restricted to transductive settings (i.e., we cannot perform predictions for unknown skills).

The output of the GNN is a node embedding for the skill attempted by the student, which is concatenated with the previous teacher-forced correctness value (i.e., at time $t - 1$) to form the input features for GraphKT. With these input features, we apply a forward pass through the causal transformer, performing multi-head attention with future values masked out (i.e., in a causal fashion). At each timestep, we apply a linear layer to the transformer output to transform it into a scalar and a sigmoid activation to transform it into a binary probability value.

To optimize the causal transformer, the GNN, and the dictionary of initial skill embeddings end-to-end, we leverage backpropagation based on a binary cross-entropy loss applied to the predicted (correctness) probability and actual correctness values, as shown below.

$$\arg \min_{\theta} - \sum_{\tau \in \mathcal{D}} \sum_{c_t \in \tau} c_t \log P(\hat{c}_t = 1) + (1 - c_t) \log P(\hat{c}_t = 0)$$

Our training function is shown, as implemented, in code below.

```
def train(model, skill_net, data_train, data_val, num_epochs, baseline = False):
    """
    Train the KT transformer and GNN end-to-end by optimizing the KT binary
    cross-entropy objective. Arguments:
        - model (transformer for KT)
        - skill_net (GNN for skill embeddings)
        - data_train (training data)
        - data_val (validation data)
        - num_epochs (number of training epochs)
    """
    for epoch in range(num_epochs):
        # Train model for num_epochs epochs.
        model.train()
        skill_net.train()
        batches_train = construct_batches(data_train, epoch = epoch)
        pbar = tqdm(batches_train)
        losses = []
        for X, y in pbar:
            optimizer.zero_grad()
            # Get node embeddings for all skills from skill_net (GNN).
            all_skill_embd = skill_net(torch.arange(110).cuda(),
                                       skill_graph.edge_index.cuda(),
                                       skill_graph.weight.cuda().float())
            # Select node embeddings corresponding to skill tagged with data.
            skill_embd = all_skill_embd[torch.where(X[..., 0] == -1000, 0, X[...
            ohe = torch.eye(110).cuda()[torch.where(X[..., 0] == -1000, 0, X[...
            # Concatenate data with skill embedding and one-hot encoding of skill
            if baseline:
```

```

        feat = [X, ohe]
    else:
        feat = [X, skill_embd, ohe]
    output = model(torch.cat(feat, dim = -1), skill_idx = y[..., 0].data)
    # Compute loss and mask padded values.
    mask = (y[..., -1] != -1000).ravel()
    loss = F.binary_cross_entropy(output[mask], y[..., -1:].ravel()[mask])
    # Backpropagate and take a gradient step.
    loss.backward()
    optimizer.step()
    # Report the training loss.
    losses.append(loss.item())
    pbar.set_description(f"Training Loss: {np.mean(losses)}")
if epoch % 1 == 0:
    # Evaluate model using validation set.
    batches_val = construct_batches(data_val, val = True)
    model.eval()
    skill_net.eval()

    # Construct predictions based on current model and compute error(s).
    ypred, ytrue = evaluate(model, batches_val, baseline = baseline)
    auc = roc_auc_score(ytrue, ypred)
    acc = (ytrue == ypred.round()).mean()
    rmse = np.sqrt(np.mean((ytrue - ypred) ** 2))
    # Report error metrics on validation set and save checkpoint.
    print(f"Epoch {epoch}/{num_epochs} - [VALIDATION AUC: {auc}] - [VALI
    torch.save(model.state_dict(), f"ckpts/model-{skill_net.tag}-{epoch}
    if not baseline:
        torch.save(skill_net.state_dict(), f"ckpts/skill_net-{skill_net.

```

Data

We leverage the ASSISTments 2009–10 SkillBuilder dataset [4] containing 124 skills (e.g., constructing pie charts) and around 525,000 total binary correctness responses (with a ~70/30% split of correct and incorrect aggregated across skills). We choose this dataset as it is moderately large and has been used throughout the knowledge tracing literature for comparisons across different methods as a reliable indicator of KT performance [2][3][5].

To pre-process the data, we remove all sequences of length 1 or sequences with undefined skill tags. We organize the sequences by grouping using the user ID, using one-hot encoding to encode the correctness. For training, we pad the sequences to a maximum of length 2,048. When computing the loss and backpropagating, we mask out these padded values.

