Vrije Universiteit Amsterdam

Bachelor Thesis

MPQE Training on Entity Types

Author: Ken Mikovíny (2642384)

supervisor: Dr. Michael Cochez
second reader: Daniel Daza

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

August 3, 2021

Abstract

Knowledge graphs, also known as semantic networks, represent a structured way of organizing and storing information and real world knowledge of various entities including objects, events, situations and even abstract concepts. This information is stored in the form of semantic triples in triple databases. To be able to learn from these graphs we employ Graph Convolutional Networks , this semi-supervised learning model is used to produce vector embeddings for every node in a graph which then represent points in a multi-dimensional embedding space.

In this paper we will be focusing on knowledge graph information retrieval in the form of query answering. Queries can be created in a structured manner by various query languages for example SPARQL. To answer these queries we introduce a model based on previous work (Daza and Cochez, 2020) consisting of a Relational Graph Convolutional Network(Schlichtkrull et al., 2017) combined with the Message Passing Query Embedding approach that represents and embeds queries as small graphs. In our experiments we show how training the model on entity type embeddings instead of individual entity embeddings improves the training convergence speed of the model while maintaining the quality of the final results. We also address the drawbacks of this approach including lower performance on large datasets as well as high variance in terms of results which is tied to hyperparameter settings and the experimental setup.

1 Introduction

Embedding the data found in knowledge graphs is done by means of a neural network architecture called Graph Convolutional Networks. These networks produce node vector embeddings by effectively obtaining and combining information from neighboring nodes during each forward pass. During training, node embedding algorithms often group similar entities together in the embedding space while moving dissimilar entities further away, the end result is an embedding space of multiple cluster-like structures that are formed by related entities, however, this is not always the case for Graph Convolutional Networks.

In our experiments we will be working with Relational Graph Convolutional Networks (Schlichtkrull et al., 2017) that are additionally able to deal with different relations between individual entities (nodes) of a knowledge graph as highly multi-relational data is very common in real world knowledge bases. To generate answers to structured queries we firstly represent and train the query as a small embedded graph by utilizing Message Passing Query Embedding (MPQE) (Daza and Cochez, 2020). Analyzing the position of the trained query embedding in relation to the trained individual entity embeddings in the high dimensional vector space then allows us to determine answers to the queries.

1.1 Experiment focus

Models like the MPQE require a substantial amount of generated queries to train on to achieve good results. We attempt to reduce the amount of training needed while still maintaining good results by focusing on the fact that entities which share the same type (class) generally share multiple characteristic and have similarities in terms of features and shared edges, consequently, they often end up in the same clusters in the final graph embeddings produced by a RGCN/GCN. Following these observations we compare the performance of 1. a MPQE model (Daza and Cochez, 2020) trained on entity embeddings to 2. a MPQE model that we first pre-train on type embeddings in phase one and afterwards transfer the neural network weights to a new model that trains on entities exactly like model 1 in phase two. Our goal is to investigate if a model that begins training on entity embeddings that had been "pre-clustered" as opposed to having randomly initialized embedding vectors will effectively increase convergence speed while retaining the quality of the results obtained and the capability of answering queries. The reduction of the training that has to be done by the model is due to the fact that the network now only has to store and train type embeddings instead of carrying this out for all individual nodes of the graph which equates to a substantially smaller amount of embeddings.

2 Related Work

Our work builds upon the idea of learning low-dimensional embeddings of knowledge graphs by means of query answering introduced by (Hamilton et al., 2018) demonstrating how to map a practical subset of logic to efficient geometric operations in an embedding space. This approach then paved the way for the models related to our work. The model used in our experiments is based on the MPQE model presented by (Daza and Cochez, 2020) which uses RGCN (Schlichtkrull et al., 2017) layers to propagate information across a graph. Concepts related to both of these models include message passing networks which are described more closely in (Gilmer et al., 2017) as well as Graph Neural Networks (Scarselli et al., 2009) and Graph Convolutional Networks which were first introduced in (Kipf and Welling, 2017). Related work also includes research done on prototypical networks (Snell et al., 2017) which use a small number of examples for each entity type in the graph to train the network. The goal of this approach is to deal with few-shot classification where the model must be capable of classifying new entity types (classes) that have not been observed during training. This generalization process is done by comparing the embedding of the unknown entity type to existing trained embeddings that represent each known type (prototypes). In our experiments we employ a similar concept where we use the trained type embeddings we obtain as prototypes for different entity classes (types).

3 Definitions

3.1 Knowledge Graphs

Knowledge graphs (Hogan et al., 2021), also known as semantic networks, represent a structured way of organizing and storing information and real world knowledge of various entities including objects, events, situations or even abstract concepts. These graphs often contain information made up of datasets from many different sources that can also often have different structures. To be able to encode all of this different information together we are in need of ontologies. Ontologies ultimately form the backbone of a knowledge graph by providing formal semantics that aid with expressing and interpreting the data of a graph. In the case of knowledge graphs, these formal semantics are represented by:

Classes: Entities of knowledge graphs are represented by nodes. Entity classes serve as a description and a means of classification in terms of a higher class hierarchy.

Relation types: An edge between two nodes (entities) of a knowledge graph is paired with an edge label. These labels serve as a description of the relationship between two entities. (e.g. child-of, friend-of, class-mate-of, co-worker-of)

Categories: Entities of the graph can be a part of different (or even multiple) categories (e.g. a professor that is a part of a specific research group or a student that is a member of a specific student association.)

Free text descriptions: These can optionally be included and take the form of notes that provide further clarification to a subject or a part of the knowledge graph.

3.2 Structured Queries

To be able to carry out information retrieval on a given knowledge graph we make use of structured queries in *conjunctive form* as seen in (Hamilton et al., 2018). These queries are formed by a conjunction of binary predicates that consist of entities or variables as their arguments. Consider for instance a knowledge base of an academic institution, an example query can look like: "select all projects P, such that topic T is related to P, and both *alice* and *bob* work on T." In other words, we search for entities P that satisfy the following condition:

$$P.\exists T, P : \text{related}\,(P, T) \wedge \text{works_on}\,(alice, T) \wedge \text{works_on}\,(bob, T)\,.$$

A query q is defined by a condition on a *target variable* V_t in the following way:

$$q = V_t.\exists V_1, \ldots, V_m : r_1(a_1, b_1) \wedge \ldots \wedge r_m(a_m, b_m).$$

where $r_i \in \mathcal{R}$ and a_i and b_i are either entities in the knowledge graph, or query variables

in $V_t, V_1, \ldots, V_m$. Following this definition, an entity is considered an answer to the given query if it satisfies this condition (Daza and Cochez, 2020).

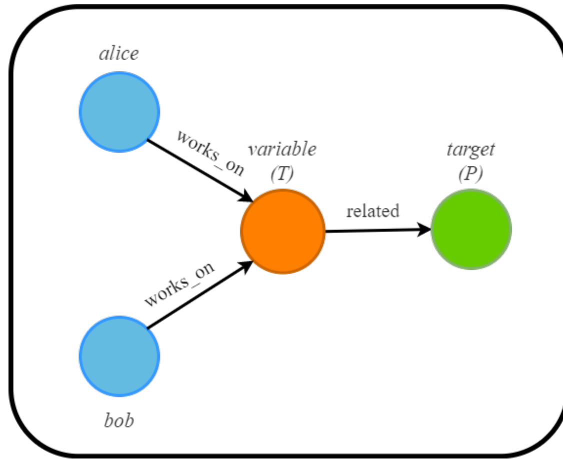


Figure 1: An example of a query graph derived from the example described above (Daza and Cochez, 2020). Alice and bob represent entity nodes (blue), topic represents a variable node (orange) and project represents a target of the query (green).

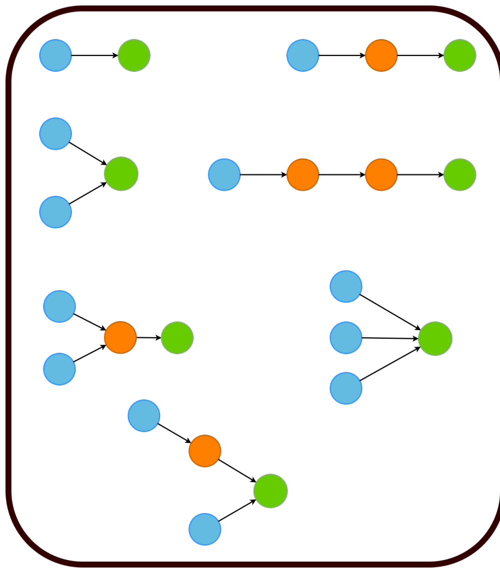


Figure 2: Query structures considered when generating and answering queries (Daza and Cochez, 2020), blue nodes represent entities and the rest corresponds to variables in the query with the green nodes being the targets. The names of the structures from left to right, top to bottom: 1hop, 2hop, 2i, 3hop, 2i-1hop, 3i, 1hop-2i. With "hop" denoting edge hops and "i" denoting intersections.

3.3 Query Answering

When answering a query our model uses a message passing query embedding (Daza and Cochez, 2020) model to generate embeddings of the graph entities as well as the query targets and variables. Once trained, we use the dot product of the generated query embedding vector to calculate a score matrix between the query and all other entities in the graph to effectively determine how similar they are to each other in the multi-dimensional embedding space generated by the model. The entities and the generated query embeddings are then ranked

based on these similarity scores. The entities with the highest score are then considered as the answers to the query.

3.4 Graph Convolutional Networks

A Graph Convolutional Network (GCN) is a form of semi-supervised learning model derived from convolutional networks with the difference being that GCN's can operate directly on graphs. In general, the network takes as input:

- A summarized description of the individual node features (In our case these are node embeddings)
- A representative description of the graph (In our experiments this is an edge index as well as a matrix containing edge types).

The goal of these networks is to propagate information across a graph, i.e. encode both local graph structures as well as features of individual input nodes and produce a graph embedding as output. Similar to classic neural networks, a GCN is formed by several neural network layers that share a common layer-wise propagation rule which can be generalized to:

$$H^{(l+1)} = f\left(H^{(l)}, A\right) \tag{1}$$

This function represents a form of message passing where information about graph nodes gets propagated along graph edges. During each step, every node effectively gets features from neighboring nodes while updating its own features accordingly.

Different choices of f result in different variants of the GCN model, for instance the model introduced by (Kipf and Welling, 2017) uses the following function:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) \tag{2}$$

$\tilde{A}$ represents the normalized adjacency matrix of an undirected graph $\mathcal{G}$ that is created by adding self loops to the original adjacency matrix and subsequently summing it with an identity matrix of the same size ($\tilde{A}$ = A + I_N). $\tilde{D}$ represents the diagonal matrix produced from $\tilde{A}$ ($\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$). W(l) is a trainable weight matrix and σ is a chosen activation function (for instance ReLU(·) = max(0,·).

A simple multiplication of the weight matrix with the normalized adjacency matrix during propagation would lead to a loss of control over the range of the hidden layer output, the model would then face the exploding or the diminishing problem of a neural network. To combat this we use the renormalization trick of multiplying the normalized adjacency matrix with the diagonal matrix $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ as described by (Kipf and Welling, 2017).

