Knowledge graph embeddings in function spaces using Neural Architecture Search

Vikrant Singh

University of Paderborn

Computer Science
University of Paderborn
DICE Group

Master Thesis

Knowledge graph embeddings in function spaces using Neural Architecture Search

Vikrant Singh

1. Reviewer	Prof. Dr. Axel-Cyrille Ngonga Ngomo Data Science Group University of Paderborn
2. Reviewer	Jun.-Prof. Dr. Sebastian Peitz Data Science for Engineering University of Paderborn
Supervisors	Prof. Dr. Axel-Cyrille Ngonga Ngomo

Abstract

Knowledge Graphs (KGs) stand as pivotal structures in the realm of data science, providing a sophisticated framework for encapsulating entities and their intricate interrelations through a network of interconnected nodes and edges. KGs underpin a multitude of cutting-edge applications, ranging from augmenting the capabilities of search engines to enabling complex question-answering systems, thereby playing a crucial role in advancing the frontier of artificial intelligence and knowledge discovery. Despite their effectiveness, most of the Knowledge Graphs are incomplete paving the way for KG embeddings. KG embeddings have been an active area of research in the previous years. However previous works have mainly focused on leveraging vector spaces for the embedding models. To this end, we propose Knowledge Graph Embedding in function spaces. This functional approach allows for an expanded parameter space, endowing the model with increased degrees of freedom and, consequently, the potential for capturing more nuanced and complex relationships within KGs.

The exploration of embeddings as functions in this research is driven by the hypothesis that a function-based representation, by its increased parameterization, can facilitate more effective learning and adaptation, thereby enhancing the performance of KG embedding models in tasks such as link prediction and KG completion. We propose experimenting with an array of spaces starting from polynomial function spaces followed by complex number function spaces and neural network function spaces on both standard and large-scale KG. To this end, we conduct experiments with several scoring functions such as vector triple product, compositional and trilinear product along with various loss functions namely binary cross entropy logits loss function, L2 norm loss function, and margin loss function. Through a meticulous examination of various function spaces, scoring functions, and loss functions, coupled with the optimization of neural network architectures, the aim is to rigorously evaluate the efficacy of the proposed approach.

We perform evaluation using established metrics such as Mean Reciprocal Rank (MRR) and Hits@K, and also report the metrics for the state-of-the-art KG embedding models, aiming to provide a comprehensive understanding of its performance and potential advantages. Our results suggest that both the complex number and neural network function spaces coupled with the compositional scoring function performed

well with smaller datasets but encountered challenges as the number of triples within a dataset increased. While these function spaces demonstrated good generalization capabilities when model parameter dimensions were scaled up, they did not exhibit the same level of performance with larger datasets.

Acknowledgement

I extend my deepest gratitude to my family for their unwavering support and sacrifices for my education. I would not have advanced in my academic career without their dedication. Their unshakable emotional support has provided me with a haven during trying times. Being a part of this family makes me feel incredibly lucky and proud. My sincere gratitude is extended to my father, mother and brother.

I would especially like to express my gratitude to my supervisor, Dr. Prof. Dr. Axel-Cyrille Ngonga Ngomo, whose support and direction have made this project exciting and fulfilling. I wholeheartedly thank Dr. Ngonga for his priceless technical advice and counsel for the development.

I earnestly thank Prof. Dr. Axel-Cyrille Ngonga Ngomo and Dr. Sebastien Peitz for their role in examining this thesis.

Lastly, I express my gratitude to the University of Paderborn for the use of its excellent facilities, which were instrumental in the computational aspects of this research.

Contents

1 Introduction

1.1 Problem Statement and Motivation

Knowledge Graphs (KGs) serve as structured representations of real-world facts and concepts, encapsulating entities and their interrelations through a network of interconnected nodes and edges [Lia+22] They underpin a multitude of applications, from enhancing search engines to facilitating sophisticated question-answering systems [Sin12; Don+15; Xu+16]. Despite their widespread utility, KGs are inherently plagued by incompleteness–an issue that significantly hampers their effectiveness in capturing the full spectrum of real-world knowledge [NTK11].

The advent of distributed representation learning, notably through the introduction of word embeddings by Mikolov et al. [Mik+13b; Mik+13a], has had a profound impact on natural language processing (NLP). This paradigm shift towards embedding words in continuous vector spaces laid the groundwork for similar advancements in KG embeddings [Dai+ar]. The seminal work of Bordes et al. [Bor+13] with the TransE model was among the first to transpose the concept of translation in-variance from word vectors to the realm of, establishing a foundation for representing entities and relations as vectors in a continuous space.

Subsequent models, such as DistMult [Yan+14] and ComplEx [Tro+16], have built upon the TransE framework, contributing significantly to the advancement of KG embeddings. DistMult models entities and relations as vectors in a real-valued space, represented as $\mathbf{e}_h, \mathbf{e}_r \in \mathbb{R}^d$, while ComplEx extends this representation into a complex-valued space, with $\mathbf{e}_h, \mathbf{e}_r \in \mathbb{C}^d$, allowing for the capture of more complex relational patterns.

In contrast to the representation of embedding as vectors, this thesis explores, whether the use of embeddings as functions can enhance the performance of the KG embedding model in tasks like link prediction and KG completion. Our approach conceptualizes embeddings as functions of entities or relations and a variable x from a domain X, represented as $f(\theta, x)$, where θ denotes the parameters specific to an entity or relation. By adopting this embedding-as-function approach, we aim to increase the model's parameter space. This approach introduces more degrees of freedom, potentially allowing for more nuanced and adaptable representations of entities and relations. Such embeddings could offer enhanced performance in critical KG tasks, including link prediction and KG completion.

Consider a scenario in a KG involving a head entity h, a relation r, and a tail entity t, traditional embeddings would represent these elements as vectors whereas, in an embedding-as-function framework, they could be represented by functions such as $f_h(\theta_h, x)$, $f_r(\theta_r, x)$, and $f_t(\theta_t, x)$, where $\theta_h, \theta_r, \theta_t$ are the parameters associated with the head, relation, and tail respectively, and x is a sampled point from domain X. This method introduces a higher degree of flexibility and adaptability, potentially enabling more nuanced representations of entities and relations.

Implementing embedding as a function, enables us with a scope of utilizing neural networks. However, identifying an effective neural network architecture to create embedding as a function is a challenging task but can be crucial to overcoming the problem of incompleteness and their application in the downstream tasks.

Major Research Question How can the concept of embeddings as functions be effectively implemented to augment the capabilities of KGs embedding models?

Minor Research Questions

- How can different function spaces be assessed and chosen to effectively construct embeddings as functions $f(\theta, x)$ for knowledge graph embedding models?

- What scoring functions ψ are most suitable for use with embeddings formulated as functions $f(\theta, x)$, and how do they impact the model's ability to discern relational patterns within knowledge graphs?

- Considering embedding as functions, what loss functions L are most effective in optimizing the model's performance, particularly in tasks such as link prediction and KG completion?

- What approaches can be devised to systematically determine the most effective neural network configurations for knowledge graph embedding models that utilize embeddings as functions?

1.2 Research Objectives

The goal of this thesis is to explore the concept of embeddings as functions within the realm of knowledge graph (KG) embedding models. This approach diverges from traditional static vector representations, aiming to introduce a higher degree of adaptability and precision in representing entities and relationships. By conceptualizing embeddings as functions $f(\theta, x)$, parameterized by θ and modulated by input x from a specified domain, we seek to capture the nuanced dynamics of KGs more effectively. This method aims to enhance the model's performance in critical tasks such as link prediction and KG completion, by providing a more flexible and context-aware representation of knowledge. Our investigation will delve into the optimal selection of function spaces, scoring and loss functions tailored for function-based embeddings, and the identification of neural network architectures that synergize with this advanced embedding paradigm.

Objective 1: *Establishing baseline function spaces*, to set a benchmark for evaluating the proposed embedding as functions. We initially plan to implement baseline function spaces, encompassing both polynomial and complex number spaces. These foundational spaces will serve as a comparative backdrop to our neural network function space, facilitating the assessment of the enhanced expressiveness and adaptability introduced by the embeddings as a function.

Objective 2: *Exploration of scoring functions* is a critical component of this research and involves implementing a wide array of scoring functions, such as compositional, vector triple product, and trilinear functions. The selection and optimization of these scoring functions are paramount, as they directly influence the model's predictive prowess in link prediction tasks, a key performance indicator for KG embeddings.

Objective 3: *Employment of diverse loss functions* to adeptly navigate the score differentiation between true triples ($\mathcal{T}^+$) and corrupted triples ($\mathcal{T}^-$. Hence we implement multiple loss functions notably, for instance, the $L2$ norm loss function, Binary Cross-Entropy Logits, and Margin Ranking Loss. The strategic use of these loss functions aims to refine the model's learning trajectory, optimizing the embeddings to accurately reflect the nuanced relationships within G.

Objective 4: We *evaluate* the efficacy of the embedding as a function model by leveraging established metrics such as Mean Reciprocal Rank (MRR) and Hits@K, with K set to 1, 3, and 10. These metrics will shed light on the model's accuracy and generalization capabilities, particularly in the context of link prediction tasks, providing a comprehensive understanding of its performance. For a fair comparison, we also evaluate the state-of-the-art KG embedding models such as TransE [Bor+13], ComplEx [Tro+16], DistMult [Yan+14], Keci [DN23], QMult and OMult [Dem+21].

Objective 5: *Optimization of Neural Network Architecture* involves dealing with the intrinsic challenge of identifying optimal neural network configurations for embedding as a function. Therefore we leverage Neural Architecture Search (NAS) to automate this process. Utilizing frameworks like NNI [Mic20], we aim to fulfil this objective and uncover efficient neural network architectures that bolster the performance of KG embeddings.

By fulfilling the aforementioned objectives, this research aspires to make a significant contribution to the realm of KG embedding.

2 Background

2.1 Knowledge Graph

Knowledge Graphs (KGs) marks a significant milestone in the way we structure and utilize data. Originating from the early semantic networks, KGs have evolved, with the term gaining prominence in the early 1970s [Sch73] and seeing a resurgence with the launch of Google's KG in 2012 [Sin12]. These graphs have become indispensable in various fields, providing a structured and interconnected way to represent real-world entities and their relationships.
KGs encapsulates entities, their attributes, and the intricate network of relationships between them. They are constructed as directed, labelled graphs that offer a versatile representation capable of encompassing a wide range of domains and applications. In recent years, the field of KGs (KGs) has witnessed significant growth with the advent of large-scale KGs such as SUMO [PNL02], YAGO [SKW07], Freebase [Bol+08], Wikidata [VK14], and DBpedia [Aue+07]. These open KGs have become instrumental in various natural language processing (NLP) applications, including NER, entity disambiguation, life sciences, and information extraction. Enterprise KGs like Amazon [Kri18], eBay [Pit+17], on the other hand, are proprietary and serve specific business or organizational goals, driving innovations in areas like search optimization, recommendation systems, and more.

***Definition 1 (Triple)*:** A triple in a KG, denoted by $\langle h, r, t \rangle$, forms the fundamental unit of information, where $h \in E$ is the head entity, $r \in R$ is the relation, and $t \in E$ is the tail entity.

Definition 2 (Knowledge Graph) [1]: A KG can be represented as a graph G of consisting a set of triples, where $G \subseteq E \times R \times E$. Here, E represents entities, symbolizing objects or concepts, and R denotes the relations between these entities. Entities are uniquely identified, often by a Uniform Resource Identifier (URI), ensuring clear and distinct representation.

[1] `https://www.ontotext.com/knowledgehub/fundamentals/what-is-a-knowledge-graph/`

The traditional structure of KGs, which uses RDF-style triples [2] to represent real-world objective information, faces challenges in computational efficiency and data sparsity. The computational complexity and poor scalability of graph algorithms used in KGs make them ill-suited for large-scale real-time computing. Additionally, the large-scale nature of these KGs leads to a significant data sparsity problem, rendering the calculation of semantic or inferential relationships between entities inaccurate. To address these challenges, the concept of KGs embedding (KGE) has gained traction.

2.1.1 Knowledge Graph Embedding

Knowledge Graph Embedding (KGE) aims to embed the entities and relations in a KG into continuous vector spaces while preserving the semantic relationships. This approach is essential for simplifying computations within the KG and overcome the challenge of incompleteness posed by them [Wan+17]. Over recent years, KGE has emerged as a focal point of research, with numerous scholars proposing a variety of innovative models. These models primarily differ in three critical aspects:

Representation of Entities and Relations This aspect focuses on how the models conceptualize and encode the entities and relationships within the KG. The representation form—whether vectorial, matrix-based, or even more complex structures—serves as the foundational component of KGE models, determining how effectively the KG's information is captured and utilized.

Definition 3 (Encoder) : Given an entity $e \in E'$ and relation $r \in R$, $\text{enc}_o(\cdot)$ produces a dense vector representation $e \in \mathbb{R}^d$ for e and $r \in \mathbb{R}^{d'}$:

$$\text{enc}_o : E' \to \mathbb{R}^d, \text{enc}_o : R \to \mathbb{R}^{d'}$$

Scoring Functions Scoring functions in KGE models evaluate the plausibility or likelihood of a given triplet (composed of two entities and a relation) being true within the context of the KG. These functions play a pivotal role in guiding the learning process, helping to distinguish between valid and invalid triples.

[2]http://www.w3.org/TR/rdf11-concepts

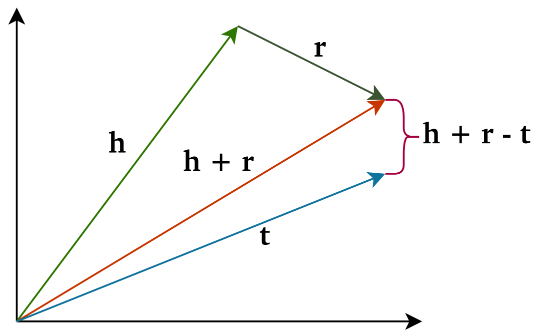


Fig. 2.1: TransE scoring function where h,r and t represent head, relation and tail respectively.

Definition 4 (Scoring function ψ) Given the representations $h \in \mathbb{R}^d$, $r \in \mathbb{R}^{d'}$, $t \in \mathbb{R}^d$ of the head h, relation r, and tail t of a triple (h, r, t), the scoring function $\psi : \mathbb{R}^d \times \mathbb{R}^{d'} \times \mathbb{R}^d \to \mathbb{R}$ calculates the likelihood that the fact holds true.

Optimization and Ranking Criteria The objective of optimization in KGE models is to refine the entity and relation embeddings such that they maximize the overall authenticity of observed triples in the KG. This often involves sophisticated ranking criteria and algorithms that balance the need for accuracy with computational efficiency.

Definition 5 (Loss function/Optimiser) Given a KG $G = \{(h, r, t)\} \subseteq E \times R \times E$, the loss function $L(G)$ measures the discrepancy between the scores generated using the encoder $\text{enc}_o(\cdot)$ and $\psi(\cdot)$, and the graph G. The optimiser aims to find θ^* such that:

$$\theta^* = \arg\min_{\theta} L(G)$$

2.1.2 KG Embedding Models

TransE TransE [Bor+13], a foundational model in the space of translation-based embeddings, conceptualizes relations as translations in the embedding space. For a triplet (h, r, t), the scoring function is defined as $f(h, r, t) = \|h + r - t\|$, where $\| \cdot \|$ denotes a norm (typically L1 or L2). This simplistic yet powerful premise enables TransE to efficiently model relationships as spatial translations, making it particularly effective for datasets where relations can be interpreted as movements from one entity to another.

RESCAL RESCAL stands out for its three-way tensor factorization technique, representing entities as vectors and relations as matrices within a KG. The scoring function for a triplet (h, r, t) in RESCAL is $f(h, r, t) = h^T M_r t$, where h and t are the vector embeddings of the head and tail entities, and M_r is the matrix embedding of the relation r. This function reflects the intricate multiplicative interplay between entity embeddings and relation matrices, allowing RESCAL to adeptly model diverse relational patterns with significant depth.

DistMult DistMult [Yan+14] streamlines RESCAL by imposing diagonal constraints on the relation matrices, enhancing efficiency and manageability. Its scoring function for a triplet (h, r, t) **is** $f(h, r, t) = h^T \mathbf{diag}(r) t$, with $\text{diag}(r)$ being a diagonal matrix containing the relation embedding r on its diagonal. This adaptation retains substantial modeling prowess for symmetric relations while bolstering DistMult's suitability for expansive KGs.

HolE HolE introduces the innovative use of circular correlation to unify entity and relation embeddings into a shared, compact space. Its scoring function employs circular correlation, denoted as $f(h, r, t) = r^T (h \star t)$, where $\star$ symbolizes the circular correlation operation. This unique method enables HolE to capture intricate entity-relation interactions, particularly excelling in representing asymmetric relationships.

ComplEx ComplEx [Tro+16] innovates by adopting complex-valued embeddings, enriching the model's capacity to explicitly represent asymmetric and reflexive relations. The scoring function for ComplEx is $f(h, r, t) = \mathbf{Re}(\langle h, r, \bar{t} \rangle)$, where $\langle \cdot \rangle$ signifies the dot product, $\text{Re}(\cdot)$ extracts the real component, and $\bar{t}$ is the complex conjugate of t. This approach allows ComplEx to adeptly navigate a broad spectrum of relational dynamics, including directional interactions and self-associations.

Structured embedding (SE) The Structured Embedding (SE) model represents an initial exploration into leveraging neural network paradigms for KG embeddings. It uniquely projects entities into spaces tailored to specific relations, enabling a nuanced understanding of diverse relational contexts. In SE, each relation r is linked to two projection matrices, $M_r^{(h)}$ and $M_r^{(t)}$, which respectively map head and tail entities into relation-specific embeddings. The scoring function for a triplet (h, r, t) in SE is articulated as $f(h, r, t) = -\|M_r^{(h)} h - M_r^{(t)} t\|_1$, where the L1 norm $\|\cdot\|_1$ quantifies the disparity between projected entity pairs. This model excels at distinguishing between different relation types by learning distinctive projections that emphasize the proximity of related entity pairs while distancing unrelated ones.

Convolutional 2D KG Embeddings (ConvE) ConvE [Det+18] propels the integration of convolutional neural networks into the realm of KG embeddings by ingeniously reshaping entity and relation embeddings into 2D matrices and applying convolutional operations. This innovative technique enables the model to discern complex interaction patterns by assimilating both spatial and compositional entity-relation features. The scoring function for ConvE involves a multi-step process: starting with the concatenation of head entity h and relation r embeddings, followed by a reshaping operation, the application of 2D convolutional layers, and culminating with a bilinear transformation against the tail embedding t. Formally, the scoring function is $f(h, r, t) = \mathbf{vec}(\mathbf{ReLU}(F * \mathbf{reshape}([h; r]) + b))^T W t$, where vec is the operation of converting a matrix into a vector, and W denotes the weights in the bilinear layer. ConvE's convolutional approach adeptly amalgamates a rich feature set extracted from entity-relation interactions, significantly enhancing the model's predictive fidelity.

QMult [Dem+21] QMult utilizes quaternion algebra for embedding entities and relations, providing a compact yet expressive representation. Its scoring function $QMult(h, r, t) = (e_h \otimes e_r) \cdot e_t$ utilizes quaternion embeddings e_h, e_r, and e_t for head, relation, and tail entities, respectively. The quaternion multiplication $\otimes$ blends the head entity with the relation, which is then projected onto the tail entity embedding through an inner product. This mechanism allows QMult to capture the intricate relationships within the KG by rotating and aligning embeddings in the quaternion space, optimizing for the closeness of the computed and true tail entity embeddings.

OMult [Dem+21] The OMult model leverages octonion algebra to enhance the representation of entities and relations in KGs. Its scoring function, defined as $OMult(h, r, t) = (e_h \star e_r) \cdot e_t$, where e_h, e_r, and e_t are the octonion embeddings of the head entity, relation, and tail entity respectively, employs octonion multiplication $\star$ to combine the head entity and relation embeddings, followed by an inner product with the tail entity embedding. This operation effectively captures the complex interactions within the graph by first rotating the head entity in the octonion space via the relation and then aligning it with the tail entity, aiming to minimize the distance for true triples during training.

Keci [DN23] Keci introduces an innovative approach to KG embeddings by utilizing Clifford algebras, offering a general framework for embedding entities and relations. Its scoring function $Keci(h, r, t) = (h \circ r) \cdot t$ employs multi-vector embeddings in a Clifford algebra space, where h, r, and t represent head entity, relation, and tail entity embeddings, respectively. The Clifford product $\circ$ fuses the head entity and relation embeddings, subsequently assessed against the tail entity embedding via a dot product. Keci's methodology encompasses the fusion and evaluation of embeddings to mirror the graph's relational structure accurately, focusing on optimizing these embeddings during training.

2.1.3 Types of Relations and their Effects on KG Embeddings

KGs consists of various types of relations that significantly impact the performance and complexity of embedding models. Understanding these relations is crucial for designing effective KG embeddings.

Symmetric relations are those where if a relation r holds from entity a to entity b, then it also holds from entity b to entity a. An example is the relation "isSiblingOf". If a is a sibling of b, then b is also a sibling of a. Models like DistMult are well-suited for symmetric relations due to their inherent geometric properties [Det+18].

Reflexive relations are those where an entity is related to itself. For instance, the relation "isPartOf" in a dataset might indicate that a region is part of itself, which is a reflexive relation. Reflexive relations are often found in datasets like YAGO, and their presence can be a cue for embedding models to learn identity mappings for such relations [LUO18].

Transitive relations indicate that if an entity a is related to b, and b is related to c, then a is also related to c. An example would be the "ancestorOf" relation. If a is an ancestor of b, and b is an ancestor of c, then a is also an ancestor of c. This type of relation is common in KGs like FB15k and WN18, and models that can capture transitive patterns, such as TransE, often perform well on such relations [Bor+13].