We preprocessed our data and constructed our batches with the following code:

```
def preprocess_data(data):  
    """  
    Pre-process data and pad to the maximum length.  
    """  
    features = ['skill_id', 'correct']  
    seqs = data.groupby(['user_id']).apply(lambda x: x[features].values.tolist())  
    # ensure sequence is not too long  
    length = min(max(seqs.str.len()), block_size)  
    seqs = seqs.apply(lambda s: s[:length] + (length - min(len(s), length)) * [
```

```

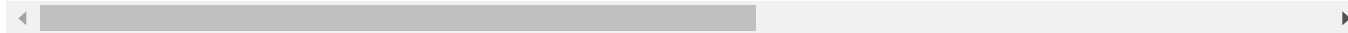
    return seqs

def construct_batches(raw_data, epoch = 0, val = False):
    """
    Construct batches based on tabular KT data with user_id, skill_id, and
    correctness. Pads to the minimum of the maximum sequence length and the
    block size of the transformer.
    """
    np.random.seed(epoch)
    user_ids = raw_data['user_id'].unique()
    # Loop until one epoch of training.
    for _ in range(len(user_ids) // batch_size):
        user_idx = raw_data['user_id'].sample(batch_size).unique()
        if not val:
            filtered_data = raw_data[raw_data['user_id'].isin(user_idx)].sort_values
            batch_preprocessed = preprocess_data(filtered_data)
            batch = np.array(batch_preprocessed.to_list())
            # Next token prediction.
            X = torch.tensor(batch[:, :-1, ..., :], requires_grad=True, dtype=torch.
            y = torch.tensor(batch[:, 1:, ..., [0, 1]], requires_grad=True, dtype=to
            for i in range(X.shape[1] // block_size + 1):
                if X[:, i * block_size: (i + 1) * block_size].shape[1] > 0:
                    yield [X[:, i * block_size: (i + 1) * block_size], y[:, i * bloc

```

Experimental Setup

To train the baseline approach, SAKT, we use a transformer with 2 layers, 8 heads, and an embedding dimension of 128. The dropout probability on the embeddings is 0.1, whereas it is 0.3 on the embedding and multi-head attention components.



To train GraphKT, we perform a grid hyperparameter search for batch size, learning rate, embedding size, and number of layers. We choose a batch size of 8 based on hardware restrictions. We use the AdamW optimizer with a learning rate of $1e-4$, a node embedding dimension of 128, and a total of 3 message passing layers for all models. For each of the GNNs, we use a dropout probability of 0.3 applied to the outputs of all but the final GNN layer. The hyperparameters and architecture for the causal transformer are identical to the aforementioned configuration for SAKT.

We utilize an 81–9–10 data split of training-validation-test. We implement our data split with the following code:

```
def train_test_split(data):  
    """  
    Performs a deterministic train-test split based on the tabular data provided  
    Note that this function needs to be called twice to perform a train-val-test  
    split as desired.  
  
    Arguments:  
    - data: tabular KT dataset (pd.DataFrame)  
  
    Returns:  
    - data_train: training dataset  
    - data_val: validation/testing dataset  
    """  
    np.random.seed(42)  
    data = data.set_index(['user_id', 'skill_name'])
```



```

idx = np.random.permutation(data.index.unique())
train_idx, test_idx = idx[:int(train_split * len(idx))], idx[int(train_split
data_train = data.loc[train_idx].reset_index()
data_val = data.loc[test_idx].reset_index()
return data_train, data_val

```

Results

To evaluate GraphKT, we use 3 metrics that are commonly used in KT: area under the ROC curve (AUC), accuracy, and root mean square error (RMSE) [5]. To visualize the performance of the models, we plot the chosen metrics as a function of the training epoch in Figure 3. Based on the subfigures, it is clear that the graph-based approaches achieve improved performance by a reasonable margin compared to SAKT (blue line). GCN achieves the highest accuracy and the lowest RMSE at the end of 15 epochs compared to other methods and baseline by around 1.5–2%.