3.5 Relational Graph Convolutional Networks

In our experiments we are going to be using an extension of GCN's called a Relational Graph Convolutional Network (R-GCN) (Schlichtkrull et al., 2017). This modified version of the model allows us to handle different relationships between entities during information propagation by using different weight matrices for each different edge type. During the forward pass, the R-GCN computes the hidden representations of the (l+1)th layer as follows:

$$h_i^{(l+1)} = \sigma \left(\sum_{r \in \mathcal{R}} \sum_{j \in \mathcal{N}_i^r} \frac{1}{c_{i,r}} W_r^{(l)} h_j^{(l)} + W_0^{(l)} h_i^{(l)} \right) \tag{3}$$

Where $h_i^{(l)} \in \mathbb{R}^{d^{(l)}}$ denotes the hidden state of a node v_i in the l-th layer of the R-GCN network, $d^{(l)}$ represents the dimensionality of the representations of this layer. In our experiments the dimensionality of the input embeddings remains preserved. $\mathcal{N}_i^r$ represents the set of neighbor indices of node i under relation $r \in \mathcal{R}$. $c_{i,r}$ is a normalization constant (chosen to be $c_{i,r} = |\mathcal{N}_i^r|$ in our case). Additionally, different edge types use different weights and only the same relation type r is associated with the same weight matrix $W_r^{(l)}$. We use the RGCNConv Pytorch Geometric implementation (Fey and Lenssen, 2019) that is based on the work done by (Schlichtkrull et al., 2017). In our experiments we do not use the basis-decomposition regularization nor the block-diagonal-decomposition regularization scheme.

4 Model Description

We will be using the tuple definition of a Knowledge Graph introduced in (Daza and Cochez, 2020) $(\mathcal{V}, \mathcal{E}, \mathcal{R}, \mathcal{T})$, with $\mathcal{V}$ representing a set of entities of a given dataset with $\mathcal{E}$ representing typed edges between individual entities (nodes). A function $\tau : \mathcal{V} \rightarrow \mathcal{T}$ that assigns a type to every node, where $\mathcal{T}$ is a set of entity types. Additionally, we denote $r(v_i, v_j)$ a relation between two nodes of a graph(v_i and v_j) where $r \in \mathcal{R}$ represents a relation (edge) type.

The two models used in our experiments both follow the work done by (Daza and Cochez, 2020), two Relational Graph Convolutional Network layers (RGCN)(Schlichtkrull et al., 2017) are used to propagate information across the graph while accounting for node neighbors and edge types. During this process the network trains on a input matrix of shape $M_n \in \mathbb{R}^{|\mathcal{V}| \times d}$ where each row contains the embedding of one node of the query and d represents the dimension of the embedding space. The generated query embedding is then used by the loss function during training together with all of the entity embeddings of a given dataset.

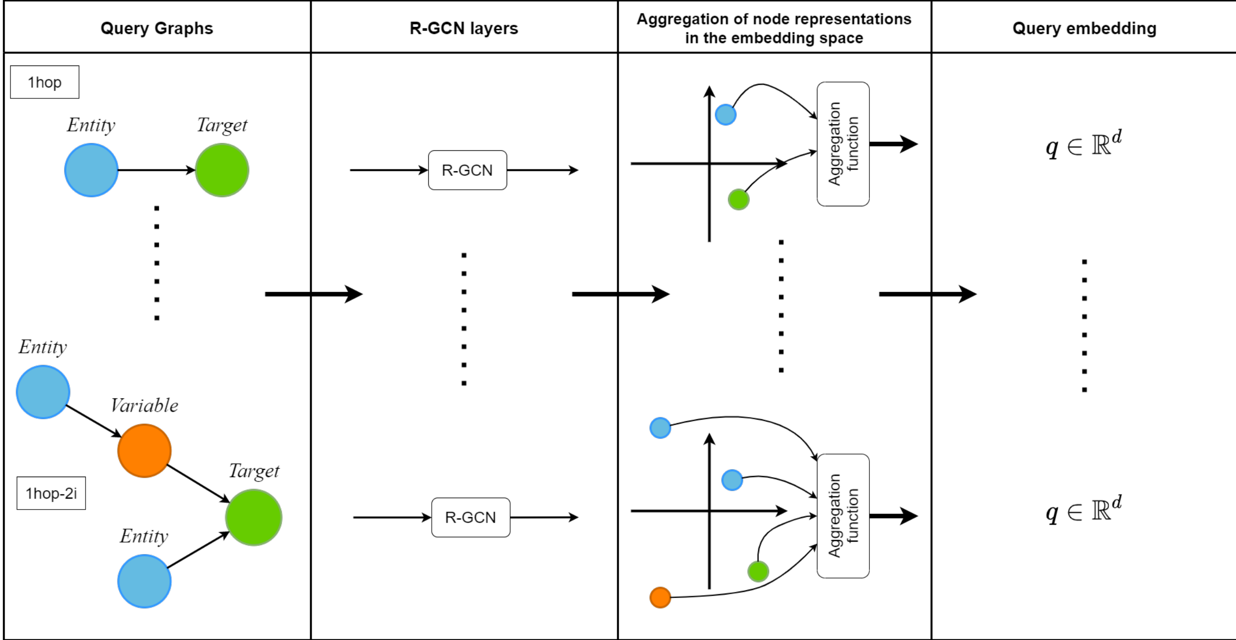


Figure 3: Graphic representation of MPQE. The R-GCN of the model takes as input a query graph (1hop and 1hop-2i in the figure only serve as illustrative examples, this concept applies to any query structure) in the form of a matrix of node embeddings, these embeddings are then updated accordingly during forward passes and backpropagation of the neural network layers. The output of the convolutional layers is then pooled together by means of an aggregation function to produce a vector embedding of the query.

4.1 Node Embeddings

To allow a graph to be embeded in a vector space the individual graph entities are represented as vectors of size d (dimension of the embedding space). These vectors represent features of each node and are then taken as input to the R-GCN layers of our model while being saved and trained throughout the backpropagation process. The model solely trained on entities (Model 1) begins training with randomly initialized embeddings sampled following the Xavier initialization which is very popular choice when it comes to training a feedforward neural network as it helps maintain near identical variances of weight gradients across layers as seen in (Glorot and Bengio, 2010). This results in a evenly dispersed set of embeddings that were sampled from a random uniform distribution. Model 2 uses the identical Xavier initialization approach for generating initial type embeddings.

4.2 Loss function

Our model uses a Binary Cross Entropy Loss function implemented by (Alivanistos et al., 2021) which is based on the Pytorch BCEWithLogitsLoss function (Paszke et al., 2019). BCELoss creates a criterion that measures the binary cross entropy between the target and the output which can be described by the following expression:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n))\right]$$

where N is the batch size. The loss function also adds a Sigmoid layer to the binary cross entropy. Combining the operations into one layer leads to the operation being more numerically stable by taking advantage of the log-sum-exp trick. The loss function takes as input a matrix of pair-wise similarity scores between the query embeddings and all other entity embeddings and a list of query targets of a given batch. The matrix scores are generated by means of a dot product similarity function (Alivanistos et al., 2021). Stochastic gradient descent is replaced by the Adam optimizer (Kingma and Ba, 2017) which combines the Adaptive Gradient Algorithm (AdaGrad) and the Root Mean Square Propagation (RMSProp), both of these maintain per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight. In the field of deep learning Adam is a very popular algorithm because it achieves considerably good results quickly.

4.3 Aggregation Function

After passing the node embeddings of a query through two RGCN layers the output embeddings get aggregated by a simple pooling function that sums the nodes together to produce a single vector of dimension d. The pooled output then represents the embedding of the given query.

4.4 Model 1: Entity MPQE Model

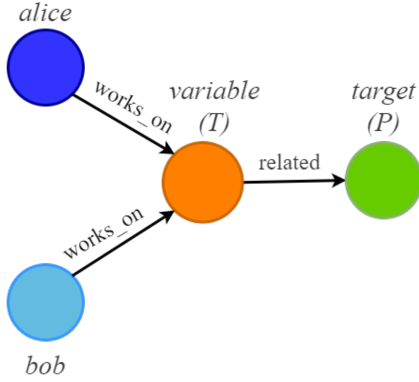
Given a query graph, we start off by initializing each of the nodes of the query with a vector representation, this matrix of embeddings is then stored by the model throughout the training process. Parts of this matrix then serve as the input to the model and are used to train all of the stored embeddings. Particularly for our first model this is a matrix of the shape $M_e \in \mathbb{R}^{|\mathcal{V}| \times d}$ with each row of the matrix containing the embedding of a single *entity* of a given query. $\mathcal{V}$ represents number of entities of the query and d the dimension of the embedding space. The model then proceeds with an application of the RGCN layers and aggregation function to obtain a query embedding and subsequently calculate the loss and update the vector representation of different nodes of the graph.

4.5 Model 2: Combined MPQE Model

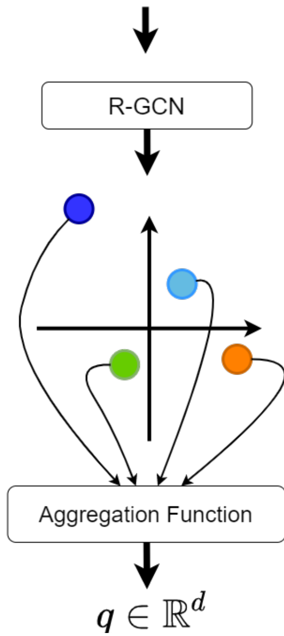
The second, Combined MPQE Model used in our experiments uses the exact same MPQE implementation as Model 1 with the only difference being that instead of initializing entity

embeddings randomly we first randomly initialize the embeddings of a type matrix $M_t \in \mathbb{R}^{|\mathcal{T}| \times d}$. We then proceed to train the network while only considering the types of entities (as illustrated in Figure 4). Afterwards, we initialize a matrix of entity embeddings $M_e \in \mathbb{R}^{|\mathcal{V}| \times d}$ by taking the vector embedding of the corresponding type from $M_t \in \mathbb{R}^{|\mathcal{T}| \times d}$ which results in an identical vector representation being assigned to entities that share the same type before training begins. The process of transfering weights between the two models is described in more detail in the upcoming sections. Afterwards, we recommence training the network in the same way as described in the previous subsection.

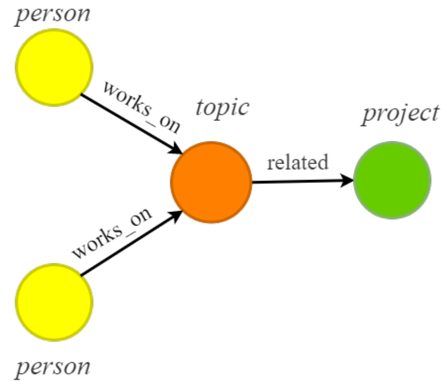
Training on entities



entity id	vector representation
0 (alice)	$x_{0\,0}, x_{0\,1}, \ldots, x_{0\,d}$
1 (bob)	$x_{1\,0}, x_{1\,1}, \ldots, x_{1\,d}$
2 (variable)	$x_{2\,0}, x_{2\,1}, \ldots, x_{2\,d}$
3 (target)	$x_{3\,0}, x_{3\,1}, \ldots, x_{3\,d}$



Training on types



type id	vector representation
0 (person)	$x_{0\,0}, x_{0\,1}, \ldots, x_{0\,d}$
0 (person)	$x_{0\,0}, x_{0\,1}, \ldots, x_{0\,d}$
1 (variable)	$x_{1\,0}, x_{1\,1}, \ldots, x_{1\,d}$
2 (target)	$x_{2\,0}, x_{2\,1}, \ldots, x_{2\,d}$

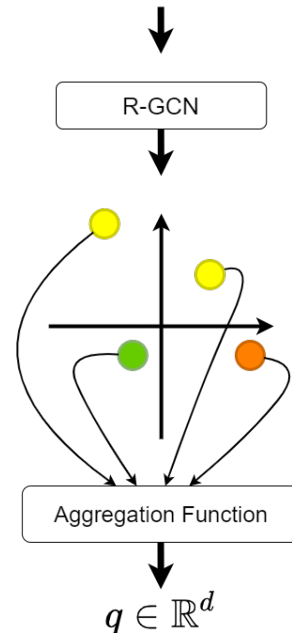


Figure 4: Comparison of the MPQE Model 1 that trains on entity embeddings (left) and the part of the Comibed MPQE model 2 that only trains on types instead (right).

