Advanced Topics in ML Group Project
Group Project Report

Team 6

Group members: Mateusz Parafinski, Sharan Gopal, Yufeng Yang, Peter Mernyei
Link to our implementation: https://github.com/sharan-dce/tucker

1 Introduction

The aim of this project was to reproduce the results of TuckER [1], a powerful linear knowledge graph embedding model. We chose this paper because we found TuckER to be a very elegant generalisation of many earlier linear models and wanted to verify if the theory matches practice. We were also impressed by how it achieved state-of-the-art results despite to simple, but effective ideas – TuckER is able to not only match but even beat more complicated non-linear models [1].

1.1 Knowledge Graph Completion and the TuckER model

Models such as TuckER are designed to solve the task of knowledge graph completion: given a set $\mathcal{E}$ of entities, a set $\mathcal{R}$ of relations and a set of known facts $\mathcal{F} \subset \mathcal{E} \times \mathcal{R} \times \mathcal{E}$, we wish to infer other facts that are likely to be true but not present in the dataset. They fall into the family of shallow embedding models, where weights are learned for the different entities and relations that allow scoring the probability of any given fact via simple arithmetic operations such as tensor multiplications (as opposed to neural network models). In the case of TuckER, the parameters consist of the following:

- An entity embedding matrix $\mathbf{E} \in \mathbb{R}^{n_e \times d_e}$, where n_e is the number of entities in the input and d_e is the dimensionality of entity embeddings. Each row is the embedding for the corresponding entity.
- A similar relation embedding matrix $\mathbf{R} \in \mathbb{R}^{n_r \times d_r}$, with each row of size d_r representing one of the n_r relations in the input.
- A core tensor $\mathcal{W} \in \mathbb{R}^{d_e \times d_r \times d_e}$, which allows for the reduction of two entity embeddings with one relation vector to produce a scalar that scores the probability of the corresponding fact.

This allows us to define the scoring function for each fact (s, r, o) as tensor multiplication:

$$\phi(s, r, o) = \mathcal{W} \times_1 \mathbf{s} \times_2 \mathbf{r} \times_3 \mathbf{o} = \sum_{ijk} \mathcal{W}_{ijk} s_i r_j o_k,$$

where the vectors are the rows corresponding to s, o in $\mathbf{E}$ and r in $\mathbf{R}$.

The output of the scoring function is an unrestricted real number, so we convert it to a probability by applying the sigmoid function. The model is trained with the standard negative likelihood loss to assign a high probability to known facts and a low probability to sampled negative examples. Although this discourages it from predicting any fact not in the training sample, with a reasonably restricted dimensionality the best possible loss is likely to be achieved by assigning high probability to unknown facts that fit well into the patterns of the training data.

1.2 Replication Goals

We aimed to reproduce the central result of the paper: achieving the reported performance on four standard knowledge graph completion benchmarks using the TuckER model using our own implementation in PyTorch, consisting not only of the model but also evaluation and data loading functionality. This is

described in Section 3.

We also decided on two extension tasks: we ran benchmarks on the RESCAL and DistMult models by restricting the parameters of our TuckER implementations, since these can be viewed as special cases of TuckER (Section 4). Secondly, we analysed the model's capability of capturing certain inference relations, both theoretically and empirically by creating synthetic toy datasets (Section 5).