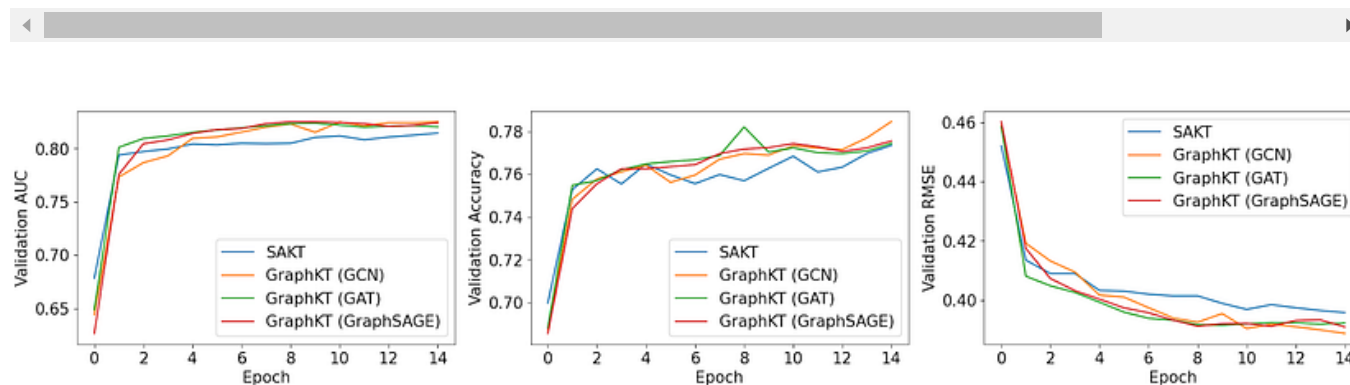


Figure 4: Validation (left) AUC, (middle) accuracy, (right) RMSE as a function of the training epoch for GraphKT and SAKT (baseline).

We evaluate the models based on the checkpoint with the best validation AUC on the test set by using bootstrap sampling across 100 samples. Based on the metrics across the samples, we construct a 95% confidence interval using the t-distribution, and the results are shown in Table 1. While the graph-based methods outperform the baseline across the metrics in terms of the average, the results are not statistically significant. Notably, GCN achieves the highest accuracy and RMSE, similarly to the validation set, with a smaller margin of improvement over the baseline. Surprisingly, GAT shows poorer performance relative to its performance on the validation set, and its performance is worse than the baseline. We believe that this may be a sign that GAT is overfitting to the task and that it may be too expressive for embedding nodes in the skill graph.

	AUC	ACC	RMSE
SAKT (Baseline)	0.818 ± 0.023	0.786 ± 0.014	0.384 ± 0.012
GraphKT (GCN)	0.820 ± 0.023	0.790 ± 0.013	0.383 ± 0.011
GraphKT (GraphSAGE)	0.821 ± 0.023	0.787 ± 0.014	0.383 ± 0.012
GraphKT (GAT)	0.819 ± 0.023	0.780 ± 0.014	0.385 ± 0.011

Table 1: Metrics on the held-out test set for the baseline and GraphKT with a 95% confidence interval.

Beyond improved performance, we demonstrate that GraphKT provides trained skill embeddings that reflect the patterns and local neighbourhood structure of the constructed skill graph, effectively modeling the interdependence between skills. In this pursuit, we can extract and visualize the trained skill embeddings constructed by the GCN using TSNE (2-dimensions). The results are shown in Figure 5, where each point represents a dimensionality-reduced embedding corresponding to a particular skill (label of the skill can be seen upon hovering).

Figure 5: 2-dimensional TSNE visualization of trained 128-dimensional skill embeddings constructed by the GCN model.

By examining the skill labels associated with each of the points in Figure 5 (by hovering), it is clear that there are clusters corresponding to pedagogically similar skills. For example, the cluster in the bottom left corner around $(-2, 3)$ consists largely of geometric skills: (“Area Parallelogram”, “Area Rectangle”, “Area Trapezoid”, “Venn Diagram”, “Volume Rectangular Prism”). As another example, near the lower center of Figure 5 around $(2, 2)$, there are skills that involve a combination of arithmetic or algebraic manipulation and some geometry (e.g., typically coordinate geometry): (“Pythagorean Theorem”, “Choose an Equation from Given Information”, “Histogram as Table or Graph”, “Interior Angles Triangle”, “Write Linear Equation from Ordered Pairs”, “Rotations”, “Reflections”). Clearly, clusters of trained skill embeddings represent pedagogically similar relationships between topics, indicating the success of the skill graph and GNN in quantifying interdependence between skills.

Conclusion

After evaluating our GNN models, GraphKT achieves a top AUC score of 0.82 and top accuracy score of 0.79. While there are reasonable improvements over the baseline approach (SAKT), the differences are not statistically

significant with the introduction of a skill graph incorporating ordered co-occurrence relations.