5 Experiments

Figures throughout the rest of this document will be referring to Model 1 as "Entity MPQE Model" and Model 2 as "Combined MPQE Model". The experiments have been conducted three times for each dataset (except AM, which is addressed in the Discussion section).

5.1 Datasets

The datasets that were used to generate queries for our experiments are publicly available knowledge graphs previously used in (Daza and Cochez, 2020; Hamilton et al., 2018; Ristoski et al., 2016):

AIFB
: Knowledge base describing the AIFB research institute in terms of its staff, research group, and publications. Entities are of the following types: "class", "organization", "person", "topic", "project" and "publication".

AM
: Dataset containing information about artifacts in the Amsterdam Museum. Entities are of type: "web resource", "aggregation", "physical thing", "proxy".

MUTAG
: A knowledge base of complex molecules, some of which potentially carcinogenic. Entities are of type: "structure", "bond", "atom", "compound".

For a more detailed description of the datasets refer to (Ristoski et al., 2016)

Dataset	AIFB	AM	MUTAG
Entities	2601	372584	22372
Relation Types	19	11	4
Entity Types	6	4	4

Table 1: Overview of relevant data of the subset of the datasets that were part of our generated queries that were used in our experiments.

5.2 Experimental Setup

We use an embedding dimension d of 128 for initialization of entity and type embeddings which then get passed through 2 R-GCN layers which have their in_channels and out_channels set to the embedding dimension d. As described in previous sections, to calculate loss we use dot product similarity as a scoring function and binary cross entropy as the loss function combined with an Adam optimizer with the learning rate set to 0.0001 and weight decay of .0005.

Model 1 (Entity MPQE Model) trains for 1000 epochs while Model 2 (Combined MPQE Model) firstly trains for 100 epochs solely on types. After transfering the trained weights and re-initializing the entity embeddings of Model 2 with their corresponding trained type embedding we proceed to train for 1000 epochs on entities only. The performance of Model 1 and Model 2 during their entity training phases (1000 epochs) is then used to compare the two models and draw results of our experiments. We use a batch size of 32 queries during training. The queries we used are allowed to have multiple targets, however during training on types duplicate targets were removed (i.e. targets that shared the same entity type and therefore were representing the same embedding) so we could continue to use the same loss function when training on entities as well as types.

18% of our dataset was removed from the graph for validation and testing purposes before sampling the queries that were used during the training process.

Training on Types To be able to train a model on types instead of entities the following steps were taken: Firstly, each model was now training a matrix of embeddings of shape $M_t \in \mathbb{R}^{|\mathcal{T}| \times d}$ instead of $M_e \in \mathbb{R}^{|\mathcal{V}| \times d}$ as described in section 4.6. Secondly, for each dataloader batch of queries entity ID's were remapped to their corresponding type ID's, this was done for the entity_ids, edge_index tensors as well as the tensor which contained query targets of the current batch. These remapped tensors were then used as input to our MPQE model.

5.3 Query Sampling

The code for the query generation process we have used in our experiments can be found in (Alivanistos et al., 2021). We use Docker[1] (Merkel, 2014) to deploy the AnzoGraph[2] database as our triple store. Afterwards, we generate queries in textual form in the SPARQL query language[3]. The queries are then converted into a more compact binary format, the binary files are then used to generate dataloaders used in our experiments.

Our datasets consist of up to 1000 samples of each query structure type seen in Figure 2. However, each dataset contains a different amount of queries per query structure type and in some cases they can contain less than 1000 or even 0 queries of a given type. For instance the MUTAG dataset does not contain 3hop and 2i-1hop queries. It is important to consider these facts when drawing conclusions from our experiments.

5.4 Visual Exploration

Each node embedding vector represents a point in the embedding space, these vectors are then used by the MPQE model during the query answering training process. We also observe

[1]https://docs.docker.com/

[2]https://www.cambridgesemantics.com/anzograph/

[3]https://www.w3.org/TR/sparql11-query/

that the MPQE model tends to cluster individual nodes in the embedding space according to their types without explicit supervision as can be seen in Figure 5.

T-SNE We visualize node embeddings using t-SNE which is a technique for visualizing high dimensional data. More specifically, we use the implementation of (Ulyanov, 2016) which is based on the modified t-SNE algorithm developed by (van der Maaten, 2014). This is a modified version of the algorithm which achieves acceleration and better performance by using two tree-based algorithms: Barnes-Hut algorithm and of the dual-tree algorithm. The t-SNE algorithm was run for 1000 iterations with perplexity set to 30.

PCA Before inputting the embeddings to t-SNE we use the sklearn implementation of the Principal Component Analysis algrithm (PCA) (Buitinck et al., 2013) to linearly reduce the dimensionality of the node embeddings to significantly reduce the running time and computational intensity of the visualization process.

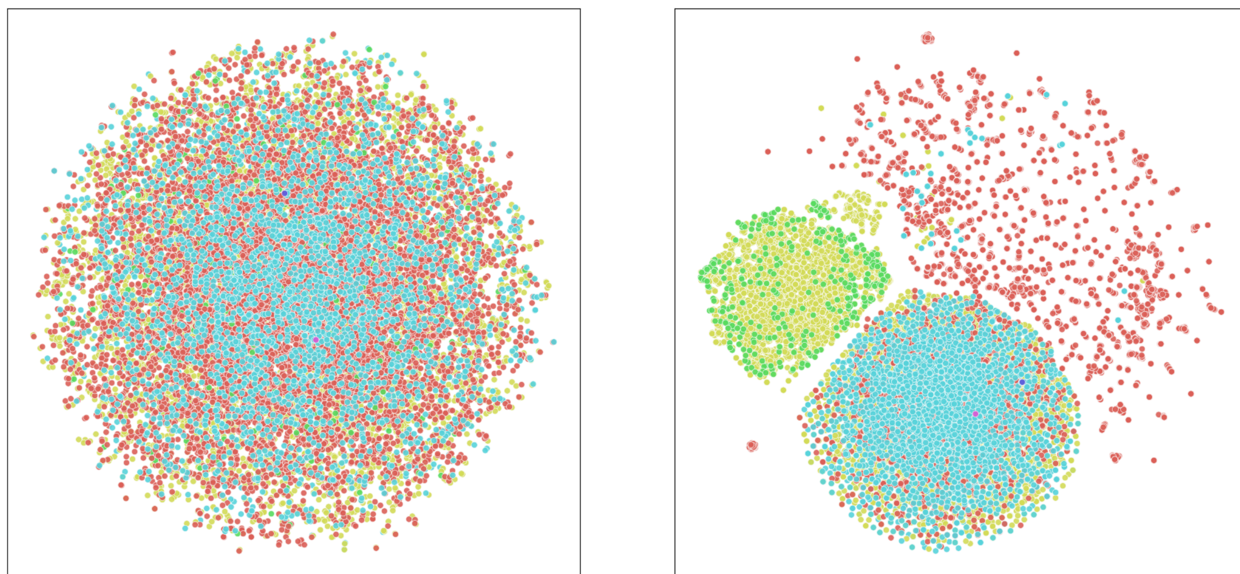


Figure 5: Visualization of how embeddings change during 1000 epochs of MPQE Model 1 training:
On the left, randomly initialized entity embeddings of the MUTAG dataset before training. On the right, entity embeddings learned by MPQE during training. Nodes are colored according to their individual entity types.

5.5 Code

The code in our experiments is based on the MPQE (Daza and Cochez, 2020) and HQE (Alivanistos et al., 2021) models. The code is written in Python 3.8.5 together with PyTorch

1.9.0 and cuda 10.2 and can be found on Github[4]. A large part of the experiments were performed using the DAS5 system of clusters[5] (Bal et al., 2016). Results were evaluated using the implementation of (Alivanistos et al., 2021) that uses the Pykeen Rank Based Evaluator (Ali et al., 2021) as its base.

6 Results

In our results section we will be focusing on the AIFB dataset, the results of experiments conducted using the MUTAG and AM knowledge bases are included in the Appendix. Each figure illustrates 3 runs of the experiment with the blue and orange areas representing standard deviation across the runs.
We compare the performance of the two models by analysing the following metrics:

Average mean rank

The rank of a query is the position on which a query target can be found in the ranked pair-wise scores matrix. The mean rank (MR) computes the arithmetic mean over all individual ranks. It is given as:

$$\text{score} = \frac{1}{|\mathcal{I}|} \sum_{r \in \mathcal{I}} r$$

The advantage of this metric over hits @ k is that it considers the entire knowledge graph not just a subset of top k results. Consequently, the evaluation metric reflects the average performance. It has the advantage over hits @ k that it is sensitive to any model performance changes, not only what occurs under a certain cutoff and therefore reflects average performance. With PyKEEN's standard 1-based indexing, the mean rank lies on the interval $[1, infinity)$ where lower is better.

Hits @ K

After calculating individual pair-wise scores between the query embedding and all other entities in the graph we proceed to analyze the number of hits. The hits @ k describes the fraction of true entities (targets) that appear in the first k entities of the sorted rank list. It is given as:

$$\text{score}_k = \frac{1}{|\mathcal{I}|} \sum_{r \in \mathcal{I}} \mathbb{I}[r \leq k]$$

For example, if we consider the 10 nodes that are the closest to a query embedding in the embedding space. Then the percentage of results that are the true targets of the query is the hits @ 10. The hits @ k, regardless of k, lies on the $[0, 1]$ where closer to 1 is better.

In our analysis of the results we will be considering :

[4]https://github.com/KenMikoviny/BachelorThesisKen
[5]https://www.cs.vu.nl/das5/home.shtml

- Average hits at 3
- Average hits at 10
- Worst hits at 3
- Worst hits at 10

During evaluation we use the filtered rank (Bordes et al., 2013) between the query embedding and other entities. This is achieved by ignoring all other correct scores while computing the rank of a given correct entity. For additional information about the evaluation metrics refer to the PyKEEN documentation[6].

6.1 Convergence Speed

The evaluation metrics discussed above are plotted for both Model 1 & Model 2 and compared together over 1000 epochs of training on entities. Figure 6 shows that the Combined MPQE Model converges noticeably faster compared to the Entity MPQE Model. During the final 10 epochs Model 1 finishes training by obtaining an average mean rank of ≈ 430 where as Model 2 reaches a rank of ≈ 416. However we observe that Model 2 reaches the rank of ≈ 430 around epoch 250 which is approximately 350 epochs faster than it takes Model 1 to reach the same average mean rank value. The change in convergence speed is not by any means colossal nevertheless it illustrates how improvements can be achieved with this approach. Proper automated hyperparameter tuning and ideal model settings would be necessary to achieve optimal and significant results. We observe similar results for the MUTAG and AM datasets as can be seen in the Appendix.