Antisymmetric relations are those where if a relation r holds from a to b, and b to a, then a and b must be the same entity. An example of an antisymmetric relation is "is equal to" in a graph representing mathematical entities. [Det+18]

Asymmetric relations are the opposite of symmetric ones; if a relation r holds from a to b, it cannot hold from b to a. An example of an asymmetric relation in a social network KG would be the "parent" relation, where if a is a parent of b, b cannot be a parent of a. [Bor+13]

Inverse relations mean that if a relation $r1$ holds from a to b, there exists a relation $r2$ which holds from b to a, and $r1$ is the inverse of $r2$. For example, "hasChild" is the inverse of "hasParent". Capturing inverse relations is challenging, and models like ComplEx are designed to handle such cases [Tro+16].

Hierarchical relations represent a parent-child relationship where one entity is subordinate to another. In the "employs" relation, if company a employs person b, b is subordinate to a. Hierarchical relations are prevalent in organizational datasets and require embedding models that can encode asymmetry, such as RotatE [Sun+19b].

Heterogeneous relations are those that connect entities of different types or domains. For example, in FB15k, the relation "wasBornIn" connects a person to a place. These relations add to the diversity of the graph and necessitate embedding models that can handle multi-modal data.

Composite relations involve multiple basic relations to express more complex interactions. They are often seen in richly structured KGs like YAGO and WN18RR, where relationships can be combinations of simpler relations.

Each type of relation introduces specific challenges and considerations for KG embedding models. The chosen embedding approach must align with the nature of the relations present in the KG to ensure that the learned embeddings are meaningful and accurate.

2.1.4 Scoring Techniques

In the domain of KG embeddings, scoring techniques are essential for evaluating the plausibility of triples, consisting of a head entity h, a relation r, and a tail entity t. A *positive triple* (h, r, t) is one that is true and exists within the KG, representing a valid fact. Conversely, a *negative triple* (h, r, t') is typically constructed by corrupting the head or tail entity of a positive triple, resulting in a fact that is likely false with respect to the KG's semantics.

$$\mathcal{T}^- = \left\{(h', r, t) \mid (h' \in \mathcal{E} \land \neg\exists (h', r, t) \in \mathcal{T}^+)\right\} \cup \left\{(h, r, t') \mid (t' \in \mathcal{E} \land \neg\exists (h, r, t') \in \mathcal{T}^+)\right\}.$$

Negative Sampling Negative sampling is a widely adopted technique in training KG embeddings [Bor+13], aimed at differentiating between positive and negative triples. It involves generating negative triples by altering either the head or the tail entity of positive triples, ensuring that the modified triple does not exist in the KG. This approach helps the model learn to score positive triples higher than the corresponding negative ones, improving its ability to discern true facts from false.

K vs All The K vs All scoring technique proposed by [Det+18] involves comparing a set of K randomly selected negative triples against all positive triples for a given relation r [NTK11]. This method allows for a focused evaluation where the model learns to distinguish between a smaller subset of negative examples and the entire set of positive facts, enhancing its specificity in scoring.

1 vs All In the 1 vs All approach [LUO18], for each positive triple (h, r, t), the model is tasked with scoring this triple against all other possible triples formed by keeping h and r constant while varying t. This exhaustive comparison helps fine-tune the model's ability to accurately score a wide range of potential triples, improving its overall predictive performance.

All vs All The All vs All scoring technique extends the idea of exhaustive comparison further by considering all possible combinations of head and tail entities for a given relation. This comprehensive approach ensures that the model is exposed to a broad spectrum of both positive and negative triples, enabling a deeper understanding of the KG's structure and the intricate relationships it encapsulates.

2.2 Neural Networks and Deep Learning

Neural Networks (NNs) are computational models inspired by the human brain, consisting of interconnected units or neurons that process data using a collection of mathematics and operations [GBC16]. Deep Learning, a subset of machine learning, involves the use of neural networks with multiple layers, which are capable of learning representations of data with multiple levels of abstraction [LBH15].

A general equation for the function approximation of an n-layered Neural Network with an activation function σ can be written as:

$$f(\mathbf{x}) = \sigma(\mathbf{W}_n f(\mathbf{x}_{n-1}) + \mathbf{b}_n)$$

where $f(\mathbf{x})$ is the output of the network, $\mathbf{W}_n$ is the weight matrix, $\mathbf{b}_n$ is the bias vector for the n-th layer, and $f(\mathbf{x}_{n-1})$ is the output of the previous layer after applying the non-linearity σ.

Following the foundational concepts of neural networks and deep learning, various architectures and layer types have been developed to address specific challenges in modeling data. These architectures are designed to capture different patterns within the data, while layer types are tailored to process the data in various forms.

2.2.1 Types of Neural Network Architectures

Chain Structured Networks Chain structured, or feedforward neural networks, are foundational architectures where information moves in only one direction from input to output. Each layer functions as a transformation stage for the preceding layer's output. They are widely used for their simplicity and effectiveness in tasks where the data points are independent of each other, such as image classification. Mathematically, a chain structured network with L layers can be expressed as $f(\mathbf{x}) = f_L(\ldots f_2(f_1(\mathbf{x})))$, where f_l represents the function of the l-th layer.

Residual Networks (ResNets) ResNets address the vanishing gradient problem encountered in training very deep networks by introducing skip connections. These connections allow gradients to flow through the network more effectively, enabling the training of networks with hundreds of layers. ResNets are particularly beneficial for tasks that benefit from very deep architectures, such as object detection and recognition in images [He+16b].This is mathematically represented as $f(\mathbf{x}) = \mathbf{x} + \mathcal{F}(\mathbf{x}, \{W_i\})$, where $\mathcal{F}$ is the residual function.

Dense Networks (DenseNets) DenseNets improve upon the idea of skip connections by connecting each layer to every subsequent layer, promoting feature reuse and reducing the number of parameters. This architecture is particularly useful when model compactness and efficiency are crucial, and it has shown significant results in areas such as image segmentation and classification [Hua+17].This can be represented as $\mathbf{x}_l = \mathcal{H}_l([\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{l-1}])$, where $\mathcal{H}_l$ is the transformation of the l-th layer, and $[\cdot]$ denotes concatenation

2.2.2 Types of Layers in Neural Networks

Fully Connected Layers Also known as dense layers, these are standard building blocks of neural networks where each input is connected to each output by a learnable weight. They are versatile and can learn any function representation but are parameter-intensive. Fully connected layers are typically used for tasks that do not require the preservation of spatial hierarchy in the input data, such as classification tasks after feature extraction. The output of a fully connected layer can be written as $\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$.

Convolutional Layers These layers use a set of learnable filters to perform convolution operations across the input. By preserving the spatial relationships in data such as images, they reduce the number of parameters compared to fully connected layers, leading to more efficient learning. Convolutional layers are the core building blocks of Convolutional Neural Networks (CNNs) used extensively in tasks like image and video recognition, image classification, and medical image analysis. A convolution operation for a single filter can be expressed as $\mathbf{h} = \sigma(\mathbf{W} * \mathbf{x} + b)$, where $*$ denotes the convolution operation.

Pooling Layers Pooling layers downsample the spatial dimensions (width, height) of the input, reducing the number of parameters and computations in the network. This operation helps to make the detection of features invariant to scale and orientation changes. Pooling layers are typically employed in CNNs to reduce the spatial size of convolved features, thus enabling the network to focus on the most salient features.The operation can be defined as $\mathbf{h} = \text{pool}(\mathbf{x})$, where pool is a function like max or average.

Recurrent Layers (LSTM/GRU) These layers are specifically designed for sequential data processing, where the current output depends on previous computations. LSTM and GRU units are adept at capturing long-range dependencies and are thus crucial for tasks like time-series prediction, natural language processing, and speech recognition.

Normalization Layers (e.g., Batch Normalization, Layer Normalization) Normalization layers are used to standardize the inputs to a layer for each mini-batch. This process stabilizes and accelerates the neural network's training. Batch normalization is especially effective for training deep networks, and combating issues like internal covariate shift. Batch normalization for a mini-batch can be represented as $\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu_{\mathbf{x}}}{\sqrt{\sigma_{\mathbf{x}}^2 + \epsilon}}$, where $\mu_{\mathbf{x}}$ and $\sigma_{\mathbf{x}}^2$ are the mean and variance of the batch[IS15].

2.2.3 Activation Functions

Activation functions are crucial components of neural networks, introducing non-linear properties that enable the network to learn complex mappings between inputs and outputs. Without these functions, a neural network would essentially become a linear regression model, incapable of solving non-linear problems.

Sigmoid The sigmoid function, defined as $\sigma(x) = \frac{1}{1+e^{-x}}$, compresses the input values to a range between 0 and 1. This characteristic makes it suitable for binary classification problems and for the output layer of networks predicting probabilities. However, its usage in hidden layers has diminished due to the vanishing gradient problem, where gradients can become very small, effectively halting the network's training process.

Tanh The hyperbolic tangent function, or $\tanh$, outputs values in the range of -1 to 1, making it a scaled version of the sigmoid function. It is mathematically defined as $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$. The $\tanh$ function is often preferred over sigmoid for hidden layers because its output is zero-centred, leading to a more efficient training process. However, like the sigmoid function, it can also suffer from the vanishing gradients problem in deep networks.

Rectified Linear Unit The Rectified Linear Unit (ReLU) function, defined as $\text{ReLU}(x) = \max(0, x)$, has become one of the most widely used activation functions in deep learning due to its computational simplicity and efficiency. ReLU helps mitigate the vanishing gradient problem, allowing models to learn faster and perform better. However, it is not without its drawbacks, such as the "dying ReLU" problem, where neurons can sometimes become inactive and stop learning entirely.

2.2.4 Regularization and Dropout

Regularization techniques play a pivotal role in enhancing the generalization capability of neural networks by mitigating the risk of overfitting. Overfitting occurs when a model learns the training data too well, including its noise and outliers, which results in poor performance on unseen data. Regularization addresses this issue by adding a penalty on the magnitude of the parameters or by introducing randomness in the learning process, encouraging the model to learn more robust features.

Weight Decay Weight decay, often associated with the L2 regularization technique, works by adding a penalty term to the loss function proportional to the square of the magnitude of the weights. This penalty term discourages the weights from growing too large, promoting simpler models that are less likely to overfit. The modified loss function with weight decay can be expressed as $L' = L + \frac{\lambda}{2}||\mathbf{W}||^2$, where L is the original loss, λ is the regularization coefficient, and $||\mathbf{W}||^2$ represents the L2 norm of the weight matrix $\mathbf{W}$.

Dropout Dropout is a unique and powerful regularization technique introduced by [Sri+14] that randomly "drops out" a subset of neurons in the network during each training iteration. This means that the dropped neurons do not participate in the forward pass and their contribution to the backpropagation is temporarily removed. Mathematically, this can be represented as $\mathbf{h}' = \mathbf{h} \odot \mathbf{r}$, where $\mathbf{h}$ is the original output vector of a layer, $\mathbf{r}$ is a vector of random binary values (with a probability p of being 1), and $\odot$ denotes element-wise multiplication. This technique prevents neurons from co-adapting too much to their context, forcing the network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons. As a result, dropout can significantly improve the generalization performance of large neural networks.

Both weight decay and dropout offer effective means to control overfitting, but they work in fundamentally different ways. Weight decay is more straightforward and mathematically grounded in its approach to regularization, while dropout introduces stochasticity into the network, providing a form of ensemble learning at a fraction of the cost. The choice between them—or the decision to use both—depends on the specific problem, dataset, and neural network architecture.

2.2.5 Backpropagation and Weight Update

Backpropagation is a fundamental algorithm for training neural networks, particularly in the context of KG embeddings, where the model parameters include the weight vectors of head entities, relations, and tail entities. The essence of backpropagation is the efficient computation of gradients of the loss function with respect to each parameter in the network, which is then used to update the model parameters in the direction that minimizes the loss.

The weight update rule in gradient descent, the simplest form of optimization, can be expressed as:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L$$

where θ represents the model parameters (including weights), η is the learning rate, and $\nabla_\theta L$ is the gradient of the loss function L with respect to the parameters. The learning rate η controls the size of the steps taken towards the minimum of the loss function, playing a critical role in the convergence of the optimization algorithm.

Issues in Gradient Descent Gradient descent can face challenges like local minima, saddle points, and particularly the vanishing gradient problem, where gradients for parameters in the earlier layers become exceedingly small, leading to negligible updates. Recent advances in optimizers have shown that these issues can be overcome by using momentum and gradient accumulation techniques.

Optimizers [Sun+19a] talks about different optimizers and their usage.
SGD (Stochastic Gradient Descent) SGD updates the model parameters using only a single training example at a time, which helps reduce the computation cost at each iteration:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; x^{(i)}, y^{(i)})$$

where $(x^{(i)}, y^{(i)})$ is the i-th training example.

AdaGrad AdaGrad adapts the learning rate for each parameter, allowing for larger updates for infrequent parameters:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \nabla_\theta L$$

where G_t is a diagonal matrix where each diagonal element is the sum of the squares of the gradients up to time t, and ϵ is a small smoothing term to avoid division by zero.

RMSprop RMSprop modifies AdaGrad to perform better in non-convex functions by changing the gradient accumulation into an exponentially weighted moving average:

$$G_t = \beta G_{t-1} + (1 - \beta)(\nabla_\theta L)^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_\theta L$$

Adam Adam combines the ideas of momentum and RMSprop, maintaining both first and second moment estimates of the gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta L$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta L)^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

where m_t and v_t are estimates of the first and second moments of the gradients, β_1 and β_2 are the exponential decay rates for these moment estimates, and t denotes the timestep.

Each optimizer addresses specific issues in gradient descent and improves upon them in various ways, such as by adapting the learning rate or by incorporating momentum to avoid local minima and saddle points.

2.3 Neural Architecture Search

Description: Overview of popular NAS frameworks and tools that facilitate the NAS process.

Neural Architecture Search (NAS) is a field of study in machine learning that focuses on automating the design of artificial neural networks. Traditional approaches to designing neural network architectures involve a considerable amount of manual effort and domain expertise. NAS aims to simplify this process through automation, enabling the discovery of optimal network architectures for a given task.

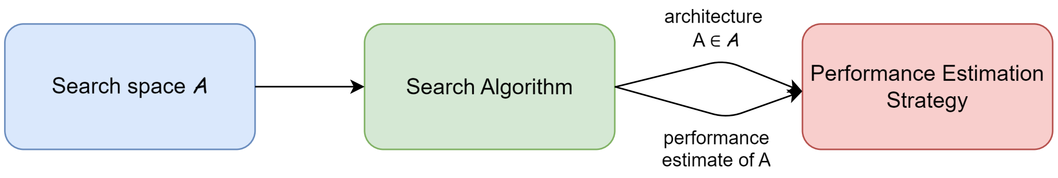

Search space $\mathcal{A}$
Search Algorithm
architecture $A \in \mathcal{A}$
performance estimate of A
Performance Estimation Strategy

Fig. 2.2: The general framework of Neural Architecture Search.

As illustrated in Figure 2.2, NAS involves a search space $\mathcal{A}$, a search algorithm, and a performance estimation strategy. The search algorithm explores the search space to select candidate architectures $A \in \mathcal{A}$ and utilizes the performance estimation strategy to evaluate their efficacy.

2.3.1 Search Space

The search space defines the set of all possible architectures that the search algorithm can explore. It is parametrized by several factors including:

- The (maximum) number of layers n, which could be bounded or unbounded.
- The type of operation executed by each layer, such as pooling, convolution, or more advanced operations like depthwise separable convolutions or dilated convolutions.
- Hyperparameters associated with each operation, for example, the learning rate, optimizer, activation function, the number of filters, kernel size, and strides for a convolutional layer and others which can influence the training process.

Example: In the context of KG embeddings, the search space may include architectures with varying numbers of convolutional layers with different kernel sizes and numbers of filters, potentially interspersed with pooling layers or recurrent layers to capture sequential patterns within the graph structure [EMH19].

2.3.2 Search Strategy

The search strategy refers to the algorithmic approach used to explore the search space. Various strategies have been employed in NAS including:

- **Evolutionary Algorithms (EAs)** These algorithms are inspired by the process of natural selection. They start with a population of candidate architectures and iteratively apply operations analogous to biological mutation and crossover to generate new architectures. Selection processes are then used to retain the most promising architectures. EAs are particularly powerful for exploring large, complex search spaces due to their ability to escape local optima and explore diverse solutions. However, they can be computationally expensive due to the need to evaluate many candidate architectures across generations [Rea+19]
- **Heuristic-Based Search** This approach uses specific rules or heuristics to guide the search process. Heuristics can be based on domain knowledge or empirical observations and can help reduce the search space by focusing on more promising regions. While this can speed up the search process, it may also introduce bias and limit the exploration of potentially better architectures outside the heuristics' scope.
- **Random Search** In random search, candidate architectures are selected randomly from the search space. This strategy is surprisingly effective as a baseline and can sometimes outperform more complex algorithms, especially in cases where the search space is not well understood. Its simplicity makes it easy to implement and parallelize across multiple computational resources.[BB12] used random search and compared it with grid search .
- **Bayesian Optimization** Bayesian optimization treats the search as a probabilistic model and aims to find the optimal architecture by taking into account the posterior probability of the performance given the architectures evaluated so far. It balances exploration and exploitation by selecting architectures that are expected to offer the most informative results. This method is sample-efficient, making it suitable for scenarios where evaluating architectures is costly.[SLA12] talks about bayesian optimization of ML algorithms.
- **Reinforcement Learning (RL)** [ZL16] RL-based search strategies treat the architecture design process as a sequential decision-making problem. An RL agent is trained to construct architectures by making a sequence of decisions, where each decision corresponds to a part of the architecture, such as the

selection of a layer type or hyperparameter. The agent receives rewards based on the performance of the architectures it generates, guiding it to make better decisions over time. This strategy has been successful in discovering high-performing architectures but requires careful design of the reward function and can be sample-inefficient.

2.3.3 Performance Estimation Strategy

In the case of KG embeddings, the performance of an architecture is often evaluated using metrics such as the Mean Reciprocal Rank (MRR). The goal is to maximize this metric, which measures the ability of the model to rank the true linking entity high among a list of candidate entities. The MRR is especially relevant when the output space is large, as is common in KGs.

2.3.4 NAS Frameworks

With the growing interest in automating the design of neural network architectures, several frameworks have emerged to facilitate Neural Architecture Search (NAS). These frameworks provide tools and environments for designing, comparing, and optimizing neural network architectures efficiently. One notable example is Neural Network Intelligence (NNI) by Microsoft.

Neural Network Intelligence (NNI) [3]

Neural Network Intelligence (NNI), developed by Microsoft, is an open-source toolkit designed to support various NAS algorithms and strategies. It provides a rich set of features for defining search spaces, running experiments, and managing and comparing different NAS trials. NNI is highly extensible, allowing researchers and practitioners to implement custom models, and training strategies, and even integrate new NAS algorithms.

It offers a user-friendly experience, setting it apart from other automated machine-learning solutions. NNI's ability to run efficiently on local machines without the need for GPUs is ideal for quick experimentation. It performs thorough explorations by running numerous trials, each assessing distinct parameter combinations, building models from the ground up, and logging performance metrics. This process

[3] https://github.com/microsoft/nni

is enhanced by NNI's WebUI, which allows for easy review of results. NNI can be utilized through its API or Python Annotation, with the latter providing an innovative yet experimental approach to neural architecture search. While the API method involves a JSON configuration, the Python Annotation allows for in-script parameter specification, which is particularly handy for architecture search. Despite the experimental status of the annotation method, the API approach remains effective for our evaluation needs.

NNI comes with built-in support for popular NAS methods like reinforcement learning, evolutionary algorithms, and gradient-based optimization. It simplifies the experimentation process through a user-friendly interface and an experiment management system that helps in tracking and analyzing the performance of different architecture search runs.

Other Frameworks

Other frameworks in the field of NAS include:

Google Vizier A service used extensively within Google for hyperparameter tuning and as part of their NAS framework[4].

AutoKeras Built on Keras, AutoKeras[5] simplifies the application of NAS with a user-friendly interface.

Auto-PyTorch This framework[6] applies NAS in conjunction with PyTorch to automatically generate well-performing models.

Auto-Sklearn As an automated machine learning toolkit that leverages Scikit-Learn, Auto-Sklearn[7] incorporates NAS to optimize model selection and hyperparameter tuning.

These frameworks, each with their unique features and capabilities, are at the forefront of automating neural network architecture design, greatly simplifying the process of developing and fine-tuning machine learning models.