2 Implementation

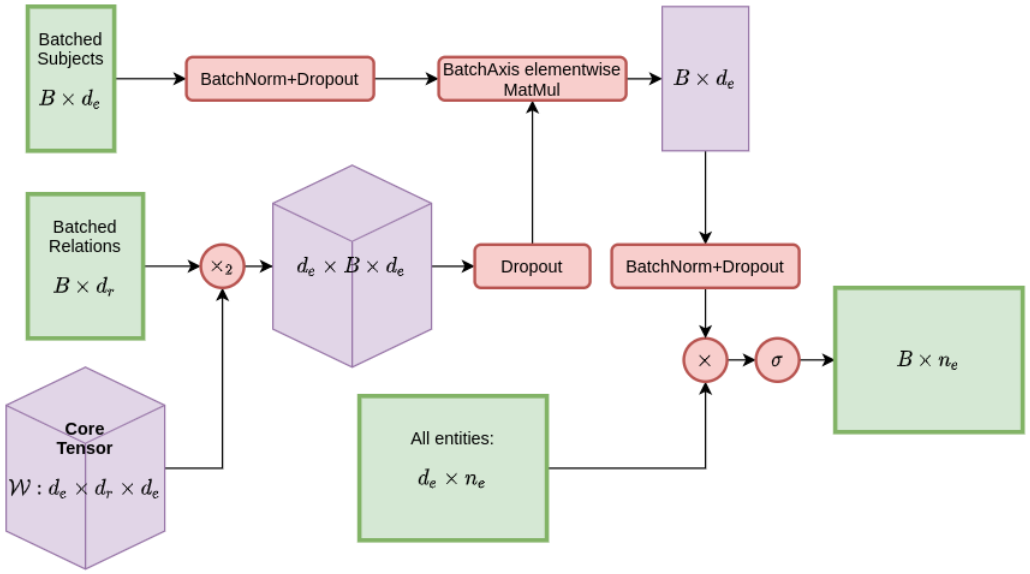

Figure 1: Batched TuckER forward pass.

This section hopes to cover some aspects of the model that were not explicitly clear from the paper, and gives a brief overview of how we went about implementing TuckER. The code is available at `https://github.com/sharan-dce/tucker`.

We implemented the model as a `torch.nn` module. During training, it takes in batched subject and relationship entities and scores them against all entities as objects in the dataset (in the literature this is referred to as 1-N *scoring* [1]). As mentioned in the paper, dropouts and batch normalization are integral to training the model and so we made sure to include these in our implementation. The first dropout and batch norm operations are applied to the subject embeddings (the subject and relation embeddings are part of the input to the model). We then carry out a tensor matrix multiplication of the core and the relationship embeddings, the result of which is also passed through a dropout layer. The processed core and processed entities then go through a tensor-matrix multiplication, which then is passed through final dropout and batch norm layers before being multiplied with all the entities (as objects) in the dataset, to yield logits. This sequence of operations is visualised in Figure 1.

TuckER's core tensor (or relationship embeddings for that matter) can be appropriately initialized and frozen to yield DistMult or RESCAL, which we implement by subclassing TuckER. For the details see Section 4.3.

Although not mentioned in the paper, Xavier Initialization [2] was used by the authors, something which we originally did not use in our code. Adding it later on yielded much faster convergence to the values stated in the original paper.

We have decided to implement our own pre- and post-processing techniques to gain a deeper understanding of the operation of the whole system and to be able to look at the original implementation with a critical eye. To this end, we have implemented both a data loader which is responsible for reading in the

datasets, data augmentation and processing the data for the 1-N scoring. We have also implemented the training and evaluation routines. The training code is rather standard and does not differ much from the original. The evaluation code works slightly differently as we try to work as much as possible with tensors instead of resorting to standard Python loops as is done in the original code. Despite not having done any benchmarking, we believe that our evaluation routine is significantly faster than the original one due to the use of tensor operations available in PyTorch which leverage parallel processing.

We use the following standard evaluation metrics

1. Mean Reciprocal Rank (MRR) - A true fact (s, r, o) sampled from the test set is scored and ranked against all candidate facts (s, r, o'), and its reciprocal is calculated. This is averaged out for all (s, r, o) triples in the training set.
2. Hits@k: Fraction of triples that come out to have at least rank k among negative facts (with the same subject & relation) and themselves. Most commonly used values of k are 1, 3 and 10.

3 Experiments

3.1 Datasets and approach

The main goal is to reproduce the results presented in Tables 3 and 4 of [1]. So, similarly to the original paper, we evaluate our model on four standard link prediction datasets:

1. **FB15k** [3] is a subset of Freebase, a large database of real world facts.
2. **FB15k-237** [4] was created from FB15k by removing the inverse of many relations that are present in the training set from validation and test sets, making it more difficult for simple models to do well.
3. **WN18** [3] is a subset of WordNet, a hierarchical database containing lexical relations between words.
4. **WN18RR** [5] is a subset of WN18, created by removing the inverse relations from validation and test sets.