[6]https://pykeen.readthedocs.io/en/latest/tutorial/understanding_evaluation.html

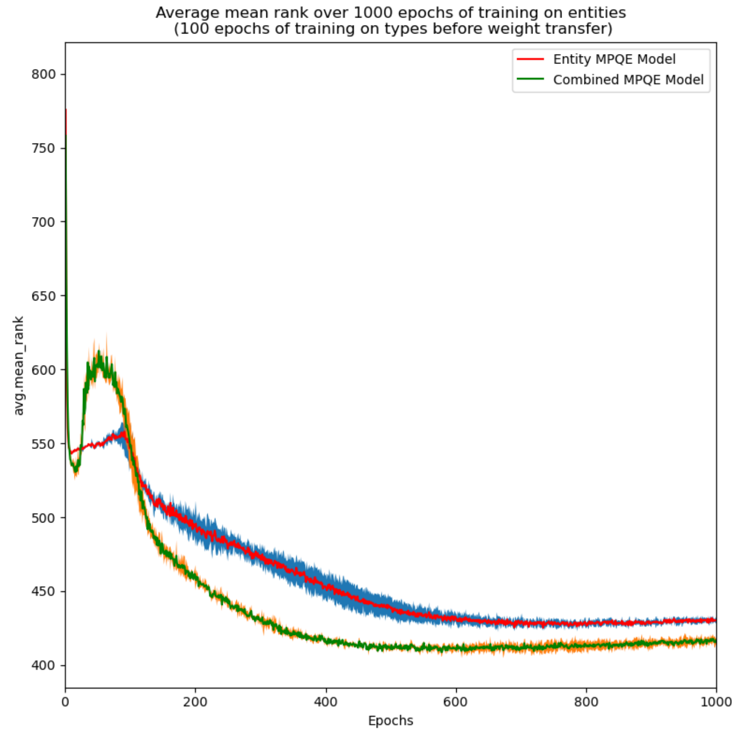
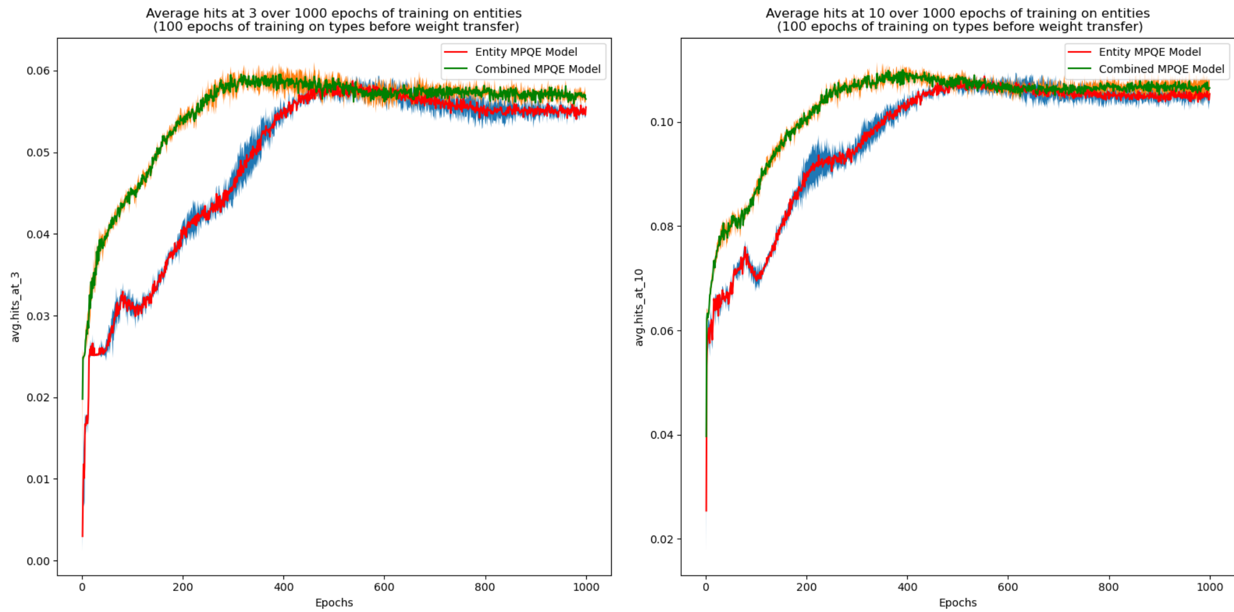


Figure 6: Average mean rank.

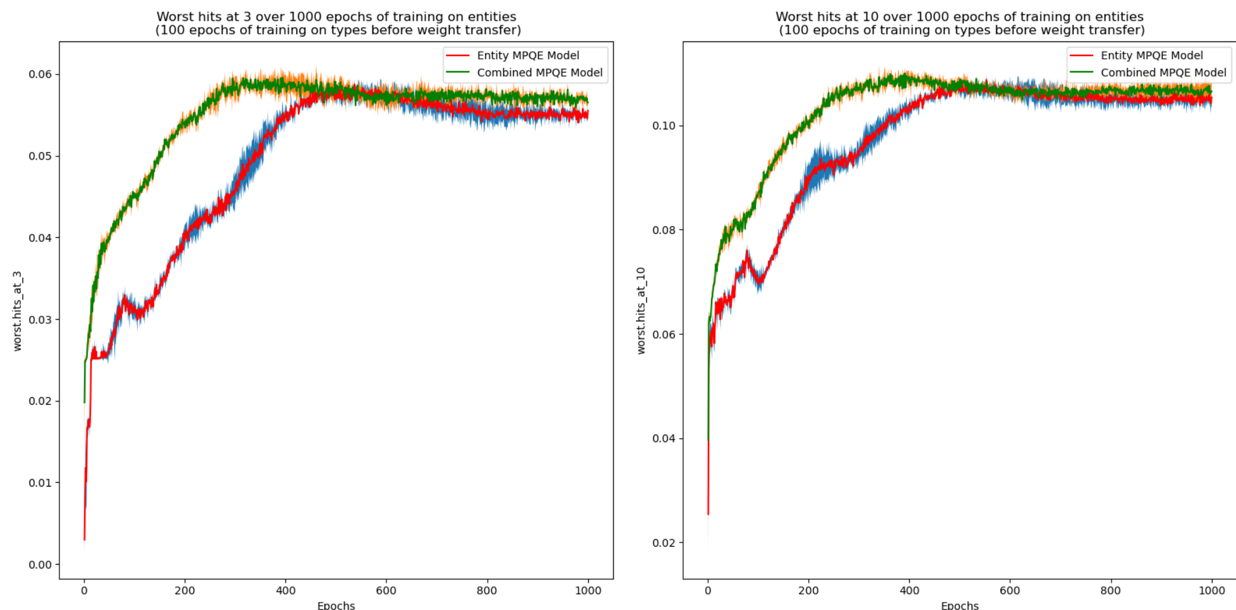


(a) Average hits at 3.

(b) Average hits at 10.

Figure 7: Average hits @ K.

Observing Figure 7 we see that in both cases the Combined MPQE Model improves convergence speed of the training process. In Figure 7a we see that the Combined model already begins to plateau around period 250 where as the Entity MPQE model achieves the same number only after 200 more epochs of training. We detect a very similar result for average hits at 10 (Figure 7b). We also observe that the combined model peaks around epoch 250 and then starts to trend down, this is most likely a result of overfitting during the training process. As with the average mean rank, we speculate that these results can be significantly improved with proper model settings and hyperparameter tuning.



(a) Worst hits at 3.

(b) Worst hits at 10.

Figure 8: Worst hits @ K.

Figure 8 presents us with similar results, we notice that there is a noticeable convergence speed increase when taking into consideration the worst hits at 3 and 10 (i.e. fraction of targets that appear in the first k entities of the sorted rank list of the worst performing query).

All figures containing the results of our experiments are included in the Appendix.

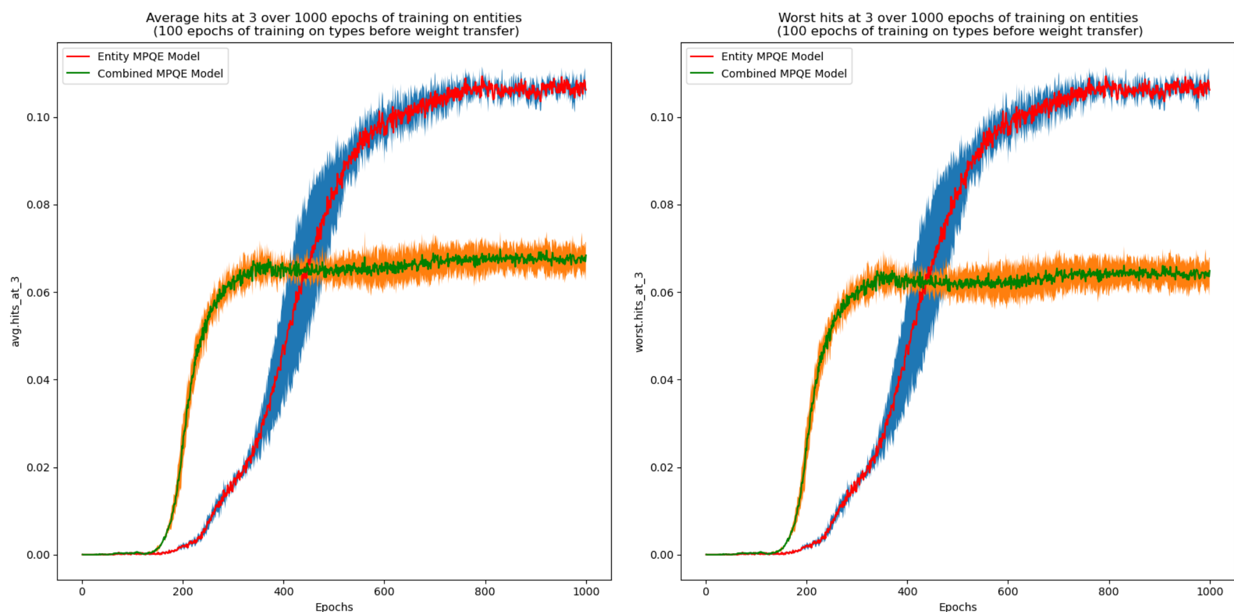
7 Discussion

Future Research Due to time constraints, the results obtained from our experiments were a result of manual hyperparameter tuning, an increase in convergence speed and improvements of the training process could be achieved by adopting a more automated and sophisticated approach. This also concerns other parts of the experimental setup including

how many epochs of type training there are before the parameter transfer is done between models as well as the random initialization of entity embeddings that occurs before training starts. Different variations of the model

Multiple Types per Entity Our model only considered entities with a single entity type (class) during the training and sampling process. An interesting area of future research includes extending this model to work with entities that have multiple types or labels assigned to them.

MUTAG Dataset Conducting the experiments using the MUTAG knowledge base resulted in abnormalities. More specifically Model 2 which was pre-trained on types reached a lower final performance (judged based on the chosen evaluation metrics described in Section 6). As we can see in Figure 9 Combined MPQE Model seems to reach a plateau instead of continuing to improve its performance, the figures for the remaining metrics are included in the Appendix. This might be due to the model reaching some kind of local optimum as a result of pre-training. During our tests we have found that increasing the number of sample queries, thus increasing the number of queries Model 2 trained on led to a performance increase for all of the evaluation metrics.