[4] `https://cloud.google.com/ai-platform/optimizer/docs/overview`
[5] `https://autokeras.com/`
[6] `https://www.automl.org/automl/auto-pytorch/`
[7] `https://automl.github.io/auto-sklearn/master/`

3 Approach and Methodology

In this chapter, we explain the overall approach used in this research. The approach involves encoding entities $h, t \in \mathcal{E}$ and relations $r \in \mathcal{R}$ using the encoding function $\text{enc}_\theta = f(\theta, x)$. This function maps the entities and relations into their respective embeddings within spaces such as polynomial, complex, and neural network function spaces. The embeddings $f(\theta_h, x)$,$f(\theta_t, x)$ and $f(\theta_r, x)$, produced by enc_θ, are then utilized by various scoring functions—compositional, trilinear, or vector triple product—to generate a scalar score. Given the functional nature of our embeddings, we employ numerical integration techniques to accurately compute the integral of these embeddings over the domain of x. Specifically, we experimented with both the trapezoid quadrature rule and Gaussian quadrature methods to evaluate their effectiveness in this context. This approach enables us to effectively utilize the continuous nature of function-based embeddings, allowing for a more nuanced assessment of triple scores. The scoring function, adapted to accommodate this numerical integration process, is then applied to both positive triples and their corrupted counterparts to generate the respective scores. These scores are subsequently fed to the loss function for optimizing the model parameters θ using the loss calculated by comparing true and negative triple. Here we worked on three loss functions —Margin ranking-based, L2 norm-based loss function and binary cross entropy with logits loss — and compared the results. To find the efficient design of a neural network approximator, neural architecture search (NAS) is employed

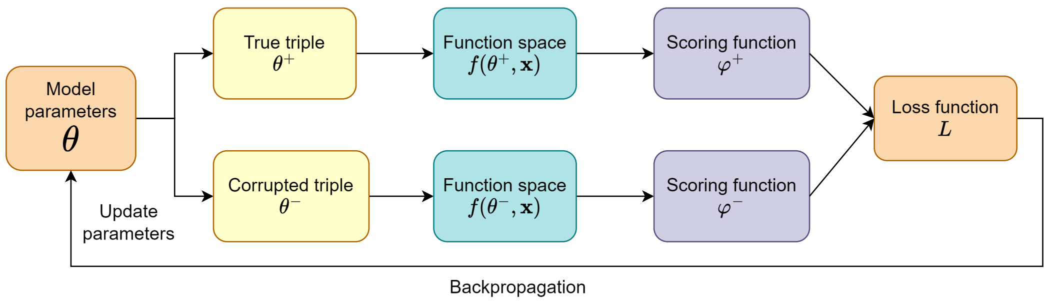

Model parameters θ
True triple θ^+
Corrupted triple θ^-
Function space $f(\theta^+, \mathbf{x})$
Function space $f(\theta^-, \mathbf{x})$
Scoring function φ^+
Scoring function φ^-
Loss function L
Update parameters
Backpropagation

Fig. 3.1: Schematic overview of the embedding as functions KG model. The diagram delineates the flow from model parameter initialization to the computation of loss for both true T^+ and corrupted triples T^-. Each triple is mapped through its respective function space, processed by the scoring function, and evaluated against the loss function, which informs parameter updates via backpropagation

to automate the discovery of optimal hyperparameters using strategies such as the definition of the search space and the search algorithm.

As depicted in the schematic figure 3.1, the model parameters θ_h, θ_r, and θ_t for head entities, relations, and tail entities, respectively, are initialized to project into a d-dimensional function space. For each positive triple, k negative triples are generated and processed through their respective function spaces $f(\theta_h, x)$, $f(\theta_r, x)$, and $f(\theta_t, x)$ to obtain the embeddings as a function where θ_h, θ_r and θ_t are model parameters associated with head, relation and tail entities respectively.

These embeddings are then evaluated by scoring functions ψ. The embedding dimension d plays a pivotal role, in determining the level of detail captured by the representation. While higher dimensions may afford a more detailed representation, they can also lead to increased computational demands and the risk of overfitting.

The continuous learning process involves updating the parameters θ_h, θ_r, and θ_t based on the loss function L, as informed by the comparison of true and corrupted triples within the knowledge graph embedding model. This iterative process of parameter adjustment through backpropagation is critical for refining the embeddings and enhancing the model's predictive performance.

3.1 Function Spaces

Knowledge graph embeddings traditionally represent entities and relations as static vectors within a low-dimensional space. Established methods like TransE [Bor+13] and DistMult [Yan+14] have affirmed the efficacy of this vector-based representation. Nonetheless, to encapsulate the intricate and complex relationships present in knowledge graphs more effectively, we propose a paradigm shift where embeddings are conceived as functions of model parameters θ over a domain X.

In our novel framework, embeddings are realized not as vectors but as functions $f(\theta, X)$, with θ denoting the model parameters associated with the head entity, tail entity, or relation within the graph, and X representing domain. To evaluate these function spaces and accurately capture the nuanced relationships they represent, we employ a numerical quadrature that approximates the embedding functions over the domain X using sample points x.

This section delves into three distinct function spaces that our research explores, including two baseline spaces and one neural network-based function space:

- Polynomial Function Space
- Complex Number Function Space
- Neural Network Function space

Each function space provides a unique approach to representing and processing the data of knowledge graphs, thereby facilitating a more profound comprehension of the graph's inherent structure.

3.1.1 Polynomial Function Space

In the context of knowledge graph embeddings, we leverage a polynomial function space to map entities and relations into a d-dimensional polynomial function space. Here, the model parameters θ associated with entities and relations are conceptualized as coefficients of polynomial functions.

Theoretical Underpinnings A polynomial function, in a single variable x, is expressed as:

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{d-1} x^{d-1}, \tag{3.1}$$

where $a_0, a_1, \ldots, a_{d-1}$ represent the coefficients that define the polynomial's characteristics, and d denotes the dimensionality, determining the function's complexity and capacity to encapsulate information.

Embedding Representation via Polynomial Functions In our approach, the polynomial coefficients $a_0, a_1, \ldots, a_{d-1}$ are directly derived from the model parametersθ corresponding to the head, relation, and tail entities. This formulation embeds these entities and relations as entire polynomial functions over a domain X, rather than as discrete points or vectors. To operationalize this within computational constraints, we may approximate the evaluation of these polynomials at specific points within X using numerical quadrature techniques, such as the trapezoid or Gaussian quadrature rules, to effectively capture the essence of these polynomial functions.

For an entity or relation represented by a parameter vector θ, its embedding is theoretically conceptualized as a polynomial function spanning the entire domain X. This embedding is represented by the function:

$$f_{\text{polynomial}}(\theta, x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_{d-1} x^{d-1}, \tag{3.2}$$

where the function is defined for all x within the domain X. In practice, the continuous nature of this function necessitates that we approximate its evaluation at a finite set of sample points within X, using numerical methods such as numerical quadrature, to obtain a practical representation of the embedding.

For each sample point x within the domain X:

$$P(x_i) = \sum_{k=0}^{d-1} \theta_k x_i^k \tag{3.3}$$

where θ is the d dimensional parameter.

This results in a set of polynomial values that map the entity and relation d dimensional parameter vector into the embedding as a polynomial function.

Example To illustrate this process, let us consider a knowledge graph triple that includes the entities "$Newton$" and "$Gravity$", and the relation "$Discovered$".

For example, the parameters for "$Newton$" as $\theta_h = [1.2, -0.5]$, which resides in a 2D space. Accompanying this, we select a sample points x, $x = \{0.1, 0.2\}$ from a subdomain $X, [-1, 1]$, which provides the points at which we wish to evaluate the polynomial embedding.

Our task is now to transform the head entity h into its corresponding embedding as a polynomial function representation. We do this by computing the value of the polynomial at each point in x within the domain X:

$$P(0.1) = 1.2 \cdot (0.1)^0 - 0.5 \cdot (0.1)^1.$$

$$P(0.2) = 1.2 \cdot (0.2)^0 - 0.5 \cdot (0.2)^1.$$

Through these calculations, we derive a set of values that articulate the transformation of the head entity parameters θ_h into a new set of points, $\{P(0.1), P(0.2)\}$, which we denote as f_h. This embedding f_h is the polynomial transformation of the entity parameter vector and represents how the entity "Newton" is depicted within the range of the polynomial function space. It is these transformed representations that will be subsequently utilized within our scoring function to evaluate the validity of the triple within the knowledge graph.

3.1.2 Complex Number Function Space

The Complex number function space leverages the unique properties of complex numbers to enhance the representation of embeddings in knowledge graphs. In this space, embeddings encapsulate not only the magnitude but also the phase information, offering a comprehensive depiction of entities and relations that go beyond what real numbers alone can convey.

Theoretical Underpinnings Mathematically, a complex number consists of a real part and an imaginary part and is typically expressed as:

$$z = a + ib,$$

where a is the real part, b is the imaginary part, and i is the imaginary unit with the property $i^2 = -1$, and $z \in \mathbb{C}$.

Encoding Using Complex Number Function To embed an entity or relation into the complex number function space, we consider the cosine and sine transformations over the domain X. This dual transformation encapsulates the phase information of the entity or relation. The embedding as a complex number function over a domain X is given by:

$$f_{complex}(\theta, x) = \sum_{k=0}^{d-1} \theta_k \cdot \cos(x) + i \cdot \sum_{k=0}^{d-1} \theta_k \cdot \sin(x). \tag{3.4}$$

Here, θ_k are the components of the model parameter vector $\theta \in \mathbb{R}^d$, and $f_{complex}(\theta, x) \in \mathbb{C}$. This expression calculates the sum of cosine and sine transformations for each dimension of the model parameter vector, producing a complex number that captures the magnitude and phase of the entity or relation within domain$X$.

The encoding function, denoted by enc_θ, transforms the model parameters of entities and relations into a complex space. For a model parameter vector θ over the domain X, the real and imaginary components are obtained by element-wise multiplication of θ with $\cos(x)$ and $\sin(x)$, respectively. These components are then combined to form a complex number representation $\theta_{\mathbb{C}} = \Re(\theta) + i\Im(\theta)$, where i is the imaginary unit.

Transformation Process The transformation process involves calculating the real and imaginary parts of the complex representation for each sample point in X, using numerical quadrature for approximation when necessary. The complex number z for each sample point is computed as follows:

$$\text{real art} = \sum_{k=0}^{d-1} \theta_k \cdot \cos(x^k), \tag{3.5}$$

$$\text{imaginary part} = i \cdot \sum_{k=0}^{d-1} \theta_k \cdot \sin(x^k). \tag{3.6}$$

The final complex representation for each sample point merges these parts, maintaining the dimensionality of the original model parameter vector. For the purposes of the scoring function within the KG embedding model, we utilize the real component $\Re$ of the embedding to generate the scores. **Example** Consider the knowledge graph triple ("Einstein", "Influenced", "Bohr") as an instance to illustrate the complex number function space. In this case, the entities "Einstein" and "Bohr" are elements of the set $\mathcal{E}$, which contains all entities in our knowledge graph, while "Influenced" is part of the set $\mathcal{R}$, which includes all relations. For this example, suppose the parameters for the head entity "Einstein" (θ_h) are given by the 2-dimensional vector $\theta_h = [0.5, 0.3]$. We select the sample points $x = \{0.1, 0.2\}$ from the domain X. We aim to transform the head entity "Einstein" into its embedding as a complex number function at each sample point x_i in X. The transformation is computed as follows:

$$\begin{aligned}
&\text{For } x_i = 0.1:\\
&\qquad z(0.1) = (0.5 \cdot \cos(0.1) + i \cdot 0.5 \cdot \sin(0.1)) + (0.3 \cdot \cos(0.1) + i \cdot 0.3 \cdot \sin(0.1)),\\
&\text{For } x_i = 0.2:\\
&\qquad z(0.2) = (0.5 \cdot \cos(0.2) + i \cdot 0.5 \cdot \sin(0.2)) + (0.3 \cdot \cos(0.2) + i \cdot 0.3 \cdot \sin(0.2)).
\end{aligned}$$

These calculated complex numbers $z(0.1)$ and $z(0.2)$ provide a detailed representation of "Einstein" within the complex number function space, encapsulating both magnitude and phase information derived from the model parameters. This enhanced representation is crucial for our scoring function, which evaluates the plausibility of triples within the knowledge graph by considering both amplitude and phase information.

3.1.3 Neural Network Function Space

Neural networks, central to the field of deep learning, have gained immense popularity across various disciplines due to their potent representational and generalization capabilities. These models excel at mapping complex nonlinear relationships, making them highly effective for tasks that involve intricate data patterns. The application of neural networks to embed knowledge graphs into continuous feature spaces has recently attracted considerable attention, signifying a promising direction in knowledge graph research [Dai+ar].

In this approach, the parameters of entities and relations, denoted by θ, are utilized as the weights $\mathbf{W}$ within the neural network, leading to the generation of embeddings as a neural network function $f(\mathbf{W}, X)$ over a domain X.
Note In the context of neural networks, the model parameters θ are referred to as weights due to their role in the network's architecture and the common terminology used in machine learning.

Neural Network Architecture The devised neural network architecture comprises multiple layers, each associated with a unique weight matrix $\mathbf{W}_n$, constructed from the d-dimensional model parameters $\theta \in \mathbb{R}^d$. The configuration of these weight matrices is dynamically determined by the number of layers n and the dimensionality d of the model parameter (θ^d).

θ^d **adjustment** To ensure an even distribution of weights across layers, the network adjusts the dimensionality of the model parameters d as follows:

$$d' = d + \Delta d, \tag{3.7}$$

where d' represents the adjusted dimension of the model parameters, and Δd is the minimal increment required to make $\frac{d'}{n}$ a perfect square, facilitating the construction of evenly sized weight matrices for each layer, given the total number of layers n.

Layer Matrix Size The dimension of the square weight matrix for each layer is given by:

$$k = \sqrt{\frac{d'}{n}}, \tag{3.8}$$

where k denotes the size of each layer's weight matrix, ensuring a uniform architecture conducive to efficient computation and effective learning.

In this research, we tried two types of network architectures namely chain-structured and residual network architecture.

Chain-Structured Architecture In this architecture, the network sequentially applies the activation function after every layer, enhancing non-linearity at each step. Layer normalization is also integrated post-activation to ensure stability throughout the learning process. The operation of this network over a domain X is described by:

$$f(\mathbf{W}, X) = \sigma(\mathbf{W}_n(\sigma(\mathbf{W}_{n-1}(\dots\sigma(\mathbf{W}_2(\sigma(\mathbf{W}_1 X))))))). \tag{3.9}$$

In this formulation, σ denotes the activation function applied consistently following each layer's transformation. Using the numerical quadrature, we select sample points x, over a subdomain with finite points, where these x act as the input to the neural network, while the model parameters θ act as weights W of the neural network.

Residual Network (ResNet) Architecture [He+16a] The ResNet architecture introduces skip connections, enabling the network to potentially bypass one or more layers for enhanced gradient flow and mitigation of the vanishing gradient problem. The operation of ResNet is articulated as:

$$f(\mathbf{W}, X) = \mathbf{W}_n\sigma(\mathbf{W}_{n-1}\sigma(\dots(\mathbf{W}_2\sigma(\mathbf{W}_1 X + X) + \mathbf{W}_1 X))). \tag{3.10}$$

In this formulation, X is added to the output of certain layers, creating a skip connection that combines the linear transformation and the activation function's output with the input of that layer.

Learning Process and Practical Implementation The optimization of entity and relation parameters θ is conducted using the Adam optimizer with a predefined learning rate. The network is constructed with n layers, each employing a k, k weight matrix, and processes batches of triples during training.

This neural network function space offers a potent method to enhance knowledge graph embeddings, utilizing the depth of neural network architecture to encapsulate the complex relationships present in the data.

Integration of Neural Architecture Search (NAS)

To enhance the performance of our neural network function space model for knowledge graph embeddings, neural architecture search (NAS) is employed. NAS enables the dynamic adaptation of the network's architecture and hyperparameters, directly influencing the learning efficacy and outcome.

Framework Utilization The neural network intelligence (NNI) [Mic21] framework was leveraged to facilitate the NAS process. NNI provides a comprehensive platform for launching, monitoring, and refining neural architecture search experiments. It integrates seamlessly with various search algorithms, allowing for efficient exploration and evaluation of different hyperparameter combinations.

Layer Manipulation and Hyperparameter Optimization In our neural architecture search (NAS) framework, a process of layer manipulation and hyperparameter optimization was undertaken to enhance the performance of the knowledge graph embedding model. We experimented with varying the number of layers and exploring different types of layers, primarily focusing on fully connected layers and convolution layers accompanied by pooling to capture a wide range of feature interactions within the graph structure.

Throughout the optimization process, we adjusted several hyperparameters to fine-tune the network's learning capabilities. The learning rate (lr) was carefully calibrated, ranging from 0.1 to 0.001, to ascertain the optimal speed at which the model updates its weights in response to the loss gradient. The activation functions were chosen to introduce non-linearity into the model, allowing it to learn complex patterns beyond linear boundaries. We also manipulated the embedding dimension, providing the model with varying degrees of freedom to represent the entities and relations.

Further, the stride and kernel sizes of convolutional layers were varied to observe their effects on the model's ability to abstract features at different levels. The number of output channels was another consideration, impacting the width of the network and its representational power. To mitigate the risk of overfitting, we employed different dropout rates, introducing a form of regularization by randomly omitting a subset of features during training.

Each of these hyperparameters plays a vital role in the learning process, and their optimal combination can significantly influence the model's performance on knowledge graph embedding tasks. Therefore, extensive experimentation was conducted to identify the most effective hyperparameter settings within the NAS framework.

Additional hyperparameters like weight decay, batch sizes were also modified to refine the training process further.

Search Algorithms In the exploration of hyperparameter space, several search algorithms were utilized, each with its distinct methodology and underlying principles.

The Tree-structured Parzen Estimator (TPE), a Bayesian optimization technique, models the relationship between hyperparameters and the probability of achieving a certain performance metric. It is adept at handling complex search spaces by building probabilistic models that guide the search towards promising areas [Ber+11].

Grid search, a traditional approach, systematically explores the predefined hyperparameter space. Despite its simplicity and exhaustiveness, it may not be efficient for large, high-dimensional spaces.

Random search, in contrast, explores the hyperparameter space randomly. While less systematic than grid search, it can often reach near-optimal solutions more quickly, especially in large search spaces [Ber+11].

Evolutionary algorithms, inspired by the process of natural selection, apply mechanisms such as mutation, crossover, and selection to evolve a set of solutions towards better hyperparameters. This heuristic approach is known for its ability to navigate complex and multimodal landscapes effectively [Rea+17].

Each of these algorithms offers unique advantages in navigating the search space, aiming to identify the most effective set of hyperparameters for the neural network.

Evaluation Metric for NAS The model's performance is evaluated using the mean reciprocal rank (MRR), a standard metric in knowledge graph embeddings. MRR provides insight into the model's predictive accuracy and is crucial for guiding the NAS process. For the tuner configuration, the metric is set to maximize mode so that the tuner tries to find the configuration which maximizes this evaluation metric.

3.2 Scoring Functions

Scoring functions in knowledge graph embeddings aim to assign scalar scores to triples consisting of entities and relations. These scores measure the likelihood of a triple being factual within the knowledge graph. The higher the score, the more plausible the triple. In practice, a positive triple, a fact from the knowledge base, is usually compared against several negative triple, which are generated by corrupting the positive triple. The goal of the scoring function is to assign a higher score to the positive triple compared to the negative ones.

To obtain these scores, the entities and relations are mapped into embeddings as a function using function spaces as discussed in the previous section and then are used by different sets of scoring functions ψ that will be discussed in this section.

This section delves into three distinct scoring functions that our research explores, namely

- Compositional Scoring Function
- Vector Triple Product Scoring Function
- Trilinear Product Scoring Function

3.2.1 Compositional Scoring Function

In the context of knowledge graph embeddings, compositional functions reflect the interaction between entities (such as head and tail) and relations as a composite function. Typically in mathematics, a compositional function is denoted as $f(g(x))$ [1], where one function is applied to the results of another.

For our adapted approach, the compositional scoring function is not merely a multiplication of embeddings but rather a sequence of function applications that reflect the compositional nature of relations and entities within the knowledge graph. Specifically, we generate a composite relation embedding function $(f(\theta_r, f(\theta_h, x)))$ by substituting the sample points x of the finite domain X with the head embedding function $f(\theta_h, x)$. This composite function is then element-wise multiplied by the tail embedding function $f(\theta_t, x)$, and the score is derived by integrating the product of these functions over the domain X, using numerical quadrature:

$$\psi_{compositional} = \int_X f(\theta_r, f(\theta_h, x)) \circ f(\theta_t, x)\, dx.$$

Here $\theta_h, \theta_r, \theta_t$ are the parameters of head, relation and tail entity, In this expression, ψ denotes the scoring function, and $\circ$ symbolizes the element-wise (Hadamard) product. This approach effectively models the interaction between the head and tail entities mediated by the relation[].

[1]https://en.wikipedia.org/wiki/Function_composition

Example

Consider a knowledge graph triple (h, r, t) with the following 2D parameters θ:

$$\theta_h = \begin{bmatrix} 1.2 \\ -0.5 \end{bmatrix}, \quad \theta_r = \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix}, \quad \theta_t = \begin{bmatrix} 0.2 \\ 0.9 \end{bmatrix}$$

To compute the score using our compositional scoring function, we proceed as follows:

- Apply the function space transformation to entity h using the associated parameters θ_h, obtaining embedding $f(\theta_h, x)$.
- Apply the compositional transformation on relation using head embedding, getting $f(\theta_r, (f(\theta_h, x))$.
- Apply the function space transformation to entity t, obtaining $f(\theta_t, x)$.
- Compute the score by integrating the Hadamard product of $f(\theta_r, f(\theta_h, x))$ and $f(\theta_t, x)$ over a set of sample points x with a finite sub domain X using numerical quadrature.

3.2.2 Trilinear Scoring Function

The trilinear scoring function is articulated through an integral that encompasses the element-wise (Hadamard) product of the embeddings corresponding to the head, relation, and tail entities over the domain X:

$$\psi_{trilinear} = \int_X f(\theta_h, x) \circ f(\theta_r, x) \circ f(\theta_t, x)\, dx,$$

where $f(\theta_h, x)$, $f(\theta_r, x)$, and $f(\theta_t, x)$ represent the embeddings as a function for the head, relation, and tail entities, respectively, evaluated within the domain X. This formulation can also be denoted using the triple product notation $\langle f_h, f_r, f_t \rangle$, emphasizing the trilinear interaction among the embeddings.