We use the same parameters as in the original paper – this includes hyperparameters such as the learning rate, decay or dropout values, but also the dimensions of the entity and relation embeddings.

3.2 Results

Link prediction results on all datasets are shown in Tables 1, 2, 3 and 4. Overall, our TuckER implementation reproduces similar results to the original paper with the same hyperparameters. The biggest difference can be seen in the case of FB15k and we believe that is due to its size – FB15k has many more relations, and consequently also facts (twice as many as FB15k-237, ten times as many as WN18), than the other datasets.

For the experiments, we ran the training for 500 epochs and that seems to be enough as for all four datasets the loss managed to converge in that time. Note that most experiments are conducted on Google Colab (`colab.research.google.com/`) by our group members, so the hardware settings are most likely different every time (especially type of the GPU). Despite that, the training took no more than 2 hours on all four datasets. As mentioned in Section 2, Xavier Initialisation yields much faster convergence on all datasets.

4 Extension: Implementation of Subsumed Models

As discussed in the paper, several well-known prior models can be considered special cases of TuckER. This meant we could also replicate these by specialising our existing TuckER implementation. We chose to focus on RESCAL and DistMult for this extension.

FB15k				
Model	**MRR**	**Hits@10**	**Hits@3**	**Hits@1**
TuckER	.795	.892	.833	.741
Our TuckER	.761	.885	.813	.687
DistMult	.654	.824	.733	.546
Our DistMult	.636	.813	.706	.532
Our RESCAL	.560	.753	.622	.453

Table 1: Link prediction results on **FB15k**

FB15k-237				
Model	**MRR**	**Hits@10**	**Hits@3**	**Hits@1**
TuckER	.358	.544	.394	.266
Our TuckER	.354	.538	.389	.262
DistMult	.241	.419	.263	.155
Our DistMult	.329	.505	.361	.240
Our RESCAL	.342	.514	.373	.255

Table 2: Link prediction results on **FB15k-237**

WN18				
Model	**MRR**	**Hits@10**	**Hits@3**	**Hits@1**
TuckER	.953	.958	.955	.949
Our TuckER	.951	.957	.953	.948
DistMult	.822	.936	.914	.728
Our DistMult	.903	.955	.945	.861
Our RESCAL	.946	.955	.950	.940

Table 3: Link prediction results on **WN18**

WN18RR				
Model	**MRR**	**Hits@10**	**Hits@3**	**Hits@1**
TuckER	.470	.526	.482	.443
Our TuckER	.463	.516	.475	.435
DistMult	.430	.490	.440	.390
Our DistMult	.449	.516	.463	.414
Our RESCAL	.457	.510	.471	.429

Table 4: Link prediction results on **WN18RR**

4.1 Relation between RESCAL and TuckER

RESCAL [6] is a bilinear model. It encodes entities $s, o \in \mathcal{E}$ through d_e-dimensional vectors $\mathbf{s}, \mathbf{o} \in \mathbb{R}^{d_e}$. A relation $r \in \mathcal{R}$ is represented as a matrix $\mathbf{M}_r \in \mathbb{R}^{d_e} \times \mathbb{R}^{d_e}$. RESCAL scores a fact $r(s, o)$ according to the function $\mathbf{s}^T \mathbf{M}_r \mathbf{o}$, which captures all pairwise interactions between the components of $\mathbf{s}$ and $\mathbf{o}$.