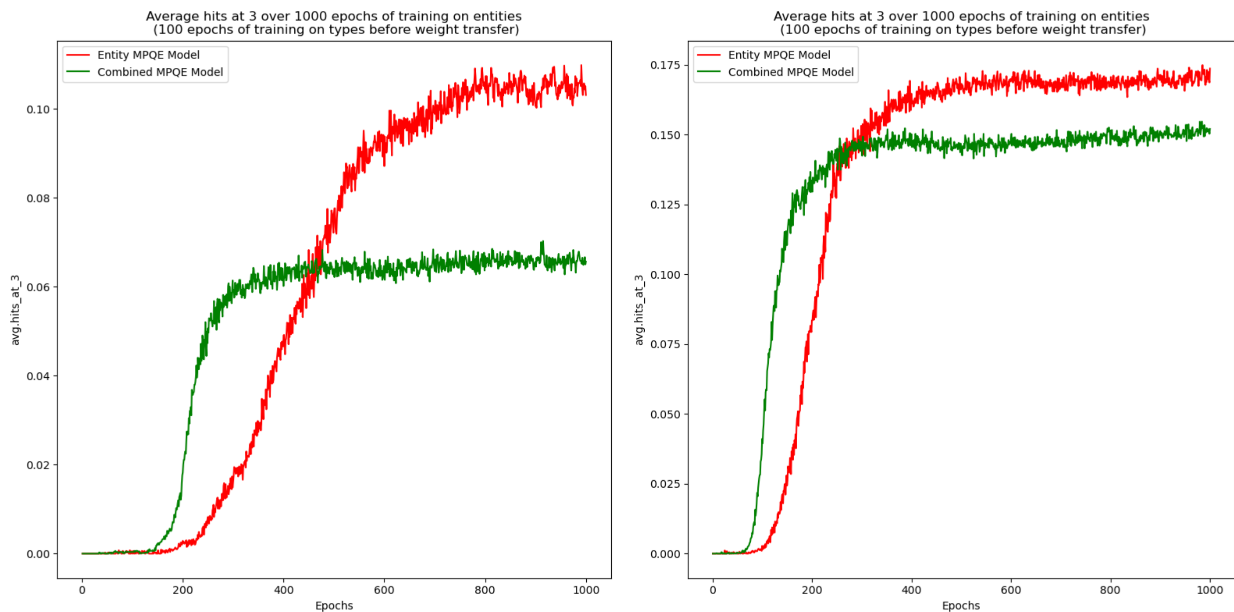
(a) Average hits at 3.

(b) Worst hits at 3.

Figure 9: Average hits at 3 & worst hits at 3 for the MUTAG dataset.

We found that increasing the number of queries sampled for each query structure as well as increasing the number of epochs of pre-training on types helped combat these issues. As can be seen in Figure 10 increasing the number of queries sampled for each query structure type

from 1000 to 3000 led to an increase of performance and a higher "ceiling" for the Combined MPQE Model compared to the Entity MPQE Model as well as a higher overall ceiling for both models. The figure uses the Average hits at 3 metrics to illustrate this phenomenon but a similar increase in terms of evaluation metric performance was also observed for all of the other metrics. This further underlines the need for hyperparameter tuning when aiming for good results with the MPQE training on types approach.



(a) 1000 query samples of each type.

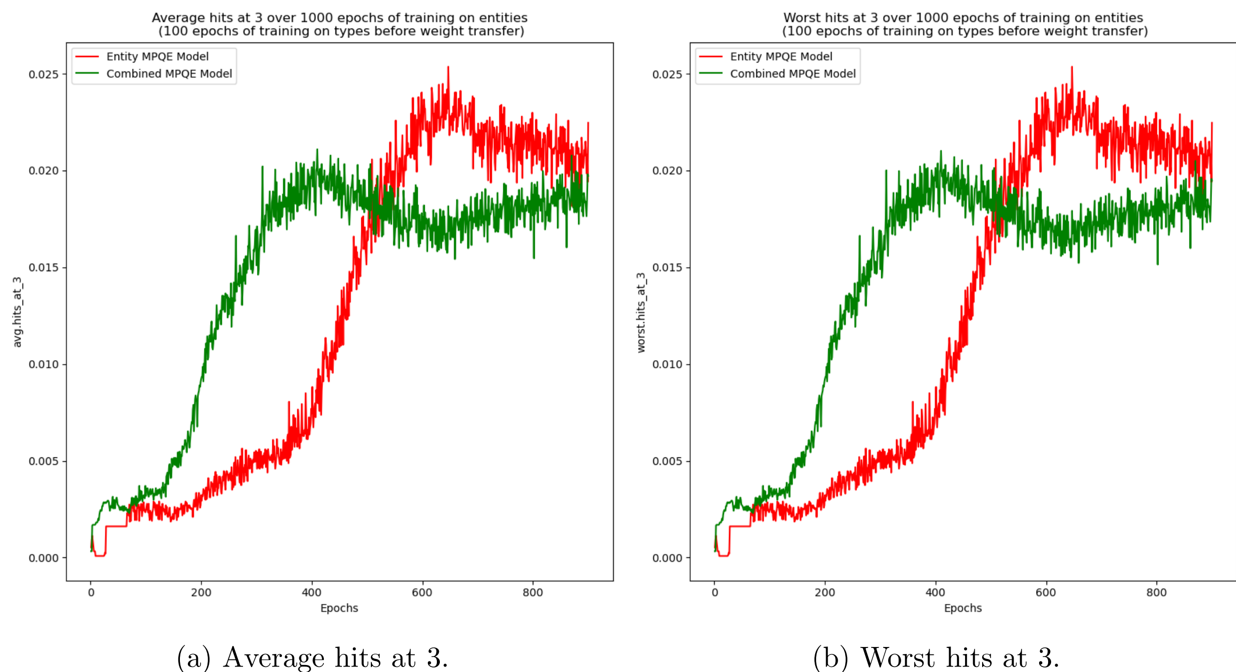
(b) 3000 query samples of each type.

Figure 10: Average hits at 3 comparison for the MUTAG dataset.

AM Dataset While examining the experiment results of tests conducted on the AM dataset we observe relatively poor results. The hits @ K metrics show that the combined model does converge faster however it reaches a lower performance ceiling compared to Model 1 as can be seen in Figure 11. The models performed especially poorly for the average mean rank metric, in Figure 12 we see that the models fail to minimize this metric during the training process.

These results are somewhat expected and are tied to the size of the AM dataset which is considerably larger than the other two datasets used in our experiments (see Table 1). We only sample a 1000 queries for each query structure type where as tens of thousands get generated in the case of the AM dataset, this amount of queries used to train the model is simply not enough in the case of a knowledge graph of this size. Furthermore, when calculating loss we consider all other entities in the graph, minimizing the loss becomes significantly more difficult with our approach. To combat these issues we emphasize the need for proper automated hyperparameter tuning as well as exploring different variants of the model and different number of queries sampled.

Due to time constraints, the AM dataset experiment was only run once and only features 900 epochs of training instead of 1000 as with the MUTAG and AIFB datasets.



(a) Average hits at 3. (b) Worst hits at 3.

Figure 11: Average hits at 3 & worst hits at 3 for the AM dataset.

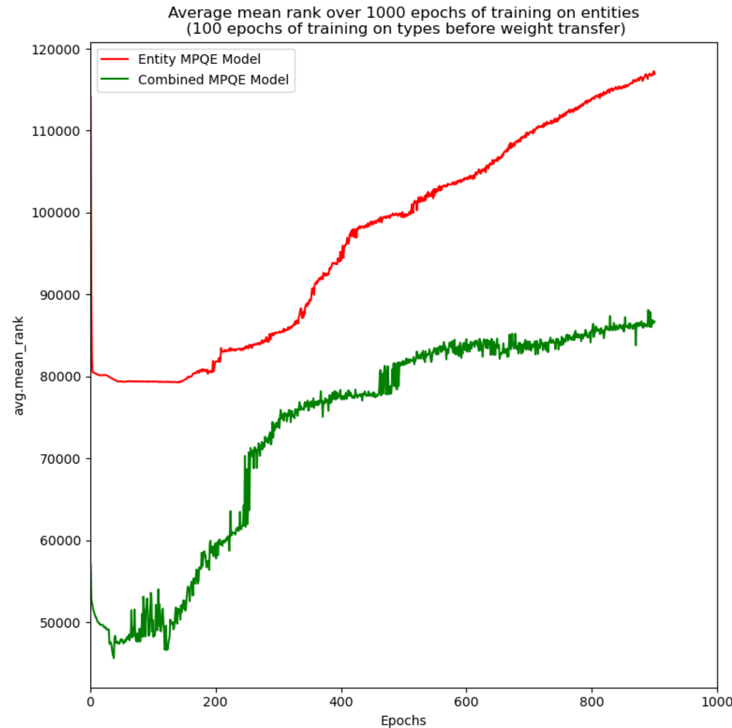


Figure 12: Average mean rank for the AM datset.

8 Conclusion

We have used a MPQE Model that draws its fundamentals from graph convolutions and neural networks to demonstrate how training on entity types instead of individual entities can potentially yield significant improvements in terms of training convergence speed. We have observed high variance in terms of results depending on hyperparameter settings, choice of dataset, number of sampled queries and pre-training epochs which emphasises the need for hyperparameter tuning and exploration of different setups for the model. Hence, the results of our research are by no means conclusive and offer space for further testing. Improvements can be achieved by optimization of the model as well as exploration of different variants of the model. This includes for example using a completely different loss function or introducing a different similarity score calculation system.

Another area that requires further investigation is applying this concept to different datasets as well as possibly considering different query structures as we have only used the ones specified in Figure 2.

References

Ali, M., Berrendorf, M., Hoyt, C. T., Vermue, L., Sharifzadeh, S., Tresp, V., & Lehmann, J. (2021). PyKEEN 1.0: A Python Library for Training and Evaluating Knowledge Graph Embeddings. *Journal of Machine Learning Research*, *22*(82), 1–6. http://jmlr.org/papers/v22/20-825.html

Alivanistos, D., Berrendorf, M., Cochez, M., & Galkin, M. (2021). Query Embedding on Hyper-relational Knowledge Graphs.

Bal, H., Epema, D., de Laat, C., van Nieuwpoort, R., Romein, J., Seinstra, F., Snoek, C., & Wijshoff, H. (2016). A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *Computer*, *49*(05), 54–63. https://doi.org/10.1109/MC.2016.127

Bordes, A., Usunier, N., Garcia-Durán, A., Weston, J., & Yakhnenko, O. (2013). Translating Embeddings for Modeling Multi-Relational Data. *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, 2787–2795.

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., & Varoquaux, G. (2013). API design for machine learning software: experiences from the scikit-learn project. *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 108–122.

Daza, D., & Cochez, M. (2020). Message Passing Query Embedding.

Fey, M., & Lenssen, J. E. (2019). Fast Graph Representation Learning with PyTorch Geometric. *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural Message Passing for Quantum Chemistry.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh & M. Titterington (Eds.), *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (pp. 249–256). PMLR. http://proceedings.mlr.press/v9/glorot10a.html

Hamilton, W., Bajaj, P., Zitnik, M., Jurafsky, D., & Leskovec, J. (2018). Embedding Logical Queries on Knowledge Graphs. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in Neural Information Processing*

Systems. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/ef50c335cca9f340bde656363ebd02fd-Paper.pdf

Hogan, A., Blomqvist, E., Cochez, M., D'amato, C., Melo, G. D., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., Ngomo, A.-C. N., Polleres, A., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., & Zimmermann, A. (2021). Knowledge Graphs. *ACM Comput. Surv.*, *54*(4). https://doi.org/10.1145/3447772

Kingma, D. P., & Ba, J. (2017). Adam: A Method for Stochastic Optimization.

Kipf, T. N., & Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, *2014*(239), 2.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Ristoski, P., de Vries, G., & Paulheim, H. (2016). A Collection of Benchmark Datasets for Systematic Evaluations of Machine Learning on the Semantic Web. *International Semantic Web Conference*, 186–194.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, *20*(1), 61–80. https://doi.org/10.1109/TNN.2008.2005605

Schlichtkrull, M., Kipf, T. N., Bloem, P., van den Berg, R., Titov, I., & Welling, M. (2017). Modeling Relational Data with Graph Convolutional Networks.