For any given triple (h, r, t), the entities and relation are first mapped to their corresponding function spaces, yielding f_h, f_r, and f_t. These function-based embeddings undergo an element-wise multiplication, capturing the interaction between the head, relation, and tail. The integration of this product over the domain X results in a scalar score, which quantitatively assesses the strength or validity of the relational link between the entities:

This scalar score, derived from the definite integral of the trilinear product over the specified domain, serves as a robust measure of the triple's compatibility within the knowledge graph framework, effectively leveraging the continuous nature of the embeddings in the function space.

3.2.3 Vector Triple Product (VTP) Scoring Function

The Vector Triple Product (VTP) scoring function adopts a mathematical framework that leverages the vector triple product operation to evaluate the interactions within a knowledge graph triple. The VTP operation is foundational in vector algebra and is described as follows [2]:

$$\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = (\mathbf{c} \cdot \mathbf{a})\mathbf{b} - (\mathbf{b} \cdot \mathbf{a})\mathbf{c}.$$

In the context of knowledge graph embeddings, the VTP scoring function is adapted to assess the compatibility of triples by computing the embeddings as functions over the domain X. This theoretical framework employs the vector triple product to generate a scalar score from the continuous interaction of head, relation, and tail embeddings:

$$\psi_{vtp} = -\int_X f(\theta_h, x)\, dx \times \int_X f(\theta_t, x) \circ f(\theta_r, x)\, dx + \int_X f(\theta_r, x)\, dx \times \int_X f(\theta_t, x) \circ f(\theta_h, x)\, dx.$$

For practical computation, the embeddings as functions are approximated over a finite subdomain of X using numerical quadrature. This approximation involves defining a sample space x over X and calculating the definite integrals of the function-based embeddings within this sample space. The resulting integrals are then combined according to the VTP formula adapted for our embeddings to produce the final scalar score.

Example Given a triple, the model parameters for the head (h), relation (r), and tail (t) entities are transformed through their respective function spaces, denoted as

[2] https://en.wikipedia.org/wiki/Triple_product

f_h, f_r, and f_t. Utilizing the adapted VTP formula within these function spaces, the definite integrals over the domain X are computed. These integrals are subsequently combined to yield the scalar score, effectively quantifying the triple's compatibility within the knowledge graph based on the continuous embeddings represented over the domain X.

3.3 Loss Functions

Loss functions are crucial in the training of knowledge graph embeddings. They guide the learning process by penalizing inaccurate predictions, thus ensuring the fidelity of the embeddings in representing the graph's structure and semantics. Specifically, these functions assess the discrepancy between the predicted scores for actual (positive) triples (h, r, t) and corrupted (negative) triples $(h, r, \hat{t})$.

3.3.1 L2 Norm

In our approach to enhancing the discriminative power of knowledge graph embeddings, we focus on increasing the separation between positive and negative triples. To quantify this separation and provide a clear optimization target, we employ the $L2$ norm as used in TransE [Bor+13], traditionally defined for a vector $\mathbf{x}$ as:

$$L2(\mathbf{x}) = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

Application to Triple Score Discrepancies

For each positive triple and its corresponding set of negative triples, we calculate the $L2$ norm of the difference between their scores, as determined by a scoring function ψ. Specifically, for a positive triple (h, r, t) and a negative triple $(h, r, \hat{t})$, the $L2$ norm is computed as:

$$\text{loss}_{L2} = \sqrt{(1 - \psi(h, r, t) + \psi(h, r, \hat{t}))^2}.$$

In the given formulation, the term "1" serves as a constant that is integrated into the difference calculation between the scores of positive and negative triples. Specifically, this constant is subtracted from the score of the positive triple and added to the score of the negative triple prior to the L2 norm computation. The inclusion of this constant effectively shifts the baseline for the score discrepancies, thereby introducing an inherent bias in the loss calculation. This ensures that the score of a positive triple must not only be greater than that of its corresponding negative triples but also surpass this additional constant hurdle to achieve a lower loss value.

Given a positive triple and multiple corresponding negative triples, we obtain a set of $L2$ losses, one for each negative triple.

Mean L2 Norm Loss for Model Optimization

To consolidate these individual $L2$ losses into a single, coherent loss value for model optimization, we compute their mean:

$$\text{Mean L2 Loss} = \frac{1}{N}\sum_{i=1}^{N} \text{loss}_{L2_i},$$

where N is the number of negative triples associated with each positive triple, and loss_{L2_i} is the $L2$ loss for the i-th negative triple.

This Mean L2 Loss serves as a comprehensive measure of the average Euclidean distance between the scores of positive triples and their associated negative triples. Minimizing this loss during the training process effectively increases the score separation between positive and negative triples, thereby improving the model's ability to distinguish between them and enhancing the overall quality of the embeddings. *Example:* Consider a scenario where a model predicts a score of 0.9 for a positive triple and scores of 0.1, 0.2, 0.3, 0.4, and 0.5 for five corresponding negative triples. The Mean L2 Loss computed across these discrepancies encourages the model to amplify the difference between the positive and negative scores, promoting a clearer distinction in the embedding space and thus facilitating more accurate triple classification.

3.3.2 Margin Ranking Loss Function

The Margin Ranking Loss[3] is specifically designed for ranking tasks, ensuring a predefined margin γ between the scores of correctly ranked items. In the context of knowledge graph embeddings, the loss is defined as:

$$\mathcal{L}_{\text{margin}} = \max(0, \gamma - (\psi(h, r, t) - \psi(h, r, \hat{t})))$$

where γ is the margin, $\psi(h, r, t)$ is the score for a positive triple, and $\psi(h, r, \hat{t})$ is the score for a corresponding negative triple.

Application in Knowledge Graph Embeddings

This loss function ensures that the model is penalized unless the score of the positive triple exceeds the score of the negative triple by at least the margin γ. This encourages the model to not only correctly identify positive triples but also to clearly distinguish them from negative ones by a significant score gap.

Example Consider a scenario with a margin $\gamma = 0.2$, a positive triple score of 0.8, and a negative triple score of 0.7. The computed loss would be:

$$\mathcal{L}_{\text{margin}} = \max(0, 0.2 - (0.8 - 0.7)) = \max(0, 0.1) = 0.1,$$

indicating that the model needs to increase the score gap between the positive and negative triples to reduce the loss.

3.3.3 Binary Cross-Entropy with Logits Loss

This loss combines a Sigmoid layer and the BCELoss [4] in one single class. This version is more numerically stable than using a plain Sigmoid followed by a BCELoss as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability. Binary cross-entropy loss is widely used in binary classification tasks. This loss function is adept at quantifying the discrepancy between two probability distributions: the true distribution and the predicted distribution.

[3]https://pytorch.org/docs/stable/generated/torch.nn.MarginRankingLoss.html
[4]https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html

In the context of knowledge graph embeddings, BCE is employed to categorize triples as either true ((h, r, t)) or false ($(h, r, \hat{t})$).

Sigmoid Function

The sigmoid function is utilized to convert scores into probabilities. It is mathematically defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

BCE Loss Formula

The BCE loss, given predicted probabilities p and actual labels y, is calculated as:

$$\text{BCE}(y, p) = -y \cdot \log(p) - (1 - y) \cdot \log(1 - p).$$

Procedure for Computing BCE Loss

The procedure for calculating BCE loss includes the following steps:

1. Transformation of the scores $\psi(h, r, t)$ and $\psi(h, r, \hat{t})$ into probabilities using the sigmoid function.
2. Assigning appropriate labels: 1 for positive triples (h, r, t) and 0 for negative triples $(h, r, \hat{t})$.
3. Computing the BCE loss individually for each triple, based on its probability and corresponding label.
4. Summation of these individual losses to determine the total loss for the model.

Example

Consider the scenario where a model predicts a score of 2.5 for a positive triple (h, r, t) and -1.5 for a negative triple $(h, r, \hat{t})$. After applying the sigmoid function, the probabilities for these triples are $\sigma(2.5)$ and $\sigma(-1.5)$, respectively. The BCE loss is then computed for both triples, guiding the model to more effectively distinguish between true and false facts within the knowledge graph.

3.4 Model Parameter Optimization

3.4.1 Gradient Descent and Backpropagation

Notations:

- $G = \{(h, r, t)\} \subset E \times R \times E$: Knowledge graph with entities E and relations R.
- $\theta_h, \theta_r, \theta_t \in \mathbb{R}^d$: Model parameters for head, relation, and tail entities, respectively.
- $L(G, \Theta)$: Loss function, where $\Theta = \{\theta_h, \theta_r, \theta_t\}$ represents the set of all model parameters.
- ψ: Scoring function.
- $f(\theta, x)$: Function mapping model parameters and sample points from domain X to embeddings, where θ could be θ_h, θ_r, or θ_t.
- η: Learning rate.
- $\nabla_\Theta L$: Gradient of the loss function with respect to the parameters Θ.

Gradient Descent Algorithm:

1. Initialize model parameters: $\theta_h^{(0)}, \theta_r^{(0)}, \theta_t^{(0)}$.
2. For each iteration i, update each parameter $\theta \in \{\theta_h, \theta_r, \theta_t\}$ as follows:
$$\theta^{(i)} = \theta^{(i-1)} - \eta \nabla_\theta L(G, \Theta^{(i-1)})$$
3. Return optimized parameters: $\Theta^{(n)} = \{\theta_h^{(n)}, \theta_r^{(n)}, \theta_t^{(n)}\}$.

Gradient Computation For the scoring function ψ involving the parameters θ_h, θ_r, and θ_t, consider a trilinear scoring function $\psi_{\text{trilinear}}$ as an example. Its gradient with respect to a specific parameter θ (which could be any of θ_h, θ_r, or θ_t) is given by:

$$\begin{aligned}\frac{\partial \psi_{\text{trilinear}}}{\partial \theta} = \int_a^b \Bigg(& \frac{\partial f(\theta_h, x)}{\partial \theta} \cdot f(\theta_r, x) \cdot f(\theta_t, x) \\ & + f(\theta_h, x) \cdot \frac{\partial f(\theta_r, x)}{\partial \theta} \cdot f(\theta_t, x) \\ & + f(\theta_h, x) \cdot f(\theta_r, x) \cdot \frac{\partial f(\theta_t, x)}{\partial \theta} \Bigg) \, dx \end{aligned} \tag{3.11}$$

This computation highlights how the gradient with respect to each specific parameter θ is calculated, taking into account its contribution to the scoring function $\psi_{\text{trilinear}}$.

Updating Model Parameters Using the chain rule, the update rule for each specific parameter in Θ at iteration i with a learning rate η is given by:

$$\theta^{(i)} = \theta^{(i-1)} - \eta \nabla_{\theta} L(G, \Theta^{(i-1)}) \tag{3.12}$$

This iterative process continues until convergence, optimizing the model parameters $\Theta = \{\theta_h, \theta_r, \theta_t\}$ with respect to the training data.

3.4.2 Gradient Computation of different Scoring Functions

Binary Cross-Entropy Logits Loss Function

The Binary Cross-Entropy Logits Loss for positive and negative samples is defined as:

$$L = -\left[y \cdot \log(\sigma(\psi_+)) + (1-y) \cdot \log(1-\sigma(\psi_-))\right], \tag{3.13}$$

where $y = 1$ for positive samples and $y = 0$ for negative samples. Simplifying the loss function, we get:

$$L = -\left[\log(\sigma(\psi_+)) + \log(1-\sigma(\psi_-))\right]. \tag{3.14}$$

The gradients of the loss function with respect to the scoring functions are computed as:

$$\frac{\partial L}{\partial \psi_+} = -(1-\sigma(\psi_+)), \tag{3.15}$$

$$\frac{\partial L}{\partial \psi_-} = \sigma(\psi_-). \tag{3.16}$$

Chain Rule Formula: The chain rule for computing the gradient of the loss with respect to model parameters is given by:

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial \psi} \cdot \frac{\partial \psi}{\partial \theta}, \tag{3.17}$$

where θ represents the model parameter ($\theta_h, \theta_r, \theta_t,$ or $\theta_{t'}$), and ψ represents the scoring function (ψ_+ or ψ_-).

Trilinear Product Scoring Function: The trilinear product scoring function is defined using the Hadamard product for positive and negative samples as:

$$\psi_+ = \int (f(\theta_h, x) \odot f(\theta_r, x) \odot f(\theta_t, x))\, dx, \tag{3.18}$$

$$\psi_- = \int (f(\theta_h, x) \odot f(\theta_r, x) \odot f(\theta_{t'}, x))\, dx. \tag{3.19}$$

Gradient with respect to head entity parameter θ_h: The gradients of the scores with respect to θ_h are:

$$\frac{\partial \psi_+}{\partial \theta_h} = \int (f(\theta_r, x) \odot f(\theta_t, x))\, dx, \tag{3.20}$$

$$\frac{\partial \psi_-}{\partial \theta_h} = \int (f(\theta_r, x) \odot f(\theta_{t'}, x))\, dx. \tag{3.21}$$

Thus, the gradient of the loss function with respect to θ_h is:

$$\frac{\partial L}{\partial \theta_h} = -(1 - \sigma(\psi_+)) \cdot \int (f(\theta_r, x) \odot f(\theta_t, x))\, dx - \sigma(\psi_-) \cdot \int (f(\theta_r, x) \odot f(\theta_{t'}, x))\, dx. \tag{3.22}$$

Gradient with respect to θ_r: The partial derivatives of the scores with respect to θ_r are:

$$\frac{\partial \psi_+}{\partial \theta_r} = \int (f(\theta_h, x) \odot f(\theta_t, x))\, dx, \tag{3.23}$$

$$\frac{\partial \psi_-}{\partial \theta_r} = \int (f(\theta_h, x) \odot f(\theta_{t'}, x))\, dx. \tag{3.24}$$

The gradient of the loss with respect to θ_r is:

$$\frac{\partial L}{\partial \theta_r} = -(1 - \sigma(\psi_+)) \cdot \int (f(\theta_h, x) \odot f(\theta_t, x))\, dx - \sigma(\psi_-) \cdot \int (f(\theta_h, x) \odot f(\theta_{t'}, x))\, dx. \tag{3.25}$$

Gradient with respect to θ_t and $\theta_{t'}$: The derivatives of the scores with respect to θ_t and $\theta_{t'}$ are:

$$\frac{\partial \psi_+}{\partial \theta_t} = \int (f(\theta_h, x) \odot f(\theta_r, x))\, dx, \tag{3.26}$$

$$\frac{\partial \psi_-}{\partial \theta_{t'}} = \int (f(\theta_h, x) \odot f(\theta_r, x))\, dx. \tag{3.27}$$

The gradients of the loss with respect to θ_t and $\theta_{t'}$ are:

$$\frac{\partial L}{\partial \theta_t} = -(1 - \sigma(\psi_+)) \cdot \int (f(\theta_h, x) \odot f(\theta_r, x))\, dx, \tag{3.28}$$

$$\frac{\partial L}{\partial \theta_{t'}} = -\sigma(\psi_-) \cdot \int (f(\theta_h, x) \odot f(\theta_r, x))\, dx. \tag{3.29}$$

2. Vector Triple Product: The scoring function using the Vector Triple Product is defined as:

$$\psi_+ = -\int f(\theta_h, x)\, dx \cdot \int (f(\theta_t, x) \odot f(\theta_r, x))\, dx \tag{3.30}$$

$$+ \int f(\theta_r, x)\, dx \cdot \int (f(\theta_h, x) \odot f(\theta_t, x))\, dx, \tag{3.31}$$

$$\psi_- = -\int f(\theta_h, x)\, dx \cdot \int (f(\theta_{t'}, x) \odot f(\theta_r, x))\, dx \tag{3.32}$$

$$+ \int f(\theta_r, x)\, dx \cdot \int (f(\theta_h, x) \odot f(\theta_{t'}, x))\, dx. \tag{3.33}$$

2.1 Gradient with respect to θ_h: The derivatives of the scores with respect to θ_h are:

$$\frac{\partial \psi_+}{\partial \theta_h} = -\int (f(\theta_t, x) \odot f(\theta_r, x))\, dx + \int f(\theta_r, x)\, dx \cdot \int f(\theta_t, x)\, dx, \tag{3.34}$$

$$\frac{\partial \psi_-}{\partial \theta_h} = -\int (f(\theta_{t'}, x) \odot f(\theta_r, x))\, dx + \int f(\theta_r, x)\, dx \cdot \int f(\theta_{t'}, x)\, dx. \tag{3.35}$$

Using the chain rule, the gradient of the loss function with respect to θ_h is computed as:

$$\frac{\partial L}{\partial \theta_h} = -(1-\sigma(\psi_+)) \cdot \left(-\int f(\theta_t, x)\, dx \cdot \int f(\theta_r, x)\, dx + \int f(\theta_r, x)\, dx \cdot \int f(\theta_t, x)\, dx\right)$$
$$- \sigma(\psi_-) \cdot \left(-\int f(\theta_{t'}, x)\, dx \cdot \int f(\theta_r, x)\, dx + \int f(\theta_r, x)\, dx \cdot \int f(\theta_{t'}, x)\, dx\right). \tag{3.36}$$

2.2 Gradient with respect to θ_r: The partial derivatives of the scores with respect to θ_r are:

$$\frac{\partial \psi_+}{\partial \theta_r} = -\int f(\theta_h, x)\, dx \cdot \int f(\theta_t, x)\, dx + \int f(\theta_h, x) \odot f(\theta_t, x)\, dx, \tag{3.37}$$

$$\frac{\partial \psi_-}{\partial \theta_r} = -\int f(\theta_h, x)\, dx \cdot \int f(\theta_{t'}, x)\, dx + \int f(\theta_h, x) \odot f(\theta_{t'}, x)\, dx. \tag{3.38}$$

Using the chain rule, the gradient of the loss function with respect to θ_r is:

$$\frac{\partial L}{\partial \theta_r} = -(1-\sigma(\psi_+)) \cdot \left(-\int f(\theta_h, x)\, dx \cdot \int f(\theta_t, x)\, dx + \int f(\theta_h, x) \odot f(\theta_t, x)\, dx\right)$$
$$- \sigma(\psi_-) \cdot \left(-\int f(\theta_h, x)\, dx \cdot \int f(\theta_{t'}, x)\, dx + \int f(\theta_h, x) \odot f(\theta_{t'}, x)\, dx\right). \quad (3.39)$$

2.3 Gradient with respect to θ_t and $\theta_{t'}$: The derivatives of the scores with respect to θ_t and $\theta_{t'}$ are zero, as the scoring function's structure leads to no direct gradient with respect to θ_t and $\theta_{t'}$ in the Vector Triple Product context. Consequently, the gradients of the loss function with respect to θ_t and $\theta_{t'}$ are also zero:

$$\frac{\partial L}{\partial \theta_t} = 0, \quad (3.40)$$
$$\frac{\partial L}{\partial \theta_{t'}} = 0. \quad (3.41)$$

3. Compositional Product: The scoring functions for the positive and negative triples are defined using the compositional product:

$$\psi_+ = \int (f(\theta_r, (f(\theta_h, x)) \odot f(\theta_t, x))\, dx, \quad (3.42)$$
$$\psi_- = \int (f(\theta_r, (f(\theta_h, x)) \odot f(\theta_{t'}, x))\, dx. \quad (3.43)$$

Here, $f(\theta_r, (f(\theta_h, x)))$ represents the relation embedding as a function of the head embedding.