Presenting it in the Tucker Decomposition format, RESCAL scoring function has the form

$$\mathcal{X} \approx \mathcal{Z} \times_1 \mathbf{E} \times_2 \mathbf{I}_{n_r} \times_3 \mathbf{E},$$

where $\mathcal{Z} \in \mathbb{R}^{d_e \times n_r \times d_e}$ is the core tensor, $\mathbf{E} \in \mathbb{R}^{n_e \times d_e}$ is the entity embedding matrix and $\mathbf{I}_{n_r} \in \mathbb{R}^{n_r \times n_r}$ is an identity matrix, so that during multiplication each relation selects a unique $d_e \times d_e$ slice of the core

tensor, which acts as its relation embedding matrix. n_e and n_r are the number of entities and relations respectively.

Therefore, we can define the scoring function ϕ for each triple $(s, r, o) \in \mathcal{F}$ as

$$\phi(s, r, o) = \mathcal{W} \times_1 \mathbf{s} \times_2 \mathbf{w}_r \times_3 \mathbf{o}.$$

$\mathbf{s}, \mathbf{o} \in \mathbb{R}^{d_e}$ are the rows of entity embedding matrix $\mathbf{E}$ representing the subject and object entity embedding vectors. $\mathcal{W} \in \mathbb{R}^{d_e \times n_r \times d_e}$ is the core tensor and $\mathbf{w}_r$ is a row of the identity matrix $\mathbf{I}_{n_r}$, which is a one-hot vector representing the index of relations.

4.2 Relation between DistMult and TuckER

DistMult [7] is also a bilinear model which restricts RESCAL's relation representation to a diagonal matrix instead of a full rank matrix. The entity encoding is the same as for RESCAL. The relation representation can be written as $\mathbf{D}_r \in \mathbb{R}^{d_e} \times \mathbb{R}^{d_e}$ which only has non-zero values on the diagonal. DistMult scores a fact $r(s, o)$ in the same way as RESCAL: $\mathbf{s}^T \mathbf{D}_r \mathbf{o}$. These diagonal matrices can be emulated in TuckER by multiplying relation embeddings with a fixed core tensor.

In the Tucker Decomposition format, DistMult scoring function has the following form

$$\mathcal{X} \approx \mathcal{Z} \times_1 \mathbf{E} \times_2 \mathbf{R} \times_3 \mathbf{E}.$$

The core tensor $\mathcal{Z} \in \mathbb{R}^{d_e \times d_e \times d_e}$ only has non-zero values 1 on the superdiagonal, all other elements are 0. As before, $\mathbf{E} \in \mathbb{R}^{n_e \times d_e}$ is the entity embedding matrix and $\mathbf{R} \in \mathbb{R}^{n_r \times d_e}$ is the relation embedding matrix.

For each triple $(s, r, o) \in \mathcal{F}$, we can define the scoring function

$$\phi(s, r, o) = \mathcal{W}_{\text{DM}} \times_1 \mathbf{s} \times_2 \mathbf{w}_r \times_3 \mathbf{o}.$$

$\mathcal{W}_{\text{DM}} \in \mathbb{R}^{d_e \times d_e \times d_e}$ is the core tensor which only has value 1 on the superdiagonal. $\mathbf{w}_r \in \mathbb{R}^{d_e}$ is the row of the relation matrix $\mathbf{R}$.

Since $\mathcal{W}_{\text{DM}}$ is just a superdiagonal core tensor, we have

$$\phi(s, r, o) = \mathcal{W}_{\text{DM}} \times_1 \mathbf{s} \times_2 \mathbf{w}_r \times_3 \mathbf{o} = \mathcal{W}_{\text{DM}} \times_1 \mathbf{o} \times_2 \mathbf{w}_r \times_3 \mathbf{s} = \phi(o, r, s)$$

which means that DistMult cannot differentiate between the head and the tail entities (thus cannot learn to represent asymmetric relations).