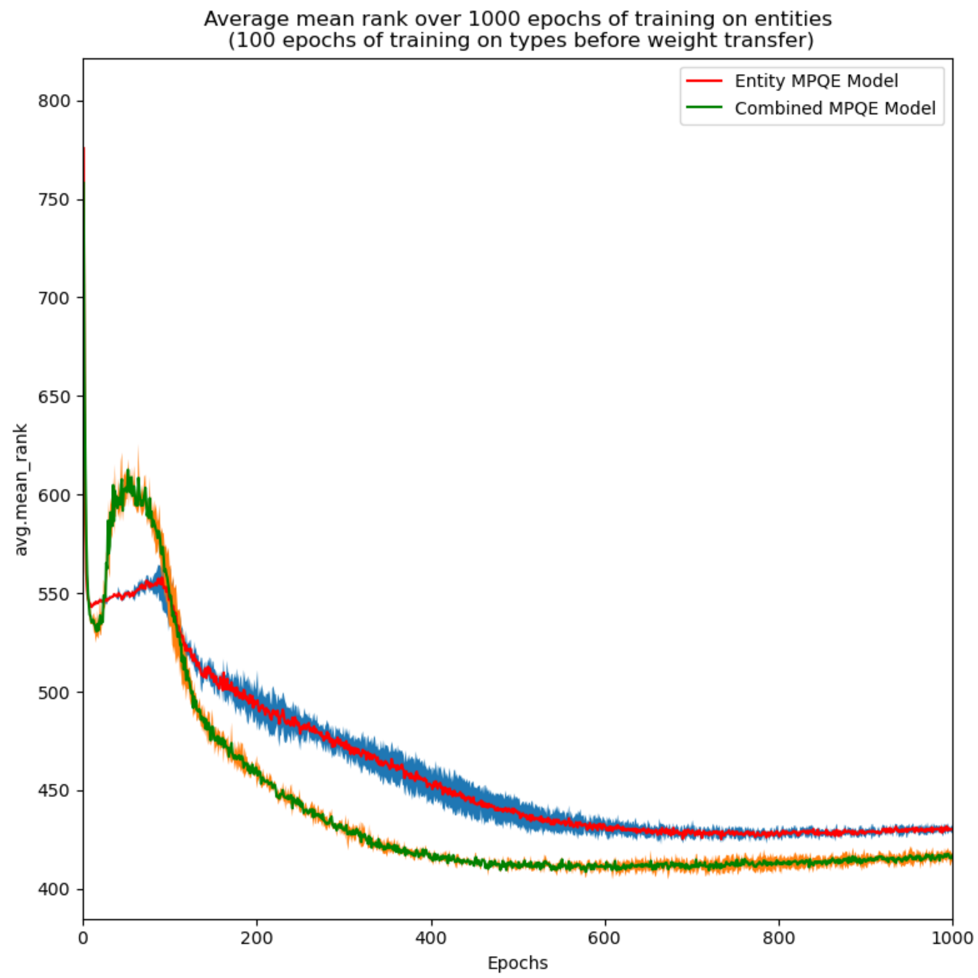
Snell, J., Swersky, K., & Zemel, R. S. (2017). Prototypical Networks for Few-shot Learning.

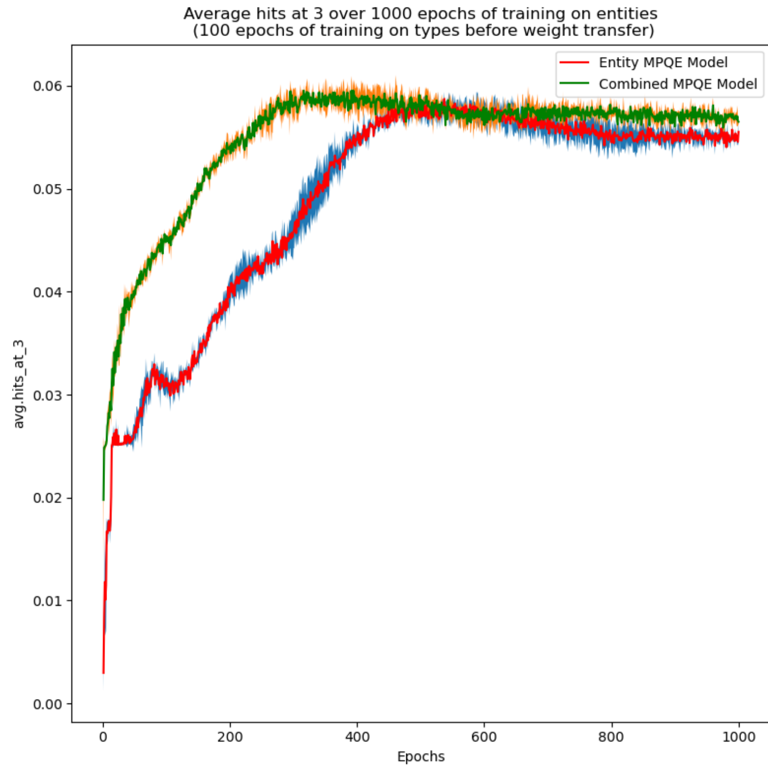
Ulyanov, D. (2016). Multicore-TSNE.

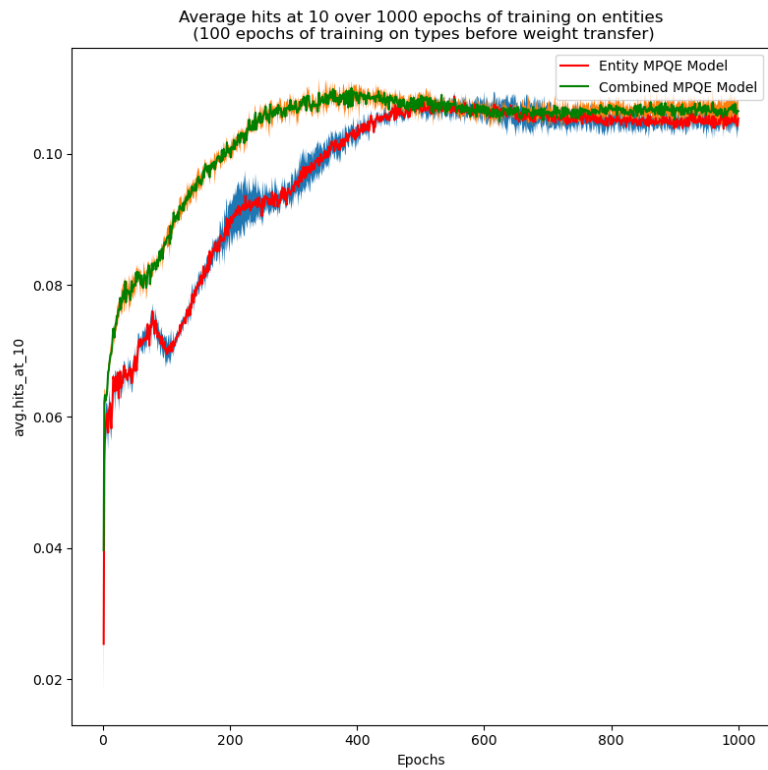
van der Maaten, L. (2014). Accelerating t-SNE using Tree-Based Algorithms. *Journal of Machine Learning Research*, *15*(93), 3221–3245. http://jmlr.org/papers/v15/vandermaaten14a.html