3.1 Gradient with respect to θ_h: The derivatives of the scores with respect to θ_h are:

$$\frac{\partial \psi_+}{\partial \theta_h} = \int \frac{\partial (f(\theta_r, f(\theta_h, x))) \odot f(\theta_t, x))}{\partial \theta_h}\, dx, \quad (3.44)$$
$$\frac{\partial \psi_-}{\partial \theta_h} = \int \frac{\partial (f(\theta_r, f(\theta_h, x))) \odot f(\theta_{t'}, x))}{\partial \theta_h}\, dx. \quad (3.45)$$

The gradient of the loss function with respect to θ_h is then computed as:

$$\frac{\partial L}{\partial \theta_h} = -(1-\sigma(\psi_+)) \cdot \int \frac{\partial (f(\theta_r, f(\theta_h, x))) \odot f(\theta_t, x))}{\partial \theta_h}\, dx \quad (3.46)$$
$$- \sigma(\psi_-) \cdot \int \frac{\partial (f(\theta_r, f(\theta_h, x))) \odot f(\theta_{t'}, x))}{\partial \theta_h}\, dx. \quad (3.47)$$

3.2 Gradient with respect to θ_r: The derivatives of the scores with respect to θ_r are:

$$\frac{\partial \psi_+}{\partial \theta_r} = \int f(\theta_t, x)\, dx, \tag{3.48}$$

$$\frac{\partial \psi_-}{\partial \theta_r} = \int f(\theta_{t'}, x)\, dx. \tag{3.49}$$

The gradient of the loss function with respect to θ_r is then computed as:

$$\frac{\partial L}{\partial \theta_r} = -(1 - \sigma(\psi_+)) \cdot \int f(\theta_t, x)\, dx - \sigma(\psi_-) \cdot \int f(\theta_{t'}, x)\, dx. \tag{3.50}$$

3.3 Gradient with respect to θ_t and $\theta_{t'}$: The derivatives of the scores with respect to θ_t and $\theta_{t'}$ are:

$$\frac{\partial \psi_+}{\partial \theta_t} = \int f(\theta_r, (f(\theta_h, x)))\, dx, \tag{3.51}$$

$$\frac{\partial \psi_-}{\partial \theta_{t'}} = \int f(\theta_r, (f(\theta_h, x)))\, dx. \tag{3.52}$$

The gradients of the loss function with respect to θ_t and $\theta_{t'}$ are then computed as:

$$\frac{\partial L}{\partial \theta_t} = -(1 - \sigma(\psi_+)) \cdot \int f(\theta_r, (f(\theta_h, x)))\, dx, \tag{3.53}$$

$$\frac{\partial L}{\partial \theta_{t'}} = -\sigma(\psi_-) \cdot \int f(\theta_r, (f(\theta_h, x)))\, dx. \tag{3.54}$$

Margin Ranking Loss Function

The Margin ranking loss for the compositional scoring function is given by:

$$L = \max(0, \text{margin} - \psi_+ + \psi_-). \tag{3.55}$$

When margin $- \psi_+ + \psi_- > 0$, the gradients are:

$$\frac{\partial L}{\partial \psi_+} = -1, \tag{3.56}$$

$$\frac{\partial L}{\partial \psi_-} = 1. \tag{3.57}$$

Compositional Scoring Function

The scoring functions for positive and negative triples using the compositional product are:

$$\psi_+ = \int (f(\theta_r, (f(\theta_h, x)) \odot f(\theta_t, x))\, dx, \tag{3.58}$$

$$\psi_- = \int (f(\theta_r, (f(\theta_h, x)) \odot f(\theta_{t'}, x))\, dx. \tag{3.59}$$

Gradient Computation

Gradient with respect to θ_h: Considering the compositional nature of $f(\theta_r, f(\theta_h, x))$, the gradient of the loss function with respect to θ_h is:

$$\frac{\partial L}{\partial \theta_h} = \frac{\partial L}{\partial \psi_+} \cdot \frac{\partial \psi_+}{\partial \theta_h} + \frac{\partial L}{\partial \psi_-} \cdot \frac{\partial \psi_-}{\partial \theta_h} \tag{3.60}$$

$$= -\int \frac{\partial (f(\theta_r, f(\theta_h, x))) \odot f(\theta_t, x))}{\partial \theta_h}\, dx + \int \frac{\partial (f(\theta_r, f(\theta_h, x))) \odot f(\theta_{t'}, x))}{\partial \theta_h}\, dx. \tag{3.61}$$

Gradient with respect to θ_r: The gradient of the loss function with respect to θ_r is computed as:

$$\frac{\partial L}{\partial \theta_r} = \frac{\partial L}{\partial \psi_+} \cdot \frac{\partial \psi_+}{\partial \theta_r} + \frac{\partial L}{\partial \psi_-} \cdot \frac{\partial \psi_-}{\partial \theta_r} \tag{3.62}$$

$$= -\int f(\theta_t, x)\, dx + \int f(\theta_{t'}, x)\, dx. \tag{3.63}$$

Gradient with respect to θ_t: The gradient of the loss function with respect to θ_t is derived by applying the chain rule to the positive score ψ_+:

$$\frac{\partial L}{\partial \theta_t} = \frac{\partial L}{\partial \psi_+} \cdot \frac{\partial \psi_+}{\partial \theta_t}, \tag{3.64}$$

where

$$\frac{\partial \psi_+}{\partial \theta_t} = \int f(\theta_r, f(\theta_h, x))\, dx. \tag{3.65}$$

Hence, the gradient with respect to θ_t is:

$$\frac{\partial L}{\partial \theta_t} = -\int f(\theta_r, f(\theta_h, x))\, dx. \tag{3.66}$$

Since ψ_- does not directly depend on θ_t, its gradient with respect to θ_t is not computed in this context. Similarly, ψ_+ does not directly depend on $\theta_{t'}$, so its gradient with respect to $\theta_{t'}$ is also not computed.

Similarly, the gradient of other loss functions can be computed using the same strategy as explained above.

The computation of gradients in knowledge graph embedding models is vital for understanding how changes in entity embeddings impact the learning process. These gradients guide the update of embeddings during training, especially in distinguishing positive samples from negative ones.

4 Implementation

In the implementation of our knowledge graph embedding, we harnessed a suite of libraries and frameworks to streamline the development process. The backbone of our implementation is PyTorch[1], utilized at its latest stable release, alongside PyTorch Lightning version 1.6.4[2] for an efficient training workflow. Our work was carried out using Python version 3.10.0[3], capitalizing on its rich ecosystem and robustness for scientific computing. The experiments were conducted on a high-performance server boasting an NVIDIA GeForce RTX 3090 GPU with 24GB of memory and a 40-core CPU, ensuring expeditious computation and model prototyping.

The development process was supported by a robust environment that leverages the parallel computation capabilities of modern hardware. We utilized the DICE Embedding Framework[4], which is optimized for large-scale knowledge graph embeddings and capable of exploiting the full potential of multi-core CPUs and GPUs.

In addition to specialized libraries, we also integrated commonly used Python libraries such as NumPy[5] for numerical operations and argparse[6] for parsing command-line options, enabling dynamic configuration and experimentation.

To further refine our model, we employed Neural Network Intelligence (NNI) by Microsoft[7], a toolkit for neural architecture search that allows for automated model design and hyperparameter tuning. NNI provided us with the flexibility to explore a wide range of architectural designs and to optimize our model's performance on the knowledge graph embedding task.

[1]`https://pytorch.org/`
[2]`https://github.com/PyTorchLightning/pytorch-lightning`
[3]`https://www.python.org/downloads/release/python-3100/`
[4]`https://github.com/dice-group/dice-embeddings`
[5]`https://numpy.org/`
[6]`https://docs.python.org/3/library/argparse.html`
[7]`https://github.com/microsoft/nni`

4.1 Preprocessing

The preprocessing step involved generating unique identifiers for each entity and relation in our dataset. Our knowledge graph dataset consists of triples in the form of (head, relation, tail), which we then represent with their corresponding ID mappings.

Dataset Selection For our experiments, we selected a dataset from a range of options such as UMLS, KINSHIP, WN18, YAGO, among others. The choice of the dataset was based on the specific application and the characteristics of the knowledge graph we aimed to model.

Parameters Initialization For initializing the parameters of entities and relations θ, we tried 2 initialization methods, one was nn.init.uniform and the other was the Xavier initialization method, which is a popular choice for deep neural networks. This approach helps maintain a stable variance of activations throughout the layers, which is crucial for the convergence during training. We also explored standard normal initialization as an alternative, provided by PyTorch's neural network module.

```python
def initialize_weights(Parameters):
    nn.init.uniform_(Parameters.weight.data, -0.05, 0.05)
    # OR
    nn.init.xavier_uniform_(Parameters.weight.data)
```

Listing 4.1: Parameters initialization of entities and relations

Negative Sampling To train our model, we employed a negative sampling technique where k negative samples were generated for each positive triple by altering either the head or the tail entity. This was done to maintain the symmetry of relationships within the knowledge graph. For instance, if we take a batch of 32 triples and k is 5, the total number of triples in training would be $32 + 32 \times 5$.

Batch Preparation The batch preparation involved creating x batches containing both positive and negative triple IDs, and corresponding y batches that contain binary values, with 1 representing a positive triple and 0 for a negative triple. These batches were then used to calculate the loss during the training epochs, as shown in the debugging process from our development environment.

```python
def prepare_batches(triples, negative_ratio):
    x_batch = []
    y_batch = []
    for triple in triples:
        x_batch.append(triple_to_id(triple))
        y_batch.append(1)  # Positive sample
        # Generate negative samples
        for _ in range(negative_ratio):
            negative_triple = generate_negative_sample(triple)
            x_batch.append(triple_to_id(negative_triple))
            y_batch.append(0)  # Negative sample
    return x_batch, y_batch
```

Listing 4.2: Sample Python code for batch preparation.

4.2 Embedding Generation and Scoring Functions

Following the batch preparation, the next step in our pipeline was to feed the triples into designated function spaces to generate embeddings as functions. The embeddings for the head entities, relations, and tail entities were formulated as $f(\theta, x)$, where θ represents the learnable parameters and x denotes the discrete points in the input space. We utilized the torch's linspace function to generate these discrete points within the interval $[-1, 1]$. The x dimension was kept at 50 for our experiments.

Function Space Implementation We implemented three different function spaces for generating embeddings as functions: complex number, polynomial, and neural network function spaces. The output of these function spaces conforms to the shape of the input x where x is a sample points vector having values between an interval [a,b] of a finite domain X, rather than directly reflecting the dimensions of the head or relation parameter vector.

```python
x = torch.linspace(-1,1,50)
head_embedding = function_space(headEntity_Parameters , x)
tail_embedding = function_space(tailEntity_Parameters, x)
relation_embedding = function_space(relation_Parameters, x)
r(head_embedding) = function_space(relation_Parameters,
                    head_embedding)
```

Listing 4.3: Dynamic embedding generation from function spaces

Dimensionality Considerations The parameter vector dimension, θ, was carefully chosen. Higher dimensionality of the vector can potentially lead to overfitting, as the model may capture too much detail from the training data, including noise. Conversely, weights with lower dimensionality may result in underfitting, as the model might not capture sufficient features of the data to perform accurately. Thus, a balance was struck to ensure the model had enough capacity to learn the features without overfitting.

Scoring Functions Upon obtaining the embeddings as functions from the function spaces, we applied different scoring functions to generate the scores for each triple. The following code snippet illustrates the computation of scores using trapezoidal integration, which serves as an approximation for the integral over the function space. We introduced various scoring functions such as compositional, vector triple product (VTP), and trilinear, each suitable for different types of relational patterns within the knowledge graph.

```

x = torch.linspace(-1,1,50)
# Compositional scoring function
score_compositional = torch.trapezoid(r(head_embedding) *
tail_embedding, x, dim=2).mean(dim=1)

# VTP scoring function
termA = -torch.trapz(tail_embedding, x, dim=2).mean(dim=1)
termB = torch.trapz(head_embedding * relation_embedding, x, dim=2)
.mean(dim=1)
termC = torch.trapz(relation_embedding, x, dim=2).mean(dim=1)
termD = torch.trapz(tail_embedding * head_embedding, x, dim=2)
.mean(dim=1)
score_vtp = termA * termB + termC * termD

# Trilinear scoring function
score_trilinear = torch.trapezoid(head_embedding *
tail_embedding * relation_embedding, x, dim=2).mean(dim=1)
```

Listing 4.4: Computation of scores using different scoring functions.

This scoring process allows us to capture the interactions between entities and relations within the knowledge graph, facilitating effective embedding learning tailored to the specificities of our dataset.

4.3 Loss Computation and Optimization

Post embedding generation and score calculation, we compute the loss to quantify the error in our predictions. Three types of loss functions were explored:

- **L2 Norm-Based Loss Function** This loss function computes the L2 norm of the difference between the predicted and actual embeddings, promoting embeddings that are close to the ground truth in the Euclidean space.

- **BCE with Logits Loss** The Binary Cross-Entropy (BCE) with logits loss combines a sigmoid layer and the BCE loss in one single class. This is more numerically stable than using a plain Sigmoid followed by a BCE loss, as it takes advantage of the log-sum-exp trick for numerical stability.
- **Margin Ranking Loss Function** This loss function is used for ranking problems and aims to ensure that correctly ranked pairs are separated by a certain margin.

After computing the loss, we perform backpropagation to adjust the model parameters in the direction that minimizes the loss:

```
# Compute loss
loss = compute_loss_function(scores, targets)
# Perform backpropagation
loss.backward()
```

Listing 4.5: Backpropagation using the computed loss.

Optimization For the optimization of our model, we chose the Adam optimizer due to its effectiveness in handling sparse gradients and adaptive learning rate capabilities. The learning rate is a crucial hyperparameter in the training process, influencing the convergence rate and the model's final performance. We experimented with learning rates ranging from 0.1 to 0.001 to find the optimal setting for our model.

```
# Initialize Adam optimizer with weight decay
optimizer = optim.Adam([
    {'params': entity_embeddings.parameters()},
    {'params': relation_embeddings.parameters()}
], lr=0.001, weight_decay=0.005)
```

Listing 4.6: Setting up the Adam optimizer with weight decay.

The choice of learning rate and weight decay was made to balance the trade-off between convergence speed and the risk of overfitting. By incorporating weight decay, we added a regularization term that penalizes large weights, further helping to prevent overfitting.

Evaluation Metrics

To rigorously assess the efficiency and effectiveness of our knowledge graph embedding model, we employ two widely recognized evaluation metrics in the domain of information retrieval and recommendation systems: Mean Reciprocal Rank (MRR) and Hits@N. More on this is written in section 5

4.4 Neural Architecture Search with NNI

In our process to identify the best performing neural network architecture for our knowledge graph embedding model, we employed the Neural Network Intelligence (NNI) [Mic20] framework developed by Microsoft. This framework facilitates an efficient search for the best-performing neural architecture by varying multiple hyperparameters including parameter dimensions, number of layers, activation functions, learning rates, number of epochs, dropout rates, weight decays, and the utilization of a residual architecture.

Managing Experiments with NNI

To initiate and manage our neural architecture search experiments, we used the following NNI command-line interface (CLI) commands:

```
# To create and start an experiment
nnictl create --config config.yml

# To stop all running experiments
nnictl stop --all

# To stop a specific experiment by ID
nnictl stop --id <experiment_id>
```

Example Search Space

An example of the search space defined for our experiments is as follows:

```
{
  "num_layers": {"_type": "choice", "_value": [2, 3, 4, 5, 6]},
  "activation_function": {"_type": "choice", "_value":
  ["tanh", "relu", "sigmoid"]},
  "num_epochs": {"_type": "choice", "_value": [20, 50, 100]},
  "batch_size": {"_type": "choice", "_value": [256, 512, 1024]},
  "weight_decay": {"_type": "uniform", "_value": [0.0001, 0.1]},
  "dropout_rate": {"_type": "uniform", "_value": [0.1, 1]},
  "learning_rate": {"_type": "uniform", "_value": [0.0001, 0.01]},
  "embedding_dim": {"_type": "choice", "_value": [32, 64, 128,
  256, 512, 1024, 2048]}
}

// to Retrieve the parameters
---- example.py
```

```
import nni

# Fetching hyperparameters
params = nni.get_next_parameter()
activation_function = params["activation_function"]
```

Listing 4.7: Example Search Space

Configuration File

The configuration for our experiments, defined in a YAML file, specifies the experimental setup including the search algorithm and resource allocation:

```
experimentName: nas_for_knowledge_graph
trialConcurrency: 1
maxExperimentDuration: 10h
maxTrialNumber: 36
searchSpaceFile: search_space.json
useAnnotation: false
trialCommand: python main.py
trialCodeDirectory: .
trialGpuNumber: 0
tuner:
  name: TPE
  classArgs:
    optimize_mode: maximize
trainingService:
  platform: local
  use_active_gpu: True
```

Listing 4.8: Configuration File Example

Key Configuration Aspects:

- **Trial Concurrency** Defines the number of trials to run concurrently, which is set to 1 to ensure that each trial has dedicated resources.
- **Use of Active GPU** Specifies whether to actively use the GPU for training, enhancing computational efficiency.
- **Search Algorithm** The Tree-structured Parzen Estimator (TPE) tuner is employed with the goal to *maximize* the Mean Reciprocal Rank, indicating our objective to improve the model's ranking performance.

This section describes our methodical approach to neural architecture search, leveraging the NNI framework to systematically explore a vast space of architectural configurations, thereby identifying the most effective structure for our knowledge graph embedding model.

5 Evaluation

5.1 Setup and datasets

5.1.1 Evaluation Setup

Dice Embedding Framework [DN22] All the experiments were run using the dice embedding framework. Dice embedding framework is based on the frameworks DASK [1], Pytorch Lightning [2] and Hugging Face [3] to compute embeddings for large-scale knowledge graphs in a hardware-agnostic manner, which can address real-world challenges pertaining to the scale of real application. They provide an open-source version of Git Hub. [4]

Configuration Our experiments were conducted on the OPAL server at the University of Paderborn, which is equipped with 40 cores and an NVIDIA GeForce RTX 3090 GPU with 24 GB of memory. This robust computational environment enabled us to utilize the Adam optimizer, renowned for its effectiveness as it combines the strengths of Adagrad and Rmsprop. The learning rate was set at a relatively high 0.01, determined through hyperparameter optimization facilitated by NNI [Mic21].

For our scoring mechanism, we opted for the Negative Sampling technique, maintaining a ratio of five negative triples for each positive triple. This was a strategic decision to manage the computational load, as the higher embedding dimension significantly increased the number of parameters, which could not be efficiently processed using the K vs All approach given the GPU constraints.

Moreover, we settled on a batch size of 1024, which struck a balance between computational efficiency and the ability to capture sufficient data variations during training.

[1] `https://docs.dask.org/en/stable/`
[2] `https://lightning.ai/docs/pytorch/stable/`
[3] `https://huggingface.co/docs`
[4] `https://github.com/dice-group/dice-embeddings`

5.1.2 Datasets

Dataset	E	R	G^{Train}	$G^{\text{Val.}}$	G^{Test}
KINSHIP	104	25	8,544	1,068	1,074
UMLS	135	46	5,216	652	661
FB15K	14,951	1,345	483,142	50,000	59,071
WN18	40,943	18	141,442	5,000	5,000
NELL-995-h100	22,411	43	50,314	3,763	3,746
NELL-995-h50	34,667	86	72,767	5,440	5,393
NELL-995-h25	70,145	172	122,618	9,194	9,187

Tab. 5.1: Statistics of the datasets used in the experiments, ordered by increasing number of entities.

YAGO3-10 The YAGO3-10 dataset is a well-known benchmark dataset for knowledge base completion. It is a subset of the larger YAGO3 dataset, which is itself an extension of the original YAGO [SKW07; MBS14] knowledge base. YAGO3-10 is distinct because it contains only those entities that are associated with at least ten different relations, making it particularly suitable for evaluating the performance of knowledge graph embedding algorithms. The dataset comprises 123,182 entities and 37 relations, with a total of 1,179,040 triples, most of which describe attributes of persons such as citizenship, gender, and profession.

FB15K The FB15k dataset is a collection of knowledge base relation triples and textual mentions from Freebase. It includes 14,951 entities and 1,345 relationships, totalling 592,213 triples. The dataset serves as a benchmark for various tasks such as link prediction and knowledge graph completion. However, FB15k-237 was introduced [TC15] as a variant to address the issue of test set leakage due to the presence of inverse relations in FB15k. FB15k-237 contains 310,116 triples, 14,541 entities, and 237 relation types, ensuring a cleaner separation between training and testing datasets for more reliable evaluation metrics.

WN18 WN18 is another dataset commonly used in knowledge graph embedding tasks. It is a subset of WordNet, a lexical database of English, designed for evaluating link prediction models. The original WN18 dataset includes 18 relations and 40,943 entities, amounting to a total of 141,442 triples. However, similar to FB15k, WN18 has its variant, WN18RR, created to remove test leakage issues, making it a more challenging dataset for link prediction.

KINSHIP [Det+18] [Hin90] The Kinship dataset is a smaller dataset used for evaluating knowledge graph embeddings. It contains information about family relationships between different entities, providing a different kind of relational structure for embedding models to learn from.

UMLS [Det+18] UMLS, or the Unified Medical Language System, provides a dataset that includes semantic relations within the biomedical domain. It's a compact dataset used for tasks that require understanding complex medical terminologies and their interrelations.

5.2 Evaluation Metrics

Mean Reciprocal Rank (MRR) The Mean Reciprocal Rank is a statistic measure for evaluating any process that produces a list of possible responses to a sample of queries, ordered by the probability of correctness. The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer:

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i},$$

where $|Q|$ is the number of queries and rank_i is the rank position of the first correct triple for the i-th query. MRR provides a measure of the model's ability to rank the correct triple highly.

Hits@N Hits@N measures the proportion of correct answers (positive triples) found in the top N ranks of the model's predictions. It is defined as:

$$\text{Hits@N} = \frac{\text{Number of hits within top N ranks}}{\text{Total number of queries}}.$$

We specifically evaluate Hits@1, Hits@3, and Hits@10 to capture the model's precision at different levels of the ranking list, providing insights into the model's top-ranked predictions' accuracy.

Relevance to Model Evaluation These metrics are particularly suited for knowledge graph embedding models as they directly quantify the model's capability to not only identify valid triples but also to appropriately rank them against invalid ones. High scores in MRR and Hits@N indicate that the model is effective at both recognizing relevant triples and prioritizing them over less relevant or incorrect ones, which is crucial for tasks such as link prediction and entity resolution in knowledge graphs.

Rank-based metrics are crucial for assessing the performance of link prediction models in knowledge graphs. We discuss several desiderata that such metrics should ideally satisfy:

- **Non-negativity** All metric scores should be non-negative.
- **Fixed optimum** The best possible score (optimum) should be a fixed value, ideally 1, to indicate perfect prediction.
- **Asymptotic pessimum** Worse ranks should approach a fixed worst-case score (pessimum), typically 0 or -1.
- **Anti-monotonicity** As the rank of the true triple improves (decreases), the metric should increase, reflecting better performance.
- **Size invariance** The metric should not be influenced by the number of candidate triples, allowing for comparability across different datasets.

Property	**MR**	**MRR**	(H_k)
Non-negativity	✓	✓	✓
Fixed optimum	×	✓	✓
Asymp. pessimum	×	✓	✓
Anti-monotonic	×	✓	×
Size invariant	×	×	×

Tab. 5.2: Summarizes how the Mean Rank (MR), Mean Reciprocal Rank (MRR), and Hits@k (H_k) satisfy these requirements. [Hoy+22]

5.3 Results

5.3.1 Impact of different Scoring Functions ψ and Loss Function L

This section focuses on the impact of different scoring functions and loss functions on baseline function spaces — Polynomial and Complex Numbers — as well as Neural Network Function Space (NNFS). Through the experiments conducted, we observed distinct variations in model performance.

Table number 5.3 and 5.4 shows the evaluation results of the various scoring function when applied with different loss functions for polynomial, complex number and neural network function space.