4.3 Implementation of RESCAL and DistMult

In our implementation of TuckER, we can set the initial core tensor, entity embedding matrix and relation embedding matrix, allowing us to create models behaving like RESCAL and DistMult. In addition, as some elements may not be involved in backward computation, one can use `gradient_mask` parameter to guarantee that the masked elements will not be updated.

```
class TuckER(torch.nn.Module):
    def __init__( self,
        num_entities: int,
        num_relations: int,
        initial_tensor,
        initial_entity_embeddings=None,
        initial_relation_embeddings=None
    ):...
```

Based on the analysis in the above subsections, RESCAL and DistMult models can be constructed simply with the help of our TuckER class. In the case of RESCAL, the core tensor is an element of $\mathbb{R}^{d_e \times n_r \times d_e}$, and the relation embedding matrix of RESCAL is an identity matrix $\mathbf{I}_{n_r}$ which does not require gradient computation:

```
class RESCAL(TuckER):
    def __init__(self,num_entities: int, num_relations: int, embedding_dim: int):
        super(RESCAL, self).__init__(
            num_entities=num_entities,
```

```
            num_relations=num_relations,
            initial_tensor=np.random.normal(
                            size=[embedding_dim, num_relations, embedding_dim]),
            initial_relation_embeddings=np.identity(num_relations, dtype=np.float32)
        )
        self.relation_embeddings.weight.requires_grad = False
```

In DistMult, the core tensor $\mathcal{Z} \in \mathbb{R}^{d_e \times d_e \times d_e}$ has non-zero value 1 only on the superdiagonal, and it should not be updated throughout the training process. As the behaviour of the entity and relation embedding matrices is the same as in TuckER, we don't have to take care of creating them manually in the constructor:

```
class DistMult(TuckER):
    def __init__(self, num_entities: int, num_relations: int, embedding_dim: int):
        self.embedding_dim = embedding_dim

        # the core tensor only has non-zero value 1 on the superdiagnoal
        ini_tensor = np.zeros([embedding_dim, embedding_dim, embedding_dim])
        ini_tensor[np.diag_indices(embedding_dim, ndim=3)] = 1

        super(DistMult, self).__init__(
            num_entities=num_entities,
            num_relations=num_relations,
            initial_tensor=ini_tensor
        )
        self.core_tensor.requires_grad = False
```

4.4 Results

We ran both models on all the benchmarks that were used in the TuckER paper (see Section 3.1). The Tables 1, 2, 3 and 4 present our results. As the results for DistMult are reported in [1], we are able to compare our implementation to the original one. For RESCAL, the results are not included in the TuckER paper, we can only compare RESCAL to DistMult and conclude that it seems to do better on all datasets except for FB15k. This makes sense considering DistMult's limitations described in Section 4.2. It is also worth noting that, within the expectations, our TuckER model outperforms DistMult and RESCAL across four datasets and all metrics – this is due to the fact that TuckER is a generalisation of the other two.

Considering that our RESCAL and DistMult models are largely based on the implementation of our TuckER model, all the tricks used for TuckER (such as dropout, batch norm, learning rate decay) are also utilized when training them. We suspect that this may be the reason for our DistMult's results being higher than the ones reported for the official implementation of DistMult.

5 Extension: Capturing inference patterns

Although the paper does not discuss this, it is common to analyse models according to what inference patterns a knowledge graph embedding model can capture, i.e. whether they can learn relation weights that ensure certain connections between specific relations for arbitrary entity embeddings [8]. We provide theoretical analysis and empirical investigation using synthetic datasets for the inference patterns of inversion, symmetry, composition and hierarchy in the case of TuckER.

5.1 Equivalence with RESCAL

When looking at specific relations, it is useful to look at TuckER's connection to RESCAL. Firstly, as described in the previous section, TuckER generalises RESCAL: any configuration of RESCAL's parameters can be represented as an appropriate configuration of TuckER. This means any inference pattern captured by RESCAL is also captured by TuckER.