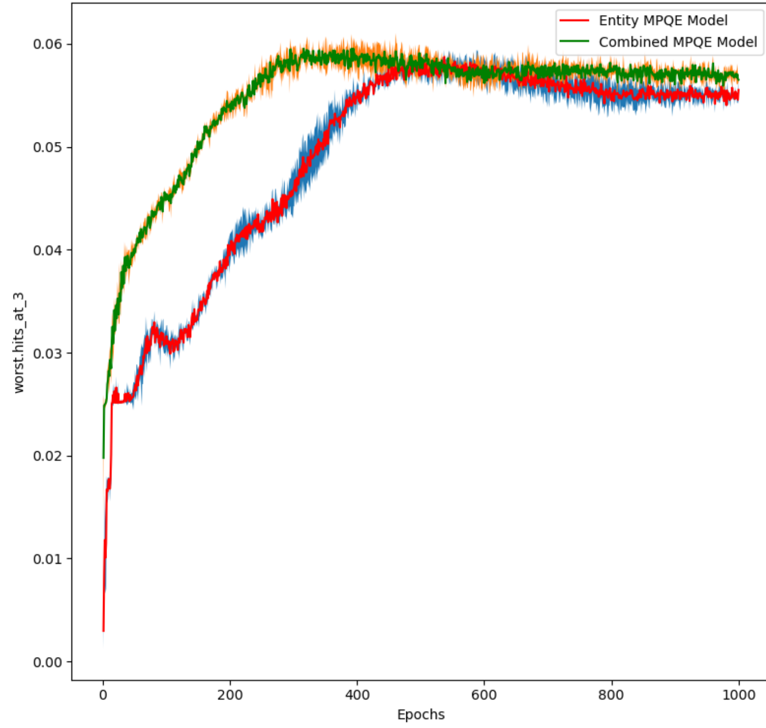
Appendix

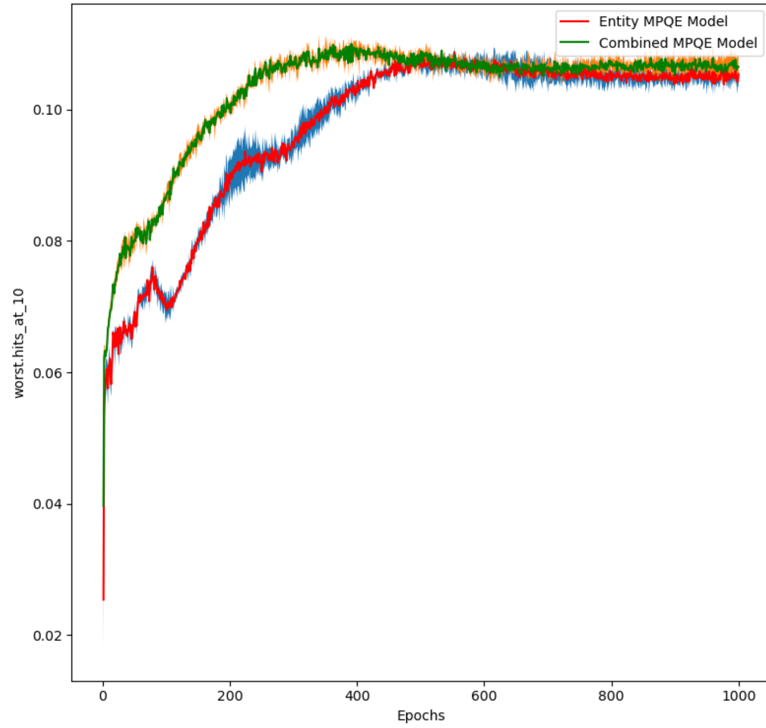
AIFB results figures



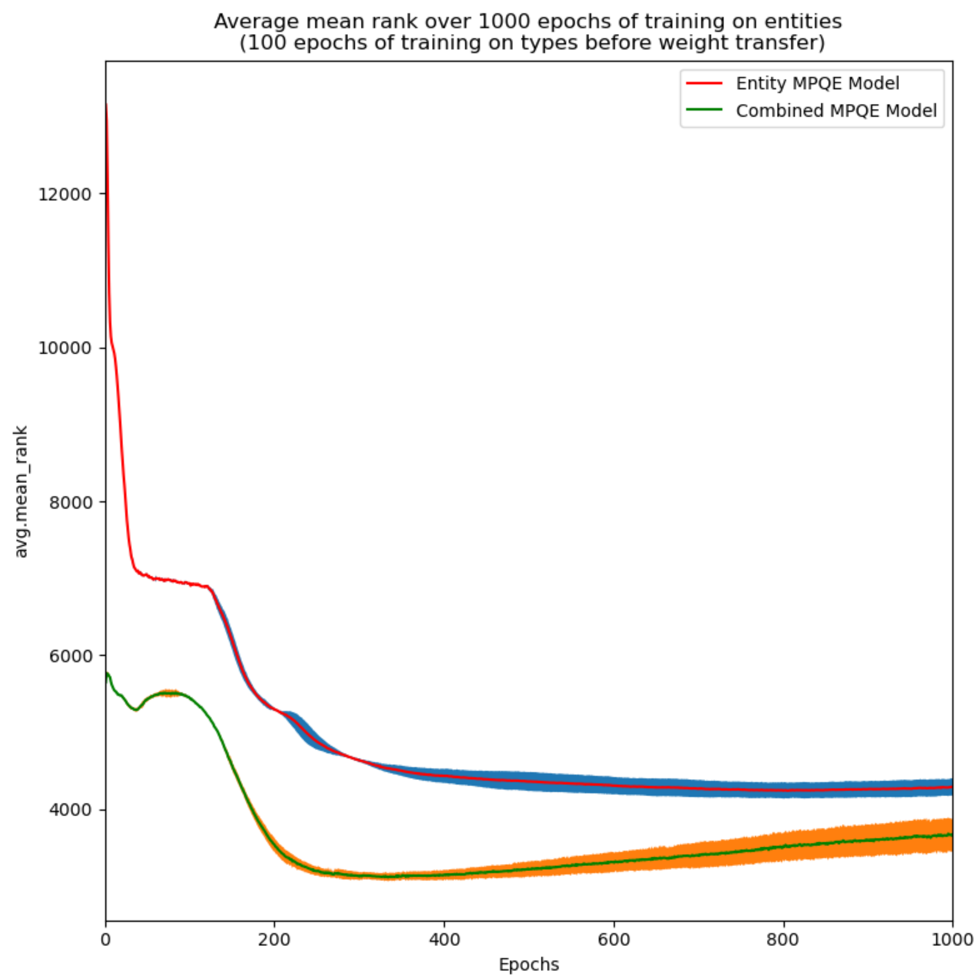

Average hits at 3 over 1000 epochs of training on entities
(100 epochs of training on types before weight transfer)
Entity MPQE Model
Combined MPQE Model
avg.hits_at_3
Epochs
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0
200
400
600
800
1000

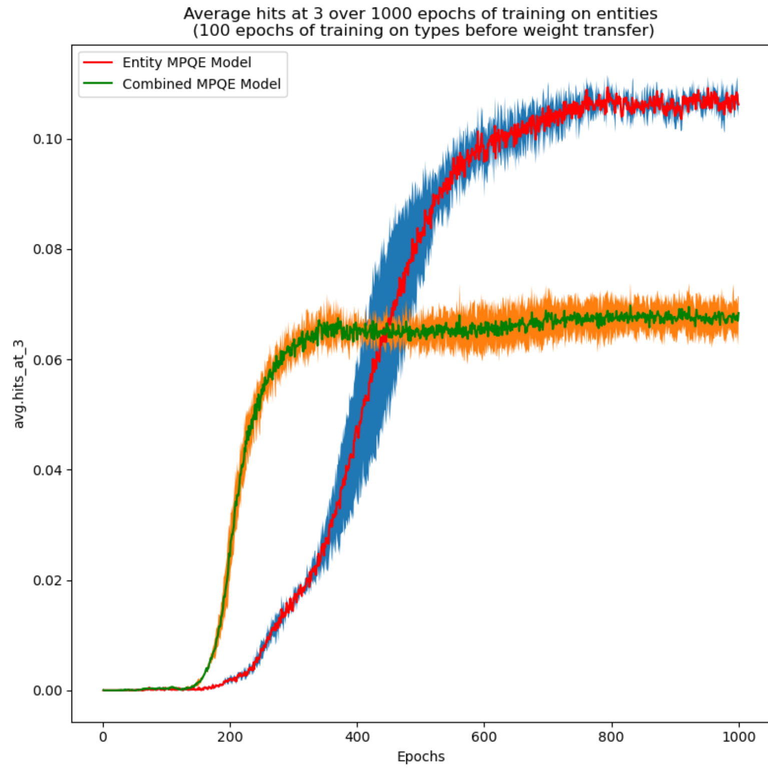

Average hits at 10 over 1000 epochs of training on entities
(100 epochs of training on types before weight transfer)
Entity MPQE Model
Combined MPQE Model
avg.hits_at_10
Epochs
0.02
0.04
0.06
0.08
0.10
0
200
400
600
800
1000

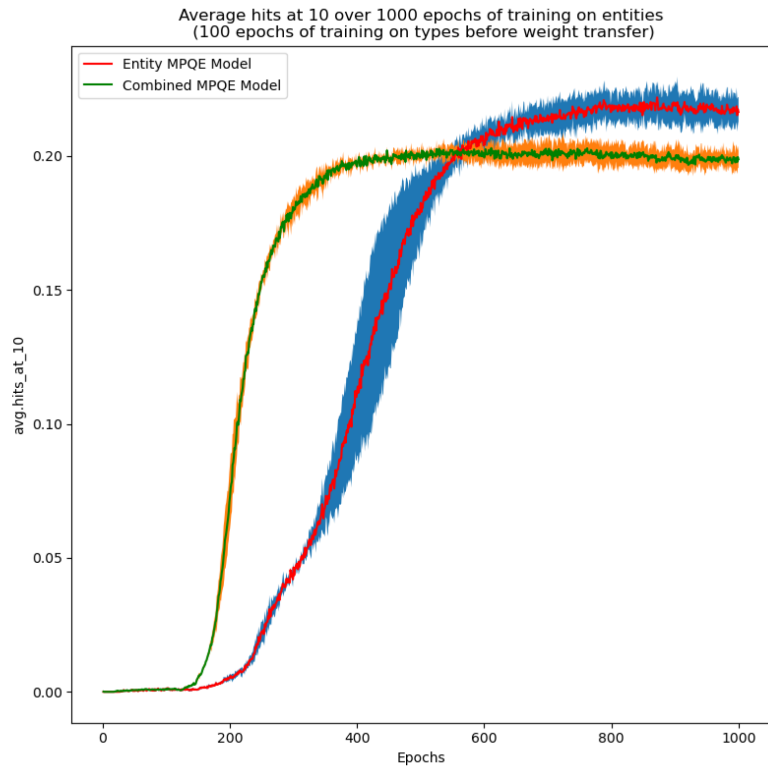

Worst hits at 3 over 1000 epochs of training on entities
(100 epochs of training on types before weight transfer)
Entity MPQE Model
Combined MPQE Model
worst.hits_at_3
Epochs
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0
200
400
600
800
1000

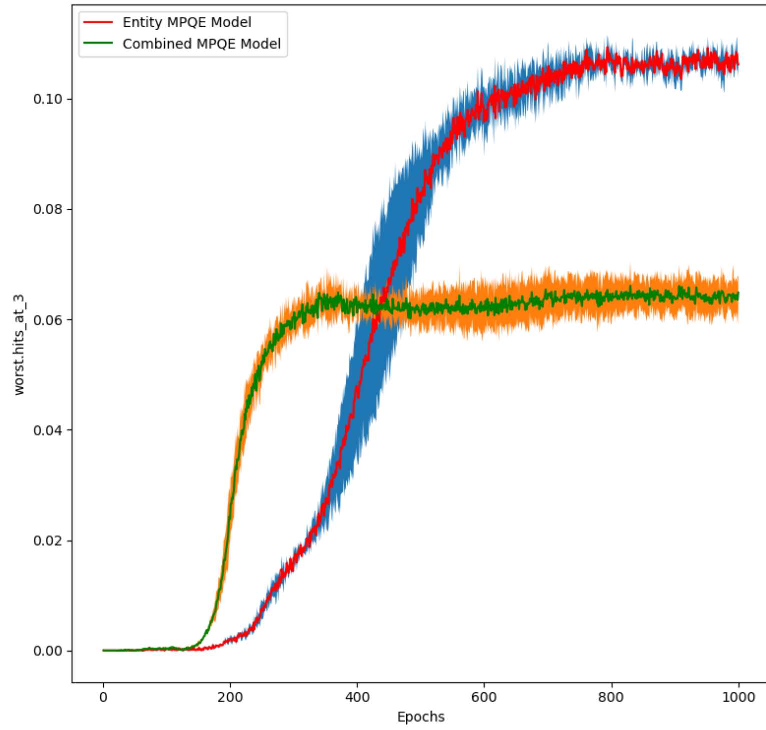

Worst hits at 10 over 1000 epochs of training on entities
(100 epochs of training on types before weight transfer)
Entity MPQE Model
Combined MPQE Model
worst.hits_at_10
Epochs
0.02
0.04
0.06
0.08
0.10
0
200
400
600
800
1000