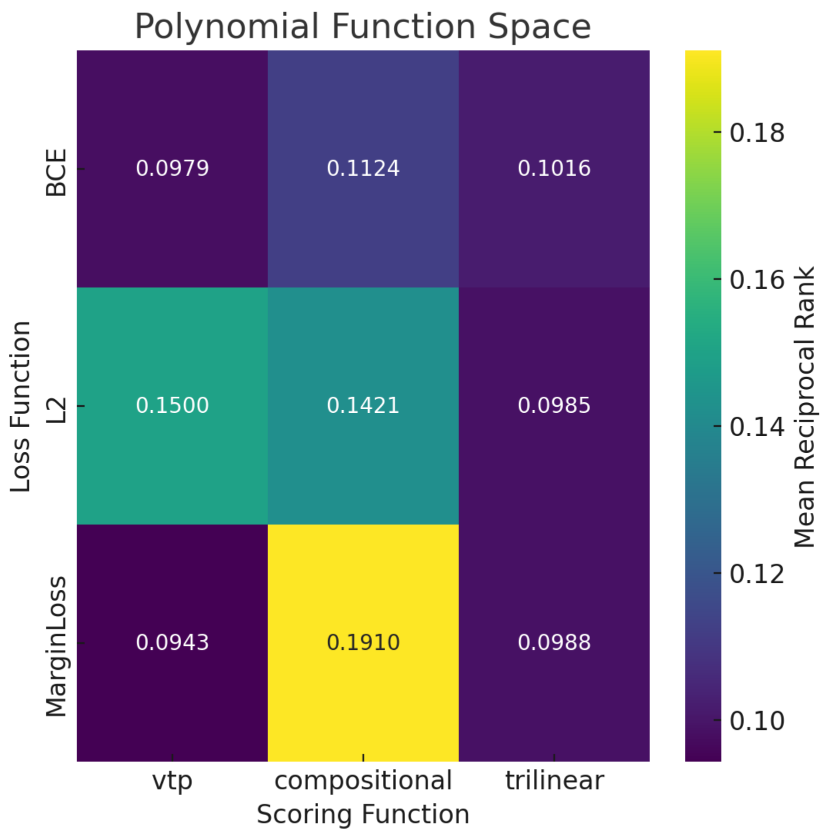


Fig. 5.1: Heatmap depicting the Mean Reciprocal Rank (MRR) performance of Polynomial function space The x-axis represents various scoring functions, and the y-axis corresponds to different loss functions. The MRR data is averaged from the UMLS and KINSHIP datasets to evaluate the most effective scoring and loss function combination within each function space. These results are based on 64 parameters for an embedding function and a negative-to-positive triple ratio (k) of 5.

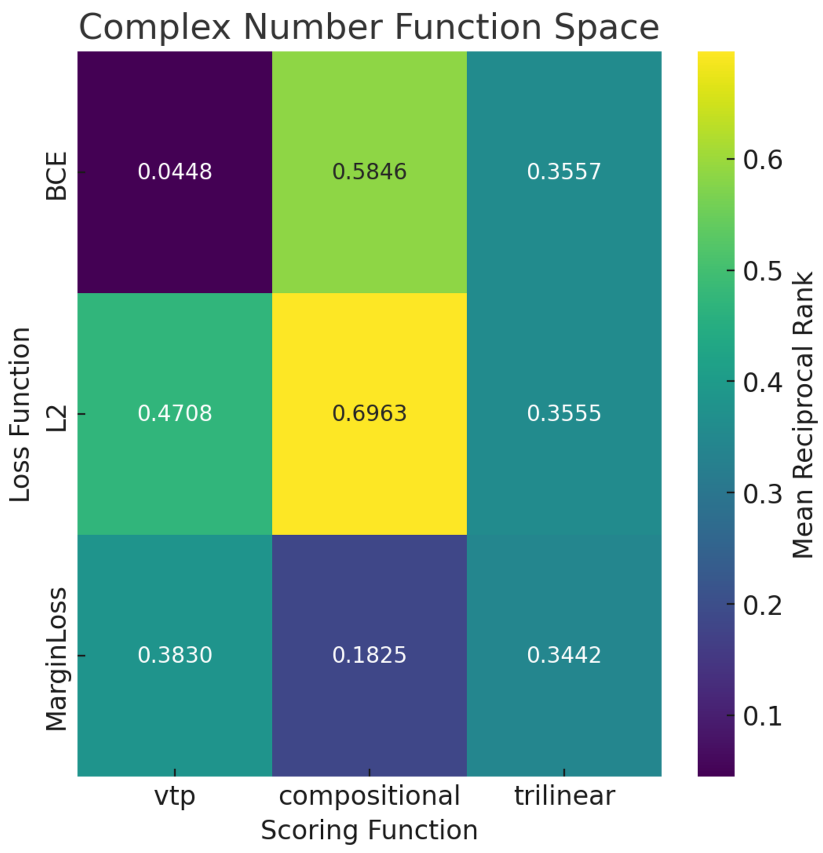


Fig. 5.2: Heatmap depicting the Mean Reciprocal Rank (MRR) performance of Complex number function space. The x-axis represents various scoring functions, and the y-axis corresponds to different loss functions. The MRR data is averaged from the UMLS and KINSHIP datasets to evaluate the most effective scoring and loss function combination within each function space. These results are based on 64 parameters for an embedding function and a negative-to-positive triple ratio (k) of 5.

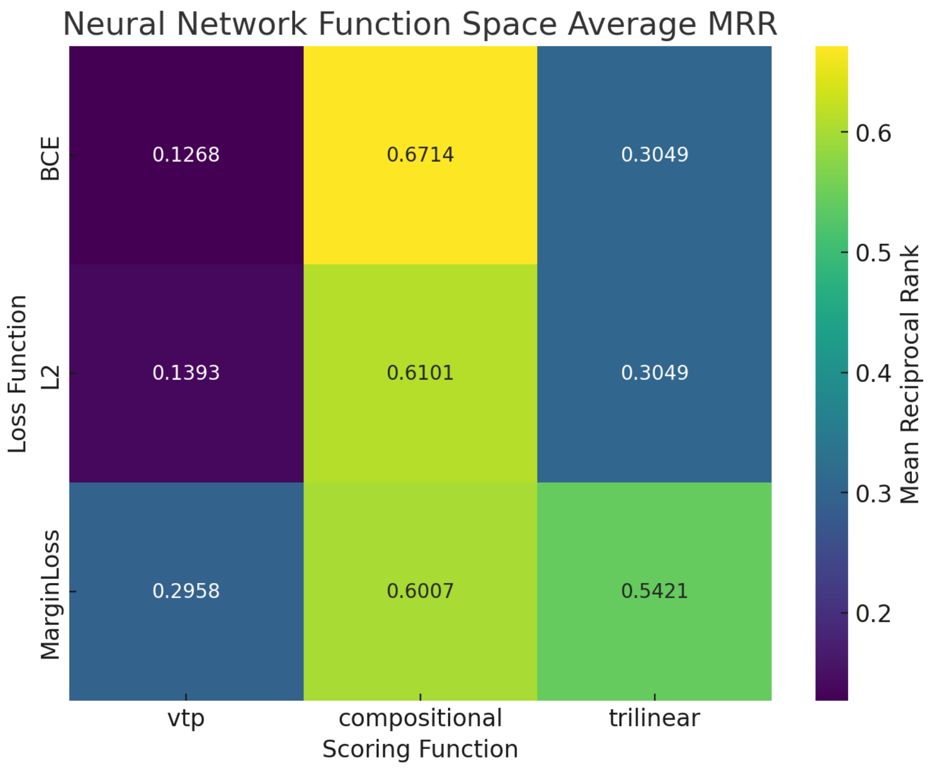


Fig. 5.3: Heatmap depicting the Mean Reciprocal Rank (MRR) performance of Neural network function space Function Space. The x-axis represents various scoring functions, and the y-axis corresponds to different loss functions. The MRR data is averaged from the UMLS and KINSHIP datasets to evaluate the most effective scoring and loss function combination within each function space. These results are based on 64 parameters for an embedding function and a negative-to-positive triple ratio (k) of 5.

Function Space	Loss Function	Scoring Function		
		vtp	compositional	trilinear
Polynomial FS	BCE	0.101587	0.124242	0.104929
Polynomial FS	L2	0.094438	0.182684	0.098315
Polynomial FS	MarginLoss	0.094219	**0.287853**	0.098767
Complex Number FS	BCE	0.028664	**0.801517**	0.057912
Complex Number FS	L2	0.472746	0.801505	0.058051
Complex Number FS	MarginLoss	0.293277	0.182424	0.053343
NNFS	BCE	0.126792	**0.765162**	0.055151
NNFS	L2	0.139285	0.762693	0.055576
NNFS	MarginLoss	0.467276	0.743818	0.554658

Tab. 5.3: MRR of different scoring functions on UMLS test dataset with 64 parameters for an embedding function and neg ratio of 5. Bold results indicate the best results.

Function Space	Loss Function	Scoring Function		
		vtp	compositional	trilinear
Polynomial FS	BCE	0.094219	0.100474	0.098315
Polynomial FS	L2	**0.205584**	0.101587	0.098767
Polynomial FS	MarginLoss	0.094438	0.094219	0.098767
Complex Number FS	BCE	0.060899	0.367597	0.653471
Complex Number FS	L2	0.468854	0.591046	**0.652906**
Complex Number FS	MarginLoss	0.472746	0.182684	0.635044
NNFS	BCE	0.126792	**0.577559**	0.554658
NNFS	L2	0.139285	0.457680	0.554198
NNFS	MarginLoss	0.124243	0.457623	0.529497

Tab. 5.4: MRR of different scoring functions on the KINSHIP test dataset with 64 parameters for an embedding function and neg ratio = 5. Bold results indicate the best results.

The compositional scoring function outweighed others, reflecting its capacity to model intricate relationships within the graph. Its formulation as an integral of the compositional relation embedding function, $\int f(\theta_r, f(\theta_h, x)) \cdot f(\theta_t, x)\, dx$, where θ_h, θ_r, and θ_t are the parameters for the head, relation, and tail embeddings respectively, allows for nuanced interactions to be captured effectively as the change in head entity weights would reflect on composite relation embedding function which would help the scoring function to learn the complex relationship between the entities and relations.

In contrast, the Vector Triple Product (VTP) scoring function underperformed, with gradient analyses indicating that variations in the tail entity did not significantly alter the scoring outcome as $\nabla_t L$ **and** $\nabla_{t'} L$ **is 0** as computed here **??**, leading to a potential deficiency in capturing relational dynamics. Meanwhile, the Trilinear

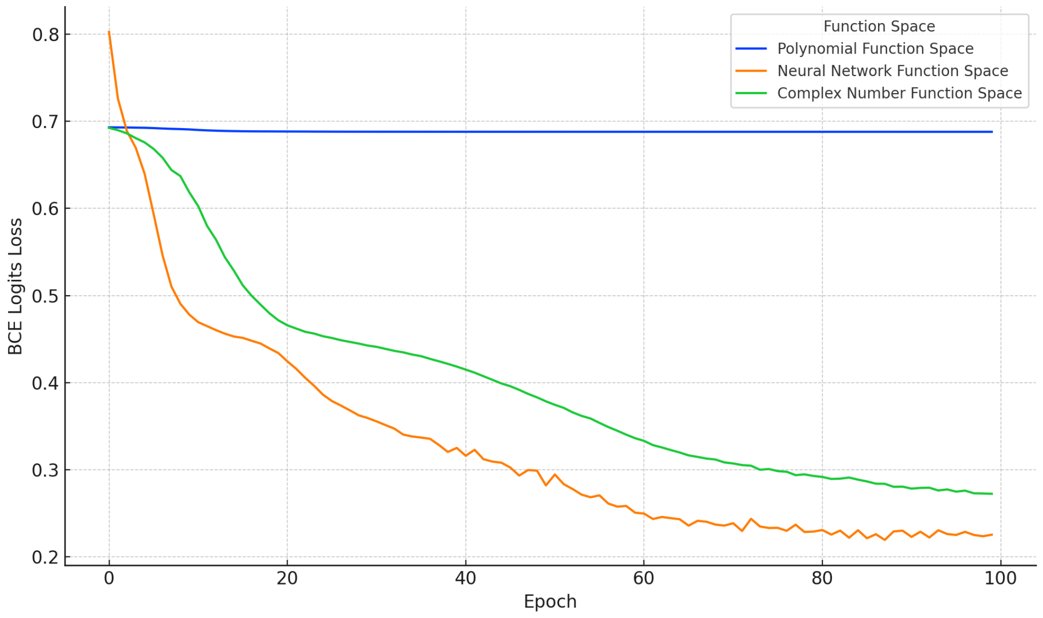


Fig. 5.4: BCE Logits Loss convergence across different function spaces with compositional scoring function ψ.

scoring function provided a middle ground, integrating head, relation, and tail embedding functions into its scoring mechanism, yielding moderate performance.

The heatmaps in figures 5.1, 5.2 and 5.3 delineate the Mean Reciprocal Rank (MRR) performance across different function spaces. Notably, the Complex Number Function Space exhibits superior performance when paired with the L2 loss function and compositional scoring function, suggesting an affinity for capturing complex patterns within smaller datasets such as Kinship and UMLS. The Polynomial Function Space, while versatile, shows a preference for the BCE logits loss with the trilinear scoring function, indicating its strength in linear separability. NNFS demonstrates robustness across various scoring functions, particularly with compositional scoring, likely due to its ability to encapsulate non-linearities inherent in the data.

Regarding the loss convergence graphs, in figure 5.4, 5.5, 5.6, it is evident that both the NNFS(Neural network function space) and Complex number function space exhibit a significant drop in loss during training, indicating effective optimization. In contrast, the Polynomial function space does not optimize as substantially, which may be attributed to its inherent limitations in modelling the non-linear complexities of the datasets in question. This disparity in optimization rates underscores the necessity of selecting an appropriate function space that aligns with the characteristics of the data and the goals of the experiment.

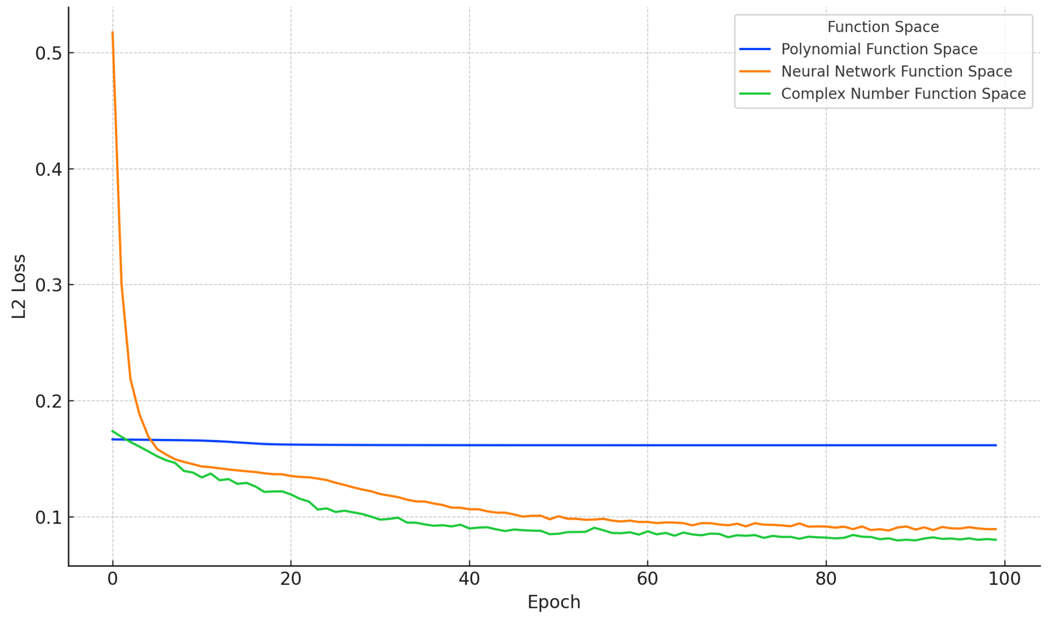


Fig. 5.5: L2 Loss convergence across different function spaces with compositional scoring function ψ.

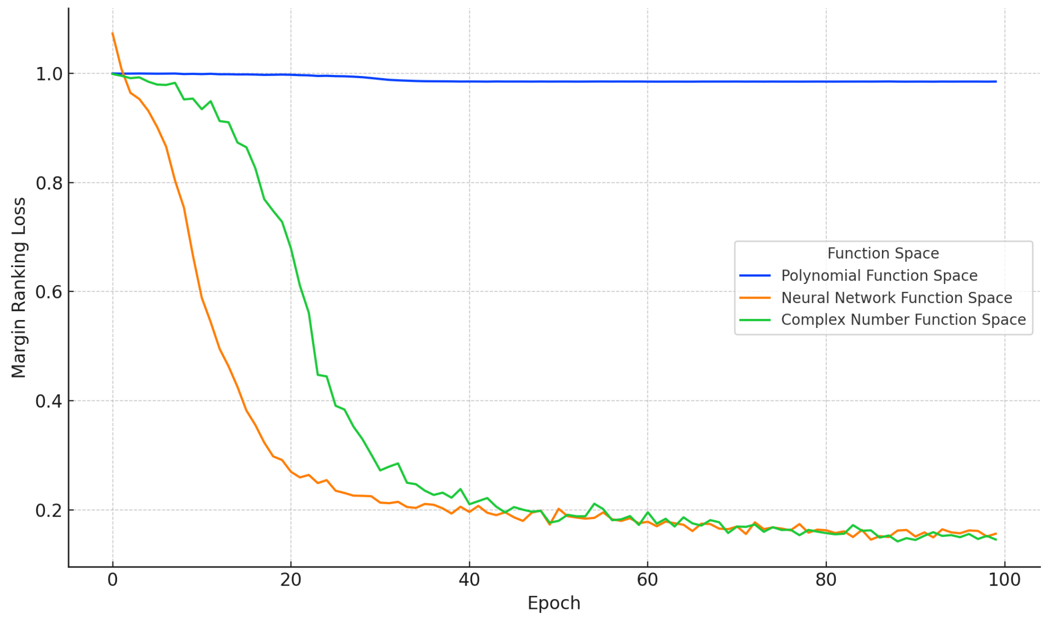


Fig. 5.6: Margin Ranking Loss convergence across different function spaces with compositional scoring function ψ.

5.3.2 Comparison with Other Models

In our comparative evaluation, we compare our Neural Network Function Space (NNFS) and the Complex Number Function Space with other state-of-the-art knowledge graph embedding models such as TransE [Bor+13], DistMult [Yan+14], ComplEx [Tro+16], QMult, OMult [Dem+21], and Keci [DN23]. We have excluded the Polynomial Function Space from this comparison due to its relative underperformance. Uniformity in our analysis is ensured by employing the L2 norm loss function and BCE logits loss function across all models and maintaining consistency in other configurations. The selection of the Compositional scoring function for our function spaces was affirmed by its superior performance, as delineated in Section 5.3.2.

KINSHIP

Loss Function	Model	Hits@1	Hits@3	Hits@10	MRR
BCE	TransE [Bor+13]	0.003	0.027	0.102	0.051
	DistMult [Yan+14]	0.267	0.504	0.846	0.437
	ComplEx [Tro+16]	0.443	0.716	0.933	0.608
	QMult [Dem+21]	0.399	0.638	0.870	0.553
	OMult [Dem+21]	0.396	0.638	0.877	0.551
	Keci [DN23]	**0.508**	**0.781**	**0.958**	**0.667**
	Complex Number FS	0.189	0.425	0.813	0.367
	NNFS	0.438	0.686	0.939	0.578
L2	TransE [Bor+13]	0.000	0.010	0.066	0.036
	DistMult [Yan+14]	0.278	0.524	0.852	0.450
	ComplEx [Tro+16]	0.414	0.666	0.907	0.574
	QMult [Dem+21]	0.463	0.712	0.915	0.613
	OMult [Dem+21]	0.453	0.693	0.893	0.602
	Keci [DN23]	0.459	0.732	0.948	0.623
	ComplexNumber FS	0.422	0.699	0.943	0.591
	NNFS	0.275	0.538	0.897	0.458

Tab. 5.5: Evaluation metrics for different KGE models on the KINSHIP dataset with 64 parameters for embedding function using BCE and L2 loss functions. Bold results indicate the best results.

In the evaluation of knowledge graph embedding models on the KINSHIP dataset, characterized by 104 entities and 25 relations, the performance metrics at two different embedding dimensions reveal insightful trends. The dataset, with its 8,544 training triples, 1,068 validation triples, and 1,074 test triples, serves as a fertile ground for assessing the effectiveness of various embedding strategies.

Model	Hits@1	Hits@3	Hits@10	MRR
TransE [Bor+13]	0.006	0.029	0.110	0.054
DistMult [Yan+14]	0.154	0.230	0.359	0.230
ComplEx [Tro+16]	0.031	0.085	0.220	0.100
QMult [Dem+21]	0.014	0.048	0.143	0.067
OMult [Dem+21]	0.015	0.045	0.108	0.060
Keci [DN23]	0.129	0.228	0.384	0.219
Complex Number FS	0.344	0.628	0.895	0.524
NNFS	**0.619**	**0.843**	**0.963**	**0.745**

Tab. 5.6: Evaluation metrics for different knowledge graph embedding models on KINSHIP Data set with 2048 parameters for embedding function and BCE logits loss function. Bold results indicate the best results.

With an embedding dimension of 64, Keci emerges as the leading model under both BCE and L2 loss functions, showcasing its strength in accurately modelling relationships within the dataset. Specifically, with a BCE loss function, Keci achieves an MRR of 0.667, surpassing other models in the 64-dimensional space. Notably, under the L2 loss function, Keci maintains its top-tier performance with an MRR of 0.623, further solidifying its efficacy. Complex Number FS and Neural Network Function Spaces (NNFS) also delivered strong results. In the aforementioned setting, Complex Number FS reports an MRR of 0.591 using the L2 loss function, and NNFS shows a commendable MRR of 0.578 with the BCE loss function. As the number of parameters for the embedding function is scaled up to 2048, a significant observation is made: while other models demonstrate signs of overfitting—evidenced by a decline in performance compared to the previous results— ComplexNumber and NN function spaces do not exhibit this trend. Instead, they maintain or improve their performance, as seen with NNFS MRR of 0.745 and Complex number function space's MRR of 0.5236 using the BCE loss function.