However, if we allow for a large increase in the number of parameters, RESCAL can also represent any function that TuckER can learn. Let $\mathbf{s}, \mathbf{o}$ be arbitrary entity embeddings, $\mathbf{r}$ an arbitrary relation embedding, and let $\mathcal{W}$ be the core tensor. Then for $M_{ik}^{(r)} = \sum_j \mathcal{W}_{ijk} r_j$, we can write TuckER's scoring

function as

$$\phi(s, r, o) = \sum_{ijk} \mathcal{W}_{ijk} s_i r_j o_k = \sum_{ik} M^{(r)}_{ik} s_i o_k = \mathbf{s}^T M^{(r)} \mathbf{o}$$

So by reducing the core tensor with the relation vector first, TuckER's relation embeddings can be considered as defining a RESCAL-like matrix embedding via weights over the core tensor's slices. This means any inference pattern captured by TuckER is also captured by RESCAL, so they capture exactly the same patterns, and it suffices to reason about RESCAL-like matrices for each relation when analysing TuckER.

5.2 Symmetry

5.2.1 Theory

Symmetry of a relation r is the property $\forall x, y.\ r(x, y) \implies r(y, x)$. In terms of matrices, this is ensured if we require $\forall \mathbf{x}, \mathbf{y}.\ \mathbf{x}^T M^{(r)} \mathbf{y} = \mathbf{y}^T M^{(r)} \mathbf{x}$. By considering standard basis vectors $\mathbf{e}_i, \mathbf{e}_j$ in place of $\mathbf{x}, \mathbf{y}$, we can see this is equivalent to $M^{(r)}_{ij} = M^{(r)}_{ji}$, i.e. $M^{(r)} = {M^{(r)}}^T$, the matrix being symmetric. Therefore this pattern can be captured by both RESCAL and TuckER.

5.2.2 Experiments

We created a simple synthetic dataset with $10,000$ entities and a single relation r, where $r(e_i, e_j)$ and $r(e_j, e_i)$ hold if and only if $i = 2k, j = 2k + 1$ for some k. From 30% of the pairs, one of the two facts was put in the test set at random; for the rest, both directions were in the training set.

As expected, the model was able to perform perfectly on the training set, with 100% top-1 accuracy and therefore 1.0 MRR. This was achieved with entity embedding dimension of 30, relation embedding dimension of 1 (more would've been pointless since we had just a single relation in the dataset), a LR of 0.01 with no decay, and no dropout or label smoothing. The model consistently converged in 100 epochs.

5.3 Inversion

5.3.1 Theory

Inversion concerns two distinct relations $r_1 \neq r_2$: $\forall x, y.\ r_1(x, y) \Leftrightarrow r_2(y, x)$. By the same reasoning as in the case of symmetry, we can see that it suffices to require $M^{(r_1)} = {M^{(r_2)}}^T$, i.e. the matrices should be transposes of each other.

5.3.2 Experiments

The empirical evaluation was also very similar: we paired up $10,000$ entities with two relations so that both $r_1(e_i, e_j)$ and $r_2(e_j, e_i)$ hold if and only if $i = 2k, j = 2k + 1$, and selected one fact from 30% of all pairs to form the test set.

The model was able to perform perfectly on the test set with the same hyperparameters we used in the case of symmetry, except for the relation embedding dimension being 2; the entity embedding dimension was 30 and a LR of 0.01 was used with no decay, without dropout or label smoothing. Again, the model converged in around 100 epochs.

5.4 Composition

5.4.1 Theory

Composition involves three distinct relations r_1, r_2 and r_3 such that $\forall x, y, z.\ r_1(x, y) \wedge r_2(y, z) \implies r_3(x, z)$. If this implication had to hold when the entity embedding vector $\mathbf{y}$ has arbitrarily large magnitude, that would also push up the scores $\phi(x, r_1, y)$ and $\phi(y, r_2, z)$ arbitrarily high, so all x and z would have to be related by r_3. So in this full generality this pattern cannot be captured by models such as TuckER and RESCAL.