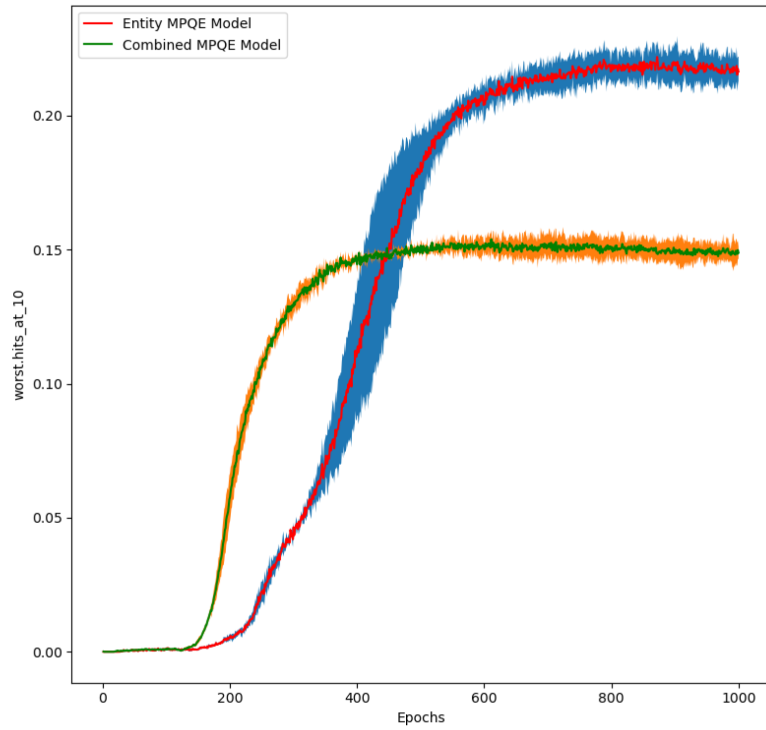
MUTAG results figures



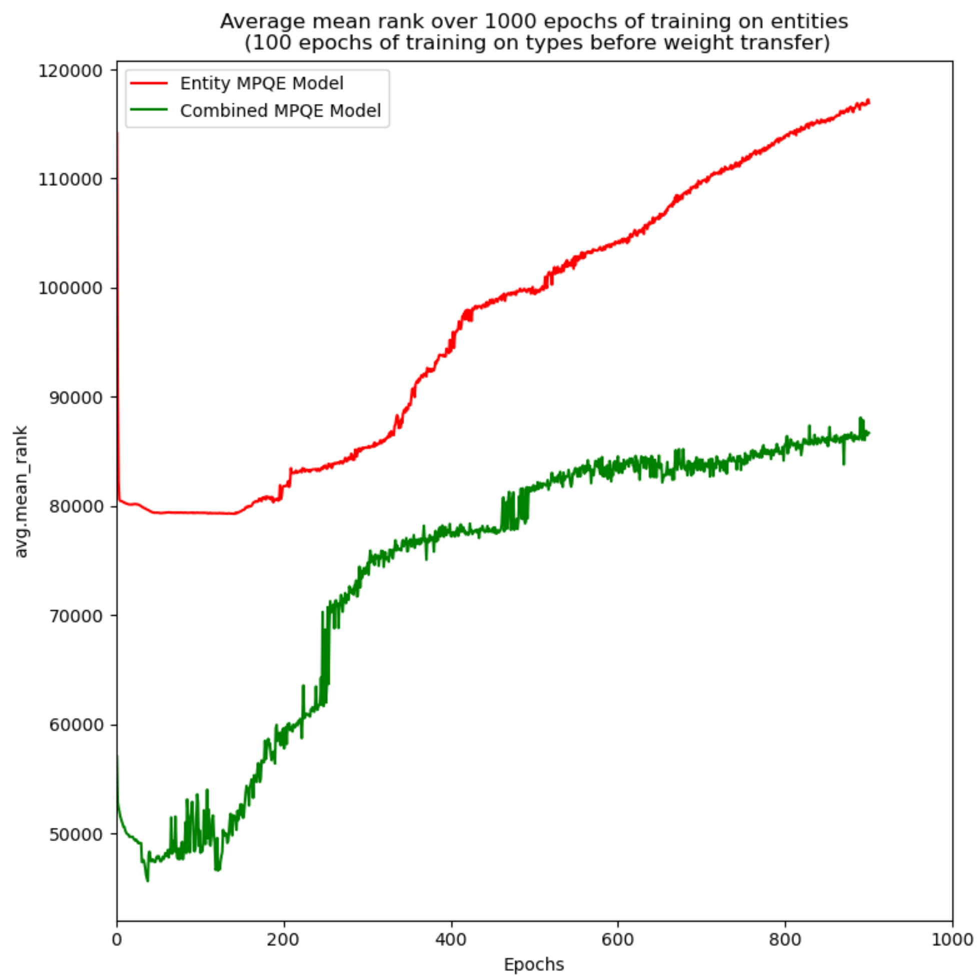

Average hits at 3 over 1000 epochs of training on entities
(100 epochs of training on types before weight transfer)
Entity MPQE Model
Combined MPQE Model
avg.hits_at_3
0.00
0.02
0.04
0.06
0.08
0.10
0
200
400
600
800
1000
Epochs

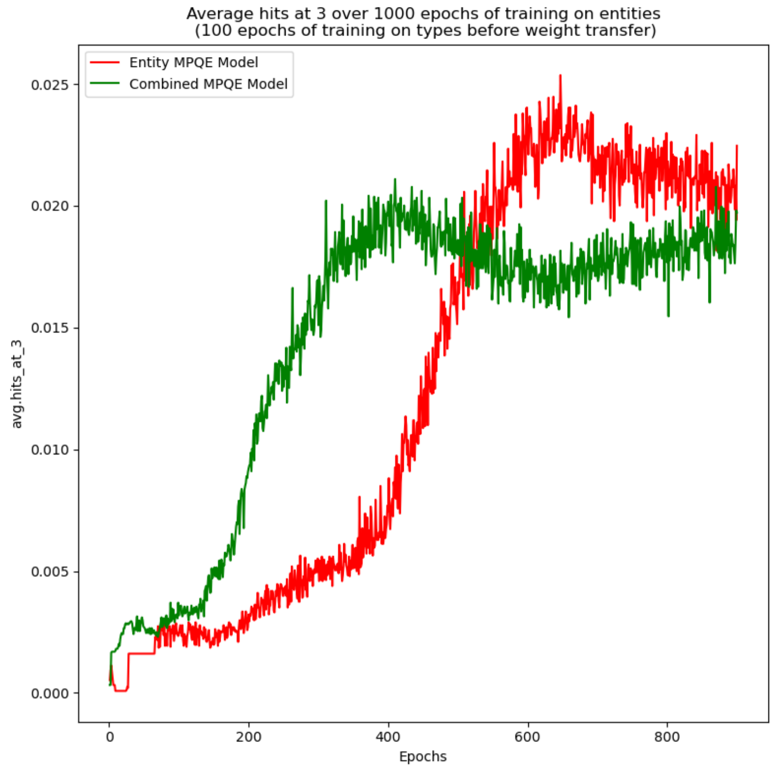

Average hits at 10 over 1000 epochs of training on entities
(100 epochs of training on types before weight transfer)
Entity MPQE Model
Combined MPQE Model
avg.hits_at_10
0.00
0.05
0.10
0.15
0.20
0
200
400
600
800
1000
Epochs

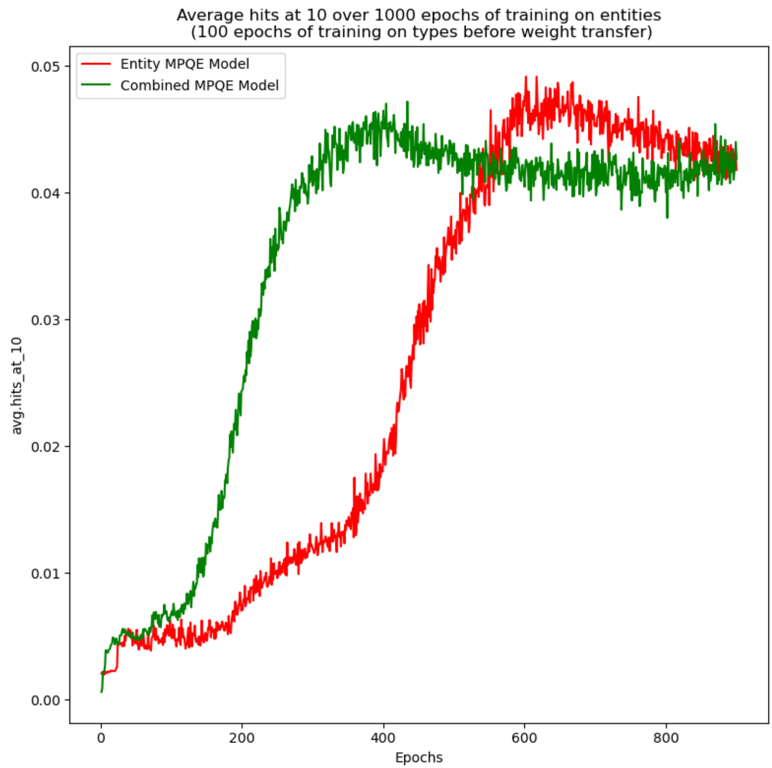

Worst hits at 3 over 1000 epochs of training on entities
(100 epochs of training on types before weight transfer)
Entity MPQE Model
Combined MPQE Model
worst.hits_at_3
0.00
0.02
0.04
0.06
0.08
0.10
0
200
400
600
800
1000
Epochs

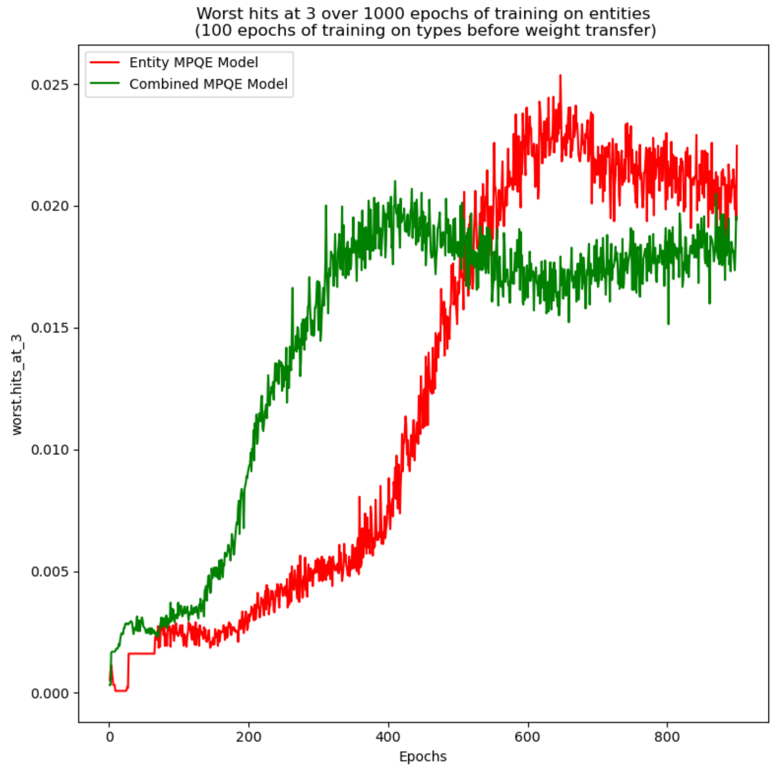

Worst hits at 10 over 1000 epochs of training on entities
(100 epochs of training on types before weight transfer)
Entity MPQE Model
Combined MPQE Model
worst.hits_at_10
0.00
0.05
0.10
0.15
0.20
0
200
400
600
800
1000
Epochs

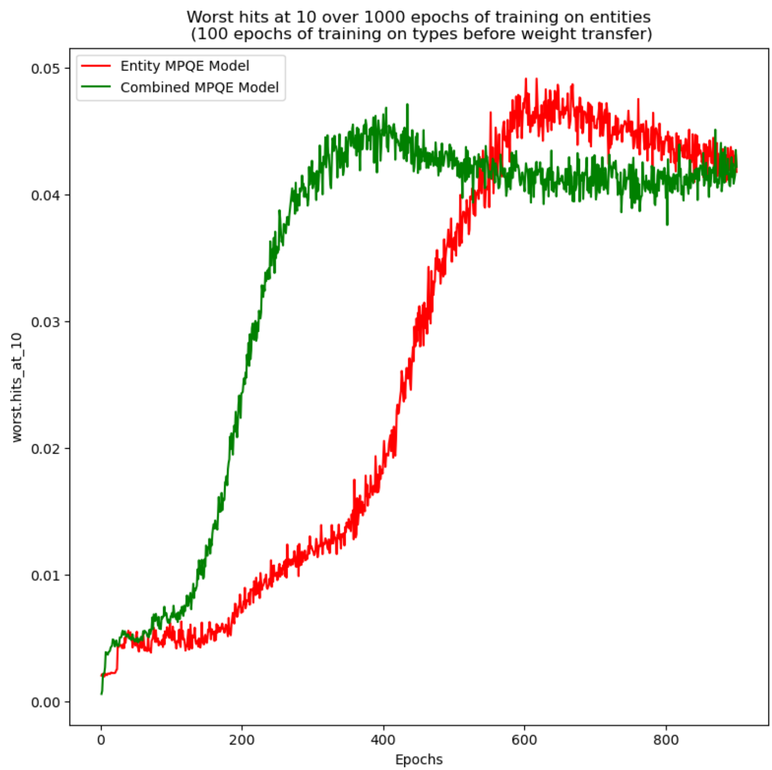
AM results figures




Average hits at 3 over 1000 epochs of training on entities
(100 epochs of training on types before weight transfer)
Entity MPQE Model
Combined MPQE Model
avg.hits_at_3
Epochs
0.000
0.005
0.010
0.015
0.020
0.025
0
200
400
600
800


Average hits at 10 over 1000 epochs of training on entities
(100 epochs of training on types before weight transfer)
Entity MPQE Model
Combined MPQE Model
avg.hits_at_10
Epochs
0.00
0.01
0.02
0.03
0.04
0.05
0
200
400
600
800


Worst hits at 3 over 1000 epochs of training on entities
(100 epochs of training on types before weight transfer)
Entity MPQE Model
Combined MPQE Model
worst.hits_at_3
Epochs
0.000
0.005
0.010
0.015
0.020
0.025
0
200
400
600
800


Worst hits at 10 over 1000 epochs of training on entities
(100 epochs of training on types before weight transfer)
Entity MPQE Model
Combined MPQE Model
worst.hits_at_10
Epochs
0.00
0.01
0.02
0.03
0.04
0.05
0
200
400
600
800