Nell-995-h50

Analyzing the performance of various knowledge graph embedding models on the NELL-995-h50 dataset—which boasts 34,667 entities, 86 relations, 72,767 training triples, 5,440 validation triples, and 5,393 test triples—yields a comprehensive view of model efficacy across different embedding dimensions and loss functions.

In the initial settings, the Complex Number Function Space (FS) model stands out particularly when paired with the L2 loss function, achieving an MRR of 0.273. This impressive performance can be attributed to the model's utilization of phase information intrinsic to complex numbers, which aligns well with the L2 loss function.

Loss Function	Model	Hits@1	Hits@3	Hits@10	MRR
BCE	TransE [Bor+13]	0.052	0.139	0.258	0.122
	DistMult [Yan+14]	0.067	0.135	0.236	0.122
	ComplEx [Tro+16]	0.124	0.201	0.314	0.186
	QMult [Dem+21]	0.034	0.074	0.153	0.073
	OMult [Dem+21]	0.052	0.105	0.195	0.099
	Keci [DN23]	0.078	0.145	0.241	0.132
	ComplexNumber FS	0.067	0.129	0.222	0.119
	NNFS	0.038	0.089	0.189	0.088
L2	TransE [Bor+13]	0.001	0.014	0.110	0.032
	DistMult [Yan+14]	0.063	0.123	0.229	0.116
	ComplEx [Tro+16]	0.115	0.187	0.294	0.173
	QMult [Dem+21]	0.066	0.132	0.237	0.122
	OMult [Dem+21]	0.028	0.069	0.152	0.070
	Keci [DN23]	0.101	0.174	0.277	0.158
	ComplexNumber FS	**0.207**	**0.300**	**0.406**	**0.273**
	NNFS	0.024	0.049	0.096	0.052

Tab. 5.7: Evaluation metrics for different KGE models on the NELL-995-h50 dataset with 64 parameters for embedding function, using BCE and L2 loss functions. Bold results indicate the best results.

Loss Function	Model	Hits@1	Hits@3	Hits@10	MRR
BCE	TransE [Bor+13]	0.045	0.122	0.236	0.109
	DistMult [Yan+14]	0.068	0.128	0.207	0.115
	ComplEx [Tro+16]	0.034	0.069	0.129	0.066
	QMult [Dem+21]	0.013	0.025	0.054	0.028
	OMult [Dem+21]	0.020	0.034	0.059	0.035
	Keci [DN23]	0.045	0.089	0.167	0.085
	ComplexNumber FS	**0.095**	**0.200**	**0.329**	**0.172**
	NNFS	0.052	0.109	0.187	0.099
L2	TransE [Bor+13]	0.001	0.001	0.001	0.001
	DistMult [Yan+14]	0.005	0.009	0.019	0.010
	ComplEx [Tro+16]	0.001	0.002	0.005	0.003
	QMult [Dem+21]	0.000	0.000	0.002	0.002
	OMult [Dem+21]	0.000	0.004	0.008	0.005
	Keci [DN23]	0.010	0.029	0.079	0.032
	ComplexNumber FS	0.093	0.158	0.253	0.146
	NNFS	0.081	0.133	0.218	0.127

Tab. 5.8: Evaluation metrics for different KGE models on the NELL-995-h50 dataset with 2048 parameters for an embedding function, using BCE and L2 loss functions. Bold results indicate the best results.

This loss function complements the model by effectively capturing the geometric relationships in the embedding space, a crucial advantage when representing complex relational patterns.

While increasing the number of function parameters to 2048, we observe that the Complex Number FS model maintains its strong performance, with an MRR of 0.146 under the L2 loss function and 0.172 under the BCE logits loss function. This suggests that the model scales well with an increased number of parameters and continues to leverage its domain-specific advantages without succumbing to overfitting, a common concern with larger embedding spaces.

In both settings, NNFS shows an increase in performance achieving the MRR of 0.127 while working with the L2 loss function. Results show that both function spaces work better with the L2 norm loss function as compared to the BCE logits loss function at varying embedding dimensions.

The results collectively highlight that while most models had a significantly better result at lower embedding dimensions, NNFS achieved better scores when the number of function parameters was increased showing its adaptive nature and not overfitting to the training dataset.

Nell–995-h100 Dataset

The NELL-995-h100 dataset provides a unique benchmark for assessing the efficacy of various knowledge graph embedding models. With a higher number of parameters, as reflected in Table 5.9, the Complex Number Function Space model dramatically outperforms the others under the L2 loss function, with a significant jump to an MRR of 0.248. The NNFS model, while showing modest performance at this large parameter setting, does not mirror the same level at the lower one, suggesting its potential limitations in exploiting function spaces in the case of this version of NELL.

This analysis reflects the complexity inherent in choosing an appropriate KGE model, where factors such as the loss function and embedding size can have profound impacts on performance. It is worth noting that DistMult stands out across both loss functions.

Other models, such as DistMult and ComplEx, show varied performance with DistMult reaching a Hits@10 of 0.193 and an MRR of 0.105, outperforming ComplEx which attains a Hits@10 of 0.100 and an MRR of 0.044. The TransE model exhibits a balanced performance with a Hits@10 of 0.227 and an MRR of 0.100. These

Loss Function	Model	Hits@1	Hits@3	Hits@10	MRR
BCE	TransE [Bor+13]	0.036	0.113	0.227	0.100
	DistMult [Yan+14]	0.062	0.111	0.193	0.105
	ComplEx [Tro+16]	0.017	0.041	0.100	0.044
	QMult [Dem+21]	0.005	0.015	0.052	0.020
	OMult [Dem+21]	0.007	0.013	0.037	0.019
	Keci [DN23]	0.021	0.056	0.123	0.055
	Complex Number FS	0.085	0.181	0.304	0.158
	NNFS	0.073	0.139	0.247	0.130
L2	TransE [Bor+13]	0.000	0.001	0.002	0.002
	DistMult [Yan+14]	0.004	0.007	0.012	0.011
	ComplEx [Tro+16]	0.001	0.002	0.003	0.003
	QMult [Dem+21]	0.000	0.000	0.000	0.000
	OMult [Dem+21]	0.000	0.000	0.000	0.000
	Keci [DN23]	0.012	0.034	0.087	0.037
	ComplexNumber FS	**0.184**	**0.271**	**0.380**	**0.248**
	NNFS	0.086	0.147	0.247	0.138

Tab. 5.9: Evaluation metrics for different KGE models on the NELL-995-h100 dataset with 2048 parameters for an embedding function, using both BCE and L2 loss functions. Bold results indicate the best results.

results reflect the strengths and limitations of each model, with NNFS demonstrating a consistent ability to navigate the complex structure of the Nell-h100 dataset effectively, likely due to its sophisticated embedding mechanism that leverages the higher-dimensional space without overfitting on the training data. **WN18 Dataset** Upon evaluating the WN18 dataset, which consists of a substantial number of training triples 40,943 and a diverse set of entities and relations, a detailed analysis of the performance of different knowledge graph embedding models reveals nuanced insights.

As the size of the dataset increases, the gap between the NNFS model and the top-performing models like ComplEx and DistMult widens. For instance, the ComplEx model surpasses NNFS with an MRR of 0.893 and a notable H@10 score of 0.924. DistMult also displays remarkable scores, particularly with H@10 of 0.932, suggesting that these models may be more adept at capturing the intricate relationships within larger datasets.

The observed trends highlight that while NNFS is a potent model, especially for smaller or moderately-sized datasets, its relative advantage diminishes in comparison to other sophisticated models as the dataset size and complexity grow.

Loss Function	Model	Hits@1	Hits@3	Hits@10	MRR
BCE	TransE [Bor+13]	0.155	0.443	0.728	0.343
	DistMult [Yan+14]	0.697	0.894	0.934	0.798
	ComplEx [Tro+16]	0.798	0.907	**0.935**	0.855
	QMult [Dem+21]	0.557	0.696	0.777	0.638
	OMult [Dem+21]	0.815	**0.909**	0.931	0.864
	Keci [DN23]	0.636	0.870	0.929	0.756
	ComplexNumber FS	0.017	0.094	0.266	0.090
	NNFS	0.037	0.084	0.178	0.087
L2	TransE [Bor+13]	0.003	0.401	0.761	0.251
	DistMult [Yan+14]	0.658	0.875	0.928	0.771
	ComplEx [Tro+16]	**0.841**	0.908	0.931	**0.877**
	QMult [Dem+21]	0.639	0.802	0.889	0.731
	OMult [Dem+21]	0.205	0.342	0.493	0.302
	Keci [DN23]	0.575	0.835	0.913	0.712
	ComplexNumber FS	0.061	0.100	0.162	0.095
	NNFS	0.081	0.147	0.261	0.141

Tab. 5.10: Evaluation metrics for different KGE models on the WN18 dataset with 64 parameters for embedding function using BCE and L2 loss functions. Bold results indicate the best results.

Loss Function	Model	Hits@1	Hits@3	Hits@10	MRR
L2	TransE [Bor+13]	0.000	0.017	0.040	0.015
	DistMult [Yan+14]	0.073	0.127	0.194	0.116
	ComplEx [Tro+16]	0.024	0.044	0.074	0.042
	QMult [Dem+21]	0.000	0.000	0.000	0.000
	OMult [Dem+21]	0.001	0.002	0.006	0.003
	Keci [DN23]	0.002	0.004	0.009	0.005
	ComplexNumber FS	0.574	0.894	**0.936**	0.735
	NNFS	0.705	0.776	0.821	0.748
BCE	TransE [Bor+13]	0.087	0.350	0.582	0.258
	DistMult [Yan+14]	0.730	**0.909**	0.932	0.821
	ComplEx [Tro+16]	**0.874**	0.908	0.924	**0.893**
	QMult [Dem+21]	0.437	0.502	0.560	0.481
	OMult [Dem+21]	0.586	0.662	0.734	0.637
	Keci [DN23]	0.052	0.078	0.119	0.075
	ComplexNumber FS	0.019	0.232	0.714	0.197
	NNFS	0.733	0.852	0.905	0.799

Tab. 5.11: Evaluation metrics for different KGE models on the WN18 dataset with 2048 parameters for an embedding function, using L2 and BCE loss functions. Bold results indicate the best results.

The comparative study reveals that the Neural Network Function Space (NNFS) and Complex Number Function Space models exhibit varying performance metrics depending on the loss function and number of function parameters. The NNFS model, while demonstrating high efficiency with the L2 loss function at a higher number of embedding function parameters, shows a significant performance drop with a reduced number of parameters, indicating sensitivity. Conversely, the Complex Number FS model performs inconsistently across loss functions. In comparison, models such as ComplEx and DistMult display robustness again embedding dimension, maintaining stable performance across diverse conditions, highlighting their potential for generalizability. This study underscores the importance of loss function selection and embedding dimension in KGE model performance, with NNFS showing promise under specific configurations albeit with limitations in scaling to bigger datasets.

FB15k Dataset

The FB15k dataset, presenting 14,951 unique entities and 1,345 relations across 483,142 training triples, 50,000 validation triples, and 59,071 test triples, offers a rigorous testing environment for various knowledge graph embedding (KGE) models. When evaluating the metrics on this dataset, it is observed that the Neural Network Function Space (NNFS) model attains a competitive Mean Reciprocal Rank (MRR) of 0.360 under the Binary Cross Entropy (BCE) loss function with 2048 parametrized embedding function. This MRR is complemented by a Hits@1 of 0.233, Hits@3 of 0.418, and Hits@10 of 0.610, indicating a commendable performance in ranking true entities among the top predictions. Nevertheless, the model does not outperform all; particularly, the ComplEx model, which achieves an MRR of 0.467 with a superior H@10 of 0.675, suggests its enhanced capability to manage the dataset's extensive scale and complexity.

A closer examination of the Complex Number Function Space model reveals that, despite a slightly higher MRR of 0.383 under the L2 loss function, its performance declines notably with a decrease in the number of parameters, achieving an MRR of only 0.222 under the BCE loss function. This decline is indicative of the model's sensitivity to the number of parameters of the embedding function.

The contrast in model performances is more pronounced when comparing their outcomes at the smaller parameter size. Here, the NNFS model exhibits a marked decrease in MRR to 0.197 under the BCE loss function, which is lower than its Complex Number FS counterpart at the same dimension under the L2 loss function, suggesting a potential overfitting issue when NNFS is trained with larger embeddings.

Model	Loss Function MRR	
	L2	BCE
TransE [Bor+13]	**0.190**	0.268
DistMult [Yan+14]	0.157	**0.348**
ComplEx [Tro+16]	0.152	0.338
QMult [Dem+21]	0.153	0.319
OMult [Dem+21]	0.137	0.319
Keci [DN23]	0.167	0.340
ComplexNumber FS	0.171	0.223
NNFS	0.101	0.200

Tab. 5.12: MRR for different KGE models on the FB15k dataset using L2 and BCE loss functions with 64 parameters for the embedding function. Bold results indicate the best results.

Overall, these observations underscore the importance of considering embedding dimensions and loss functions in the performance of KGE models. While NNFS shows promise in specific configurations, its generalizability across various settings raises concerns, particularly when contrasted with the more consistent performance of models like ComplEx and the varied adaptability of the Complex Number FS model.

Loss Function	Model	Hits@1	Hits@3	Hits@10	MRR
BCE	TransE [Bor+13]	0.139	0.249	0.394	0.226
	DistMult [Yan+14]	**0.368**	**0.545**	**0.687**	**0.479**
	ComplEx [Tro+16]	0.354	0.531	0.675	0.466
	QMult [Dem+21]	0.270	0.406	0.546	0.363
	OMult [Dem+21]	0.296	0.419	0.545	0.380
	Keci [DN23]	0.243	0.397	0.550	0.348
	ComplexNumber FS	0.166	0.367	0.532	0.298
	NNFS	0.233	0.418	0.610	0.360
L2	TransE [Bor+13]	0.004	0.027	0.064	0.026
	DistMult [Yan+14]	0.009	0.021	0.055	0.028
	ComplEx [Tro+16]	0.002	0.006	0.021	0.011
	QMult [Dem+21]	0.000	0.001	0.003	0.002
	OMult [Dem+21]	0.000	0.000	0.000	0.000
	Keci [DN23]	0.050	0.118	0.245	0.115
	ComplexNumber FS	0.273	0.440	0.587	0.383
	NNFS	0.079	0.172	0.315	0.159

Tab. 5.13: Evaluation metrics for different KGE models on the FB15k dataset with 2048 parameters for the embedding function, using both BCE and L2 loss functions. Bold results indicate the best results.

5.3.3 Effects of Entity and Relation's Parameter Dimension on the KG Embedding Models

The entity and relation's parameter dimension is a fundamental hyperparameter in the realm of knowledge graph embeddings. It determines the size of the vector space in which the entities and relations are represented. A larger parameter dimension allows the model to capture more details and nuances of the data, potentially leading to better performance on tasks such as link prediction and entity resolution. However, increasing the dimensionality also raises the risk of overfitting, especially when the **dataset is small**, as the model may start to memorize the training data rather than learn to generalize from it.

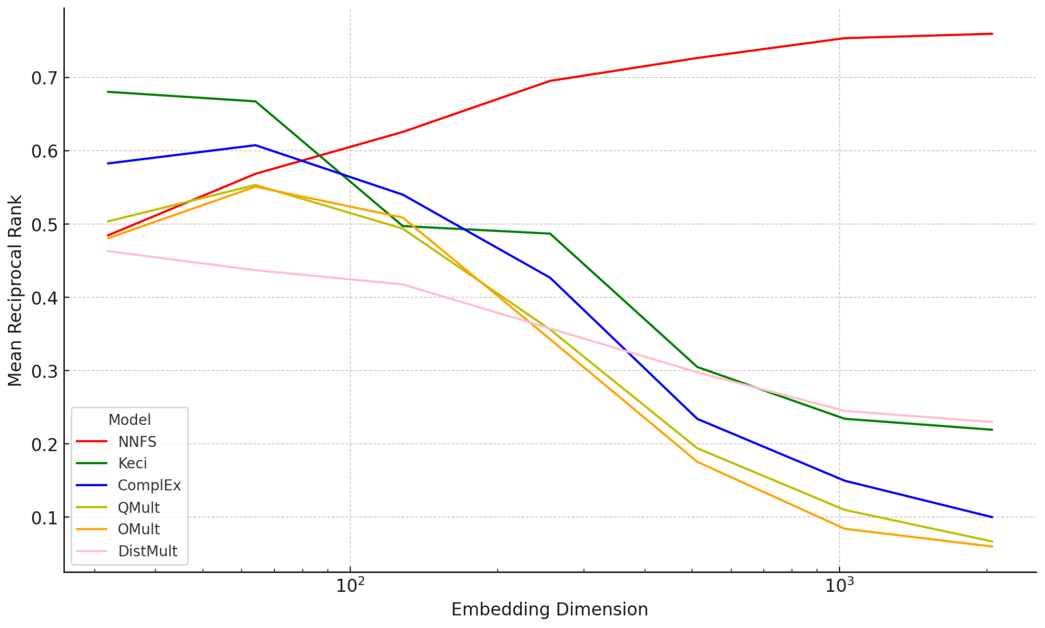


Fig. 5.7: Comparison of Mean Reciprocal Rank for Various Models on the Kinship Test Dataset across Parameter Dimensions Ranging from 32 to 2048.

As depicted in Figure 5.7, when the parameter dimension is increased, the MRR for various models, except for NNFS (Neural Network using Function Space), generally decreases on the Kinship test dataset. This trend indicates that most models tend to overfit with an increase in parameter dimension, improving performance on the training dataset but not on the validation and test sets. In contrast, NNFS exhibits an increase in accuracy with larger parameter dimensions, suggesting a better generalization capability. Unlike other models, NNFS does not exhibit diminished performance on the test set as the parameter dimension grows, which may be attributed to its ability to leverage the function space for a more robust representation that avoids overfitting.

5.3.4 Effects of different k Values in Negative Sampling Scoring Technique

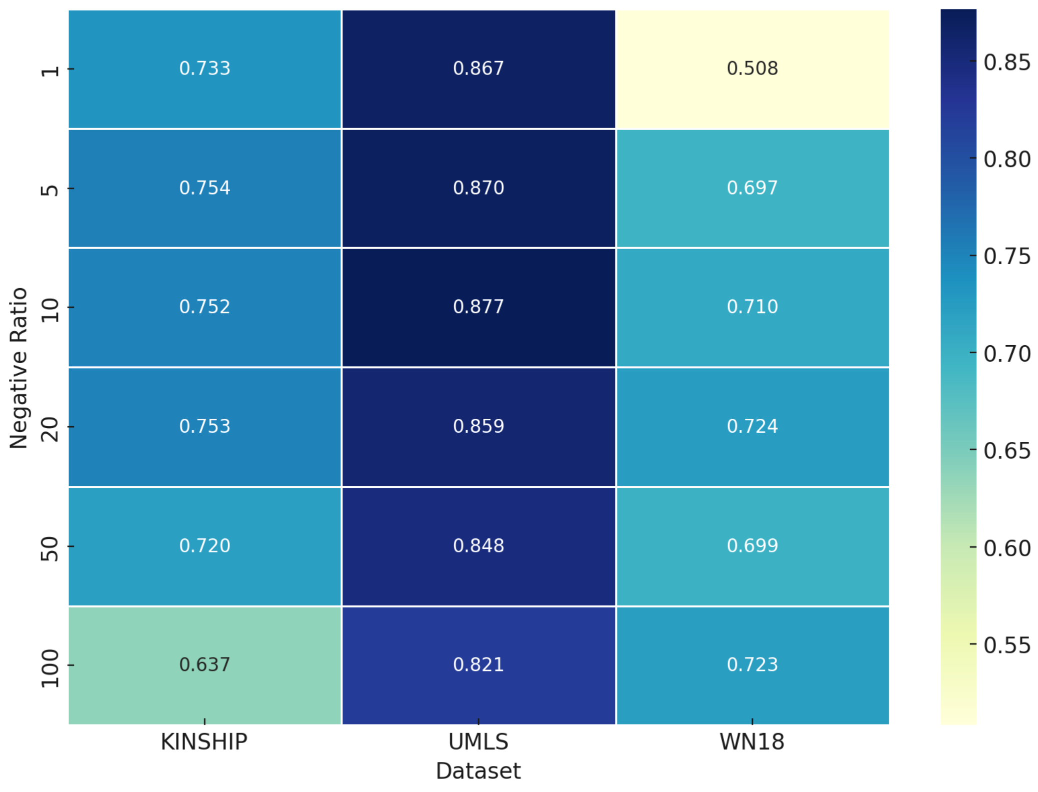


Fig. 5.8: Heatmap reflecting the impact of varying negative ratios (k) on Mean Reciprocal Rank (MRR) scores across the KINSHIP, UMLS, and WN18 datasets.

Based on the heatmap visualizing the impact of varying negative ratios (k) on Mean Reciprocal Rank (MRR) scores across the KINSHIP, UMLS, and WN18 datasets, we observe that the technique of negative sampling plays a significant role in the performance of knowledge graph embedding models. In our analysis, "k" represents the number of corrupted triples generated per positive triple, a critical parameter in negative sampling strategies. The heatmap reveals that the peak MRR values for these datasets are predominantly observed within the range of 10 to 20 for the negative ratio. This observation suggests an optimal balance in the quantity of negative samples, which is crucial for effective model training and performance.