We hope that this blog post, in which we demonstrated the effectiveness of various graph neural network layers on Knowledge Tracing, has been an interesting read on the possibilities that GNNs have to offer. Graph neural networks are incredibly versatile at modeling complex data, and we encourage any readers to explore them on the plethora of datasets out there. Thanks for reading!

References

- [1] Corbett, Albert T., and Anderson, John R. Knowledge Tracing: Modeling the Acquisition of Procedural Knowledge (1995).
- [2] Piech, C. et al. Deep Knowledge Tracing (2015).
- [3] Pandey, S. et al. A Self-Attentive model for Knowledge Tracing (2019).
- [4] Feng, M., Heffernan, N.T., and Koedinger, K.R. Addressing the Assessment Challenge in an Intelligent Tutoring System that Tutors as it Assesses (2009).

- [5] Badrinath, A. et al. pyBKT: An Accessible Python Library of Bayesian Knowledge Tracing Models (2021).
- [6] Kipf, Thomas N., and Welling, Max. Semi-Supervised Classification with Graph Convolutional Networks (2017).
- [7] Hamilton, William L. et al. Inductive Representation Learning on Large Graphs (2018).
- [8] Veličković, P. et al. Graph Attention Networks (2018).

All images and figures were created by the authors except those otherwise specified.

Graph Neural Networks

Knowledge Tracing



Written by Anirudhan, Jacob, Zach Final Project

4 Followers · Writer for Stanford CS224W GraphML Tutorials

Follow

More from Anirudhan, Jacob, Zach Final Project and Stanford CS224W GraphML Tutorials

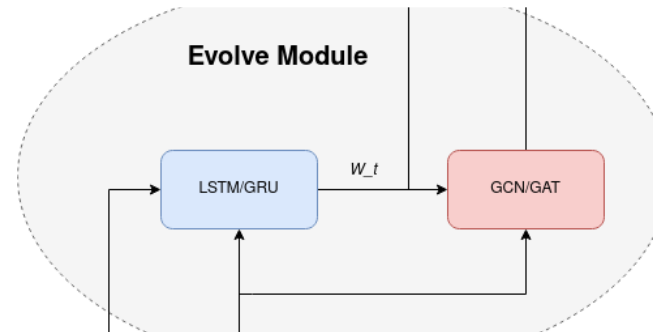


 Benjamin Witten... in Stanford CS224W GraphML ...

Spotify Track Neural Recommender System

By Eva Batelaan, Thomas Brink, and Benjamin Wittenbrink

22 min read · May 16

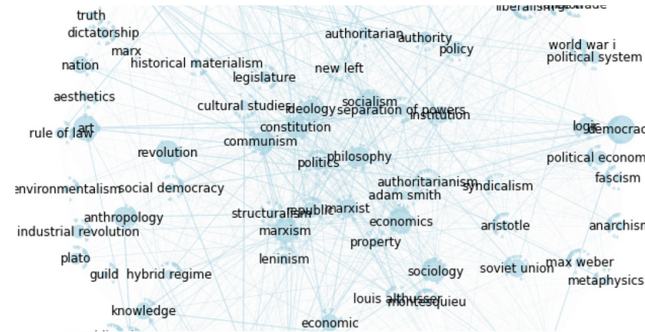


 Torstein Elias... in Stanford CS224W GraphML Tut...

Evolve GAT — A dynamic graph attention model

A new framework for reasoning over temporal graph data.

10 min read · May 14



Arjun Karan... in Stanford CS224W GraphML Tuto...

Augmenting Your Notes Using Graph Neural Networks

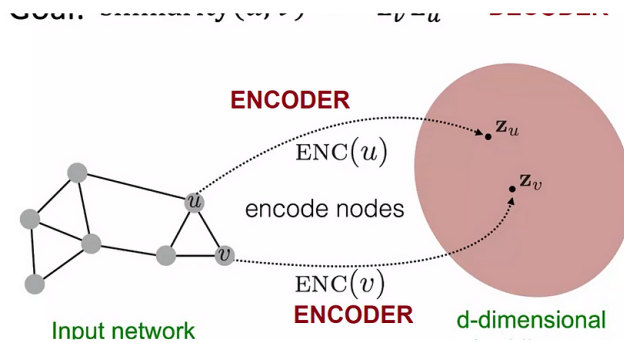
CS224w Course Project by Arjun Karanam
and Michael Elabd