However, there are cases where it's not so problematic: suppose it is possible to find a configuration where all true facts (x, r, y) have the vectors $\mathbf{x}$ and $M^{(r)}\mathbf{y}$ point in exactly the same direction (i.e. they have

cosine similarity distance $d(\mathbf{x}, M^{(r)}\mathbf{y}) = d({M^{(r)}}^{-1}\mathbf{x}, \mathbf{y}) = 0)$ while this is not the case for any negative samples. Then if both $r_1(x, y)$ and $r_2(y, z)$ hold, we have $d({M^{(r_1)}}^{-1}\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, M^{(r_2)}\mathbf{z}) = 0$, so by the triangle inequality we also have $d({M^{(r_1)}}^{-1}\mathbf{x}, M^{(r_2)}\mathbf{y}) = d(\mathbf{x}, M^{(r_1)}M^{(r_2)}\mathbf{y}) = 0$. So if r_3 has matrix $M^{(r_3)} = M^{(r_2)}M^{(r_1)}$ then $r_3(x, z)$ will also hold.

Note that although it is not likely that all facts can be represented so perfectly in general in real-world datasets, it might be close for a large fraction of them, so this makes matrix composition a reasonable candidate for partially capturing the composition pattern.

5.4.2 Experiments

We created a simple synthetic dataset for this pattern similarly to the earlier ones: we constructed triples of entities (e_i, e_{i+1}, e_{i+2}) with facts $r_1(e_i, e_{i+1}), r_2(e_{i+1}, e_{i+2})$ and $r_3(e_i, e_{i+2})$ and sampled one of these facts from 30% of the triples as our test set. Note that this is a very simple case where the extra condition above is satisfied.

We found that the model could converge to perfect performance as expected, but it was much more difficult to find hyperparameters where it achieved this consistently. We succeeded with an entity embedding dimension of 20, a relation embedding dimension of 15, and a LR starting at 0.05 with an exponential decay with parameter 0.995. Dropout and label smoothing were not used. This led to consistent perfect performance within 300 epochs.

It is worth noting that the relation embedding dimension had a large effect on performance despite the fact that 3 dimensions should theoretically suffice. One possible explanation is that with low dimensionality, there is too much weight sharing between the three relations so it is harder to optimise them to make one the composition of others. However, when we tried training RESCAL, it never quite achieved perfect performance. A better explanation might be that this allowed the model to use slices of the core tensor that happened to be better initialised for representing the composing matrices that were needed. This might also give insight into how TuckER outperforms RESCAL in general.

Another interesting finding: with other hyperparameters, we sometimes found the model converged to nearly zero loss while performing very badly on the test set. This seems to imply the training data does not fully constrain the model as intended, but it is not entirely clear how.

5.5 Hierarchy

5.5.1 Theory

The hierarchy pattern involves a relation r_1 being a subset of a relation r_2: $\forall x, y.\ r_2(x, y) \implies r_1(x, y)$. In general, this requires $\forall \mathbf{x}, \mathbf{y}.\ \mathbf{x}^T M^{(r_1)}\mathbf{y} \geq \mathbf{x}^T M^{(r_2)}\mathbf{y}$. Setting $\mathbf{x} = \mathbf{e}_i, \mathbf{y} = \mathbf{e}_j$ here gives $M_{ij}^{(r_1)} \geq M_{ij}^{(r_2)}$. However, setting $\mathbf{x} = \mathbf{e}_i, \mathbf{y} = -\mathbf{e}_j$ gives $M_{ij}^{(r_1)} \leq M_{ij}^{(r_2)}$. So these models cannot capture this pattern well except in the trivial case $r_1 = r_2$.

As in the case of compositions, we can consider weakened versions: for example, if all entity weights are positive, it suffices to have $M_{ij}^{(r_1)} \geq M_{ij}^{(r_2)}$. It seems unlikely that this could be a realistic solution in real-world datasets, but something similar might be sufficient for simple synthetic ones.

5.5.2 Experiments

When creating a dataset for this pattern, we wanted to ensure the trivial $r_1 = r_2$ is not a sufficient solution. We did this by creating two mutually exclusive relations r_1, r_2 that were disjoint subsets of r_0. Similarly to the case of compositions, we considered triples of entities e_i, e_{i+1}, e_{i+2} and added $r_0(e_i, e_{i+1}), r_0(e_i, e_{i+2}), r_1(e_i, e_{i+1})$ and $r_2(e_i, e_{i+2})$ as facts, selecting one of the last two (involving r_1, r_2) for the test set in 30% of all triples.