16 min read · May 16



See all from Stanford CS224W GraphML
Tutorials

Recommended from Medium



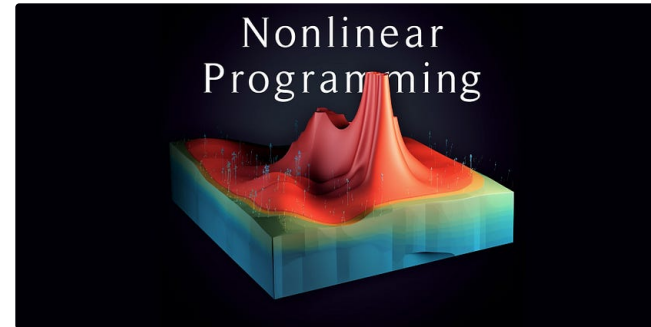
Faxi Yuan

Graph Neural Networks (GNNs): Comparison between CNNs and...

Based on my previous story on message passing framework for node classifications, I...

★ · 7 min read · May 30

2



Maxime Labonne in Towards Data Science

The Art of Spending: Optimizing Your Marketing Budget with...

Introduction to CVXPY to maximize marketing ROI

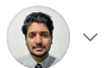
★ · 9 min read · May 23

205 2



Search Medium

Write



Lists

**Staff Picks**

341 stories · 96 saves

**Stories to Help You Level-Up at Work**


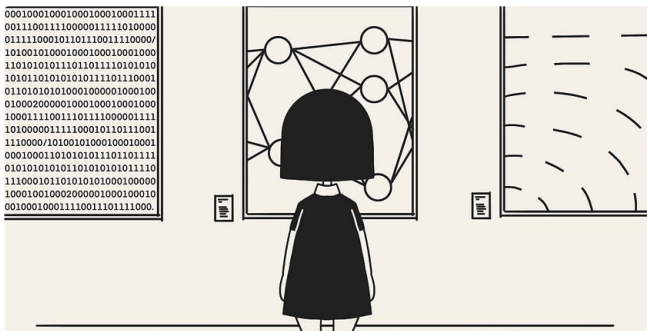
19 stories · 76 saves

**Self-Improvement 101**

20 stories · 135 saves


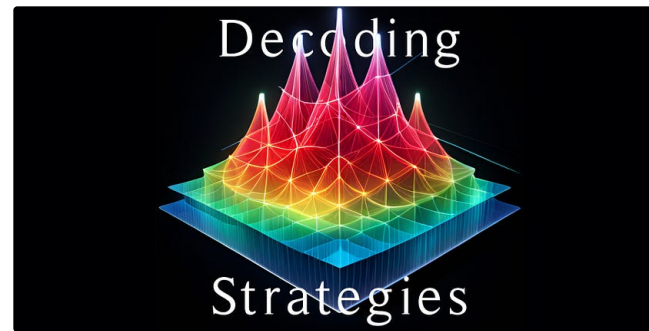
**Productivity 101**

20 stories · 147 saves

 Leonie Monigatti in Towards Data Science**10 Exciting Project Ideas Using Large Language Models (LLMs) f...**

Learn how to build apps and showcase your skills with large language models (LLMs). Ge...

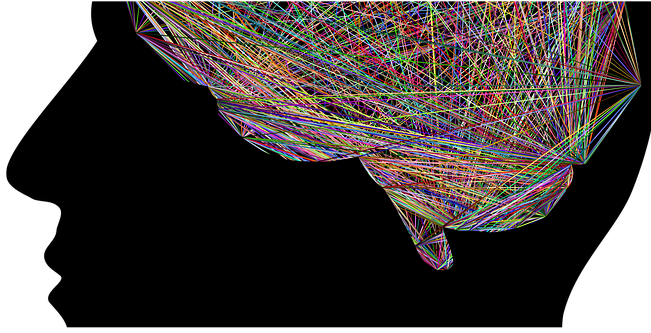
★ · 11 min read · May 15

 1.1K  7 ... Maxime Labonne in Towards Data Science**Decoding Strategies in Large Language Models**

A Guide to Text Generation From Beam Search to Nucleus Sampling

★ · 15 min read · 3 days ago

 264  3 ...



AI TutorMas... in Artificial Intelligence in Plain Eng...

Graph Neural Networks— Introduction for Beginners

A Graph Neural Network (GNN) is a type of neural network that is designed to work with...

★ · 10 min read · Jan 14



422



Nazlı Alagöz in Towards Data Science

Crossing the Bridge: A Comparison of Data Science in Academia and...

A Ph.D. student's exploration of the surprising parallels between academic and industrial...

★ · 8 min read · May 29



225



2



See more recommendations