The results we found are interesting: using the same configuration as for inversions, the model performed near perfectly after 30 epochs, with both MRR and top-1 accuracy above 0.99. However, it quickly

declined with more training, down to 0.72 MRR and 0.76 top-1 accuracy at epoch 100. This suggests it is possible to create a perfect parametrisation for this dataset, but it is somehow not stable, and the loss is better optimised with some set of parameters that performs worse.

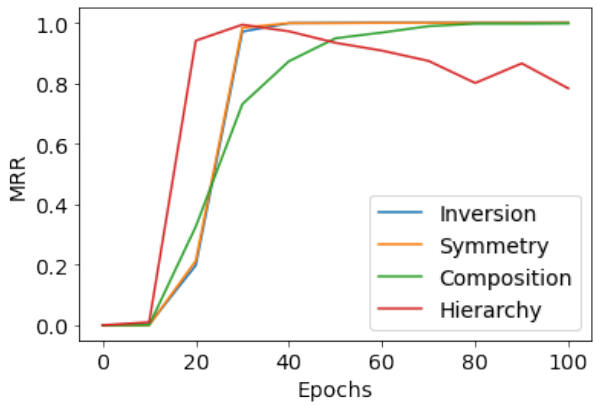

Figure 2: MRR convergence on the syntethic datasets for all inference patterns.

6 Conclusion

TuckER is a simple linear and fully expressive model. In this project we managed to reproduce the results from the original paper by almost matching the MRR and hits@k on all four benchmarks. We were also able to achieve a significant speedup during the evaluation of the model's performance by leveraging tensor operations. We discovered that dropout and augmentation (using reverse relations) are integral to achieving good performance, and Xavier initialization significantly improves the convergence rate allowing us to replicate the results in 500 epochs. Knowing that TuckER generalises other linear models, we decided to implement DistMult and RESCAL as subclasses of TuckER. The results of this experiment somewhat match our expectations – both DistMult and RESCAL perform worse than TuckER due to the limits imposed on the embeddings. Finally, to extend our analysis and understanding of TuckER, we trained TuckER on synthetic toy datasets, specifically generated to capture inversion, symmetry, composition and hierarchy. Our results match the theory – TuckER was able to learn inversion and symmetry perfectly in a small number of epochs and managed to perform well in special cases of composition and hierarchy. Nonetheless, there are a few questions that remain unanswered, including better than expected performance of our implementation of DistMult or the declining MRR when training TuckER on the synthetic hierarchy dataset.

References

[1] I. Balažević, C. Allen, and T. Hospedales, "Tucker: Tensor factorization for knowledge graph completion," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 5188–5197, 2019.

[2] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Y. W. Teh and M. Titterington, eds.), vol. 9 of *Proceedings of Machine Learning Research*, (Chia Laguna Resort, Sardinia, Italy), pp. 249–256, PMLR, 13–15 May 2010.

[3] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *Neural Information Processing Systems (NIPS)*, pp. 1–9, 2013.

[4] K. Toutanova, D. Chen, P. Pantel, H. Poon, P. Choudhury, and M. Gamon, "Representing text for joint embedding of text and knowledge bases," in *Proceedings of the 2015 conference on empirical methods in natural language processing*, pp. 1499–1509, 2015.

[5] T. Dettmers, P. Minervini, P. Stenetorp, and S. Riedel, "Convolutional 2d knowledge graph embeddings," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[6] M. Nickel, V. Tresp, and H.-P. Kriegel, "A three-way model for collective learning on multi-relational data," in *ICML*, 2011.

[7] B. Yang, W.-t. Yih, X. He, J. Gao, and L. Deng, "Embedding entities and relations for learning and inference in knowledge bases," *arXiv preprint arXiv:1412.6575*, 2014.

[8] İ. İ. Ceylan, "Lecture 1: Relational data & embedding models." Advanced Topics in Machine Learning Course, 2021.