5.3.5 Neural Architecture Search Experiments

In this subsection, we discuss our evaluation method for finding effective neural network designs using Neural Architecture Search (NAS). We chose the NNI framework by Microsoft [Mic20] because it supports various search algorithms and offers helpful visualization tools. NNI made it easier to try out different network structures and see which ones worked best, streamlining our search for the most suitable architecture.

```
{
  "activation_function0": {
    "_type": "choice",
    "_value": ["relu", "tanh", "sigmoid", "identity"]
    .
    .
    .
  "activation_functionN": {
    "_type": "choice",
    "_value": ["relu", "tanh", "sigmoid", "identity"]
  },
    "residual_connection": {
    "_type": "choice",
    "_value": [0, 1]
  },
  "layer_norm": {
    "_type": "choice",
    "_value": [0, 1]
  }
    "batch_norm": {
    "_type": "choice",
    "_value": [0, 1]
  },
  "num_layers": {
    "_type": "choice",
    "_value": [2, 3, 4, 5, 6]
  },
  "embedding_dim": {
    "_type": "choice",
    "_value": [64, 256, 512, 1024]
  }
  ---------
  For Conv
    "kernel_size": {
    "_type": "choice",
    "_value": [3, 5, 10, 20]
  }
}
```

Listing 5.1: Search space of the neural network model.

We utilized a defined search space to identify the optimal architecture for the Neural Network Function Space (NNFS). This exploration involved evaluating various architectures, including fully connected and convolutional layers, with differing layer counts, weight matrix dimensions based on the parameter vector dimension θ, and layer-specific activation functions denoted by "activationfunctionN", where N corresponds to the nth layer. Additionally, we assessed the efficacy of chain-structured versus residual architectures within our search space.

To run a comparative analysis on the search algorithm for our system, we chose our 2 layers Neural Network Function Space with the search space as defined above, which was a comparatively small search space and a reduced number of layers, we evaluated four tuners by conducting 50 trials on the Kinship dataset and assessed their performance based on speed and effectiveness. The tuners and their respective Mean Reciprocal Rank (MRR) were:

Tuner	**MRR**	**Description**
TPE	0.748	Tree-structured Parzen Estimator, a Bayesian optimization algorithm.
Evolution	0.715	Evolutionary algorithm-based tuner.
Grid Search	0.693	Exhaustive search over specified parameter values.
GP	0.712	Gaussian Process, a Bayesian optimization technique.
Hyperband	0.690	Hyperband, a bandit-based configuration evaluation method.

Tab. 5.14: Performance comparison of tuners based on Mean Reciprocal Rank (MRR) using the Kinship dataset.

In our architectural explorations, we also experimented with integrating convolutional layers alongside fully connected layers. The operation for each segment of the embedding can be expressed as follows, where X is the input, $W^{(i)}$ represents the weights of the i-th layer, and LayerNorm denotes layer normalization:

$$X^{(i+1)} = \text{LayerNorm}(W^{(i)} \cdot X^{(i)}) \tag{5.1}$$

For even-indexed layers, different activation functions σ were applied:

$$X^{(i+1)} = \text{LayerNorm}(\sigma(W^{(i)} \cdot X^{(i)})) \tag{5.2}$$

$$X^{(i+1)} = \begin{cases} \text{LayerNorm}(W^{(i)} \cdot X^{(i)}) & \text{if } i = 0 \\ \text{LayerNorm}(\text{ReLU}(W^{(i)} \cdot X^{(i)}) + X^{(i-1)}) & \text{if } i = 1 \text{ and residual connection} \\ \text{LayerNorm}(\text{ReLU}(W^{(i)} \cdot X^{(i)})) & \text{otherwise} \end{cases} \tag{5.3}$$

A variation of the setup tried was : The convolutional operations involved the application of convolution followed by pooling for the first layer, and similarly for the second, which can be formulated as:

$$X_{\text{conv}}^{(i+1)} = \begin{cases} \text{pool1}(\text{conv1}(\text{LayerNorm}(\sigma(W^{(i)} \cdot X^{(i)})))) & \text{if } i = 0 \\ \text{pool2}(\text{conv2}(X_{\text{conv}}^{(i)})) & \text{if } i = 1 \end{cases} \tag{5.4}$$

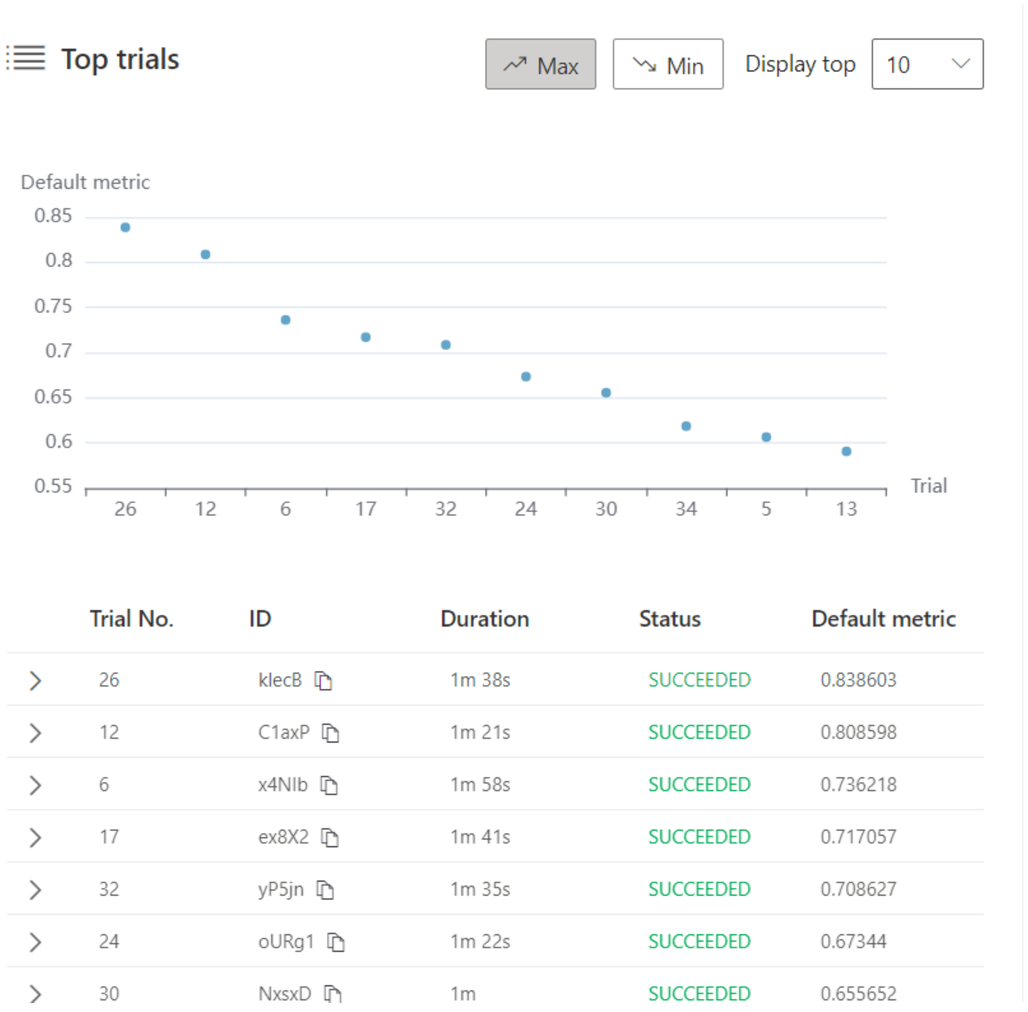
The output of the convolutional layers is then flattened and passed through a fully connected layer:

$$f(X) = \text{FC}(\text{flatten}(X_{\text{conv}})) \tag{5.5}$$

Following the selection of the Tree-structured Parzen Estimator (TPE) as our tuner, further trials indicated the effectiveness of layer normalization after each layer and the use of an activation function at every alternate layer. A comparison was made between a convolutional neural network architecture followed by a fully connected layer, as represented by the convolution and pooling operations, and a fully connected residual network as described by Equation 5.3. It was found that the two-layered residual setup with ReLU activation outperformed the convolutional architecture in our comparisons.

Also, we tried to search for the best-performing hyperparameter for our neural network using the NNI API. As described in the implementation section, we tried a variation of learning rate, number of epochs, dropout and weight decay for regularization.

In the overall evaluation, the Neural Architecture Search (NAS) showed great promise in automating the tedious job of finding the best-suited neural network architecture for customized applications. While it indeed demands significant computational resources, the potential benefits such as optimized performance, efficiency, and the ability to discover novel architectural patterns that might not be intuitive to human designers, outweigh these costs. Moreover, the advancement in NAS methodologies, including efficient search strategies and meta-learning approaches, continues to reduce the computational overhead, making NAS more accessible and practical for a wider range of applications. The exploration of NAS in our study not only highlighted its capability to enhance model performance but also opened avenues for future research in automating machine learning pipeline optimization, thereby pushing the boundaries of what's possible in AI design and application.



Trial No.	ID	Duration	Status	Default metric
26	klecB	1m 38s	SUCCEEDED	0.838603
12	C1axP	1m 21s	SUCCEEDED	0.808598
6	x4NIb	1m 58s	SUCCEEDED	0.736218
17	ex8X2	1m 41s	SUCCEEDED	0.717057
32	yP5jn	1m 35s	SUCCEEDED	0.708627
24	oURg1	1m 22s	SUCCEEDED	0.67344
30	NxsxD	1m	SUCCEEDED	0.655652

Fig. 5.9: NNI [Mic21] framework showing different trials results. MRR of 0.83 was achieved on the UMLS test dataset using Neural Network Function Space.

6 Conclusion

In this thesis, we ventured into the innovative domain of representing embeddings as functions, a marked departure from the conventional vector-based embeddings prevalent in traditional Knowledge Graph (KG) embedding methods such as TransE, ComplEx, and DistMult. The crux of this approach involved conceptualizing embeddings as functions of model parameters (θ) and a domain variable (x), thereby leveraging different quadrature methods to approximate these functions using sample points from a sub-domain X. This methodology was predicated on the hypothesis that function-based embeddings, with their expanded parameter space and increased degrees of freedom, could potentially offer superior optimization and model performance in comparison to their vector-based counterparts.

6.1 Tools and Methodology

Our exploration utilized polynomial function space, complex number function spaces and neural network function spaces, which showed promising results, particularly for standard datasets such as UMLS and KINSHIP. The complex number function space excelled due to its ability to map parameter phases and magnitudes effectively, while the neural network function space introduced beneficial non-linearity, capturing more complex relational patterns within the dataset. Three scoring functions were investigated: compositional, vector triple product, and trilinear scoring functions. The compositional scoring function emerged as the most effective, leveraging the concept of functional composition to intertwine head embeddings with relation embeddings in a novel manner. Conversely, the vector triple product scoring function underperformed due to its gradient issues concerning the tail entity, and the trilinear scoring function, while competent, did not match the efficacy of the compositional approach.

Our investigation into loss functions revealed that binary cross-entropy generalized well across different function spaces, with L2 Norm and Margin Loss also showing commendable performance, particularly in conjunction with complex number function spaces. Addressing the challenge of identifying optimal neural network architectures, we employed Neural Architecture Search (NAS), utilizing the Neural Network Intelligence framework developed by Microsoft. This approach facilitated the automation of architecture selection, with a two-layer neural network configuration demonstrating superior performance within our experimental setup.

6.2 Challenges and Observations

A critical observation was that both the complex number and neural network function spaces performed well with small-scale datasets but encountered challenges as the dataset size increased. While these function spaces demonstrated good generalization capabilities when model parameter dimensions were scaled up, they did not exhibit the same level of performance with larger datasets, mitigating overfitting issues seen in other models but struggling with scalability.

6.3 Future Directions

Looking ahead, a more thorough investigation into Neural Architecture Search (NAS) could yield more efficient architectures that not only perform well but also scale effectively to larger datasets. Additionally, the development and evaluation of novel scoring functions that adeptly capture the nuanced interactions between entities and relations across diverse relation types could further bolster the efficacy and applicability of function-based KG embeddings. This future work holds the promise of advancing the field of KG embeddings by harnessing the full potential of function-based representations to tackle complex relational patterns and large-scale knowledge graph challenges. Another future direction to look at is the introduction of an inductive setting instead of the transductive setting where we expect the KGE model to generalize and perform link prediction in case of the new unseen entities.

Bibliography

[Aue+07] S Auer, C Bizer, G Kobilarov, et al. “Dbpedia: A nucleus for a web of open data”. In: *Proceedings of the Semantic Web, International Semantic Web Conference, Asian Semantic Web Conference, ISWC 2007 + ASWC 2007*. Busan, Korea, 2007, pp. 722–735 (cit. on p. 5).

[Ber+11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. “Algorithms for Hyper-Parameter Optimization”. In: *Advances in neural information processing systems*. NeurIPS. 2011 (cit. on p. 32).

[BB12] James Bergstra and Yoshua Bengio. “Random search for hyper-parameter optimization”. In: *Journal of Machine Learning Research* 13.Feb (2012), pp. 281–305 (cit. on p. 19).

[Bol+08] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. “Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge”. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. Vancouver, BC, Canada, Sept. 2008, pp. 1247–1250 (cit. on p. 5).

[Bor+13] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. “Translating embeddings for modeling multi-relational data”. In: *Advances in Neural Information Processing Systems*. 2013, pp. 2787–2795 (cit. on pp. 1, 4, 7, 10, 11, 24, 36, 67–69, 71, 72, 74).

[Dai+ar] Yuanfei Dai, Shiping Wang, Neal N. Xiong, and Wenzhong Guo. “A Survey on Knowledge Graph Embedding: Approaches, Applications and Benchmarks”. In: *Name of Journal* (Year). Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou University, Fuzhou 350108, China and Department of Mathematics and Computer Science, Northeastern State University, Tahlequah, OK 003161, USA (cit. on pp. 1, 29).

[Dem+21] Caglar Demir, Diego Moussallem, Stefan Heindorf, and Axel-Cyrille Ngonga Ngomo. “Convolutional Hypercomplex Embeddings for Link Prediction”. In: *Proceedings of the 13th Asian Conference on Machine Learning (ACML 2021)*. arXiv:2106.15230 [cs.LG]. 2021. arXiv: `2106.15230 [cs.LG]` (cit. on pp. 4, 9, 67–69, 71, 72, 74).

[DN23] Caglar Demir and Axel-Cyrille Ngonga Ngomo. “Clifford Embeddings – A Generalized Approach for Embedding in Normed Algebras”. In: *Machine Learning and Knowledge Discovery in Databases: Research Track*. Springer, Sept. 2023 (cit. on pp. 4, 9, 67–69, 71, 72, 74).

[DN22] Caglar Demir and Axel-Cyrille Ngonga Ngomo. “Hardware-agnostic computation for large-scale knowledge graph embeddings”. In: *Software Impacts* (2022) (cit. on p. 57).

[Det+18] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. “Convolutional 2D Knowledge Graph Embeddings”. In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*. 2018 (cit. on pp. 8, 10, 12, 58, 59).

[Don+15] Li Dong, Furu Wei, Ming Zhou, and Ke Xu. “Question Answering over Freebase with Multi-Column Convolutional Neural Networks”. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 2015, pp. 260–269 (cit. on p. 1).

[EMH19] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. “Neural Architecture Search: A Survey”. In: *Journal of Machine Learning Research* 20.55 (2019), pp. 1–21 (cit. on p. 18).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016 (cit. on p. 12).

[] *Hadamard product (matrices) - Wikipedia — en.wikipedia.org*. `https://en.wikipedia.org/wiki/Hadamard_product_(matrices)`. [Accessed 11-01-2024] (cit. on p. 33).

[He+16a] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016 (cit. on p. 30).

[He+16b] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cit. on p. 13).

[Hin90] Geoff Hinton. *Kinship*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5WS4D. 1990 (cit. on p. 58).

[Hoy+22] Charles Tapley Hoyt, Max Berrendorf, Mikhail Galkin, Volker Tresp, and Benjamin M Gyori. “A Unified Framework for Rank-based Evaluation Metrics for Link Prediction in Knowledge Graphs”. In: *arXiv preprint arXiv:2203.07544* (2022) (cit. on p. 60).

[Hua+17] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. “Densely connected convolutional networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708 (cit. on p. 13).

[IS15] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167* (2015) (cit. on p. 14).

[Kri18] Arun Krishnan. *Making search easier: How Amazon's Product Graph is helping customers find products more easily*. `https://blog.aboutamazon.com/innovation/making-search-easier`. Accessed: 2022-07-03. 2018 (cit. on p. 5).

[LUO18] Timothée Lacroix, Nicolas Usunier, and Guillaume Obozinski. "Canonical Tensor Decomposition for Knowledge Base Completion". In: *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 2018 (cit. on pp. 10, 12).

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444 (cit. on p. 12).

[Lia+22] Zongwei Liang, Junan Yang, Hui Liu, et al. "SeAttE: An Embedding Model Based on Separating Attribute Space for Knowledge Graph Completion". In: *Electronics* 11.7 (2022), p. 1058 (cit. on p. 1).

[MBS14] Farzaneh Mahdisoltani, Joanna Biega, and Fabian Suchanek. "YAGO3: A knowledge base from multilingual Wikipedias". In: *7th Biennial Conference on Innovative Data Systems Research (CIDR'14)* (2014) (cit. on p. 58).

[Mic20] Microsoft. *Neural Network Intelligence*. `https://github.com/microsoft/nni`. 2020 (cit. on pp. 4, 54, 77).

[Mic21] Microsoft. *Neural Network Intelligence*. Version 2.0. Jan. 2021 (cit. on pp. 31, 57, 80).

[Mik+13a] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeff Dean. "Efficient Estimation of Word Representations in Vector Space". In: *International Conference on Learning Representations*. 2013 (cit. on p. 1).

[Mik+13b] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. "Distributed Representations of Words and Phrases and their Compositionality". In: *Advances in Neural Information Processing Systems*. 2013 (cit. on p. 1).

[NTK11] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. "A Three-Way Model for Collective Learning on Multi-Relational Data". In: *Proceedings of the 28th International Conference on Machine Learning*. 2011, pp. 809–816 (cit. on pp. 1, 12).

[PNL02] A. Pease, I. Niles, and J. Li. "The Suggested Upper Merged Ontology: A Large Ontology for the Semantic Web and Its Applications". In: *Proceedings of the Working Notes of the AAAI-2002 Workshop on Ontologies and the Semantic Web*. Vol. 28. Edmonton, AB, Canada, 28–29 July 2002, pp. 7–10 (cit. on p. 5).

[Pit+17] R.J. Pittman, Amit Srivastava, Sanjika Hewavitharana, Ajinkya Kale, and Saab Mansour. *Cracking the Code on Conversational Commerce*. `https://www.ebayinc.com/stories/news/cracking-the-code-on-conversational-commerce/`. Accessed: 2022-07-03. 2017 (cit. on p. 5).

[Rea+19] Esteban Real et al. “Regularized Evolution for Image Classifier Architecture Search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 4780–4789 (cit. on p. 19).

[Rea+17] Esteban Real, Sherry Moore, Andrew Selle, et al. “Large-Scale Evolution of Image Classifiers”. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, June 2017, pp. 2902–2911 (cit. on p. 32).

[Sch73] Edward W Schneider. “Course Modularization Applied: The Interface System and Its Implications For Sequence Control and Data Analysis”. In: 1973 (cit. on p. 5).

[Sin12] Amit Singhal. *Introducing the Knowledge Graph: things, not strings*. `https://www.blog.google/products/search/introducing-knowledge-graph-things-not-strings/`. Accessed: 2022-07-03. 2012 (cit. on pp. 1, 5).

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *Advances in Neural Information Processing Systems*. 2012 (cit. on p. 19).

[Sri+14] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. “Dropout: A simple way to prevent neural networks from overfitting”. In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958 (cit. on p. 15).

[SKW07] FM Suchanek, G Kasneci, and G Weikum. “Yago: A core of semantic knowledge”. In: *Proceedings of the 16th International Conference on World Wide Web*. ACM. 2007, pp. 697–706 (cit. on pp. 5, 58).

[Sun+19a] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. “A Survey of Optimization Methods from a Machine Learning Perspective”. In: *arXiv preprint arXiv:1906.06821* (2019) (cit. on p. 16).

[Sun+19b] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. “RotatE: Knowledge graph embedding by relational rotation in complex space”. In: *International Conference on Learning Representations*. 2019 (cit. on p. 11).

[TC15] Kristina Toutanova and Danqi Chen. “Observing the unobservable: Learning to infer the compositionality of natural language”. In: *arXiv preprint arXiv:1506.03667* (2015) (cit. on p. 58).

[Tro+16] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. “Complex embeddings for simple link prediction”. In: *International Conference on Machine Learning*. 2016, pp. 2071–2080 (cit. on pp. 1, 4, 8, 10, 67–69, 71, 72, 74).

[VK14] D Vrandečić and M Krötzsch. “Wikidata: A free collaborative knowledgebase”. In: *Communications of the ACM* 57 (2014), pp. 78–85 (cit. on p. 5).

[Wan+17] Quan Wang, Zhenguang Mao, Bin Wang, and Li Guo. "Knowledge graph embedding: A survey of approaches and applications". In: *IEEE Transactions on Knowledge and Data Engineering* 29.12 (2017), pp. 2724–2743 (cit. on p. 6).

[Xu+16] Kun Xu, Siva Reddy, Yansong Feng, Songfang Huang, and Dongyan Zhao. "Question Answering on Freebase via Relation Extraction and Textual Evidence". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016, pp. 2326–2336 (cit. on p. 1).

[Yan+14] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. "Embedding entities and relations for learning and inference in knowledge bases". In: *International Conference on Learning Representations (ICLR)*. 2014 (cit. on pp. 1, 4, 8, 24, 67–69, 71, 72, 74).

[ZL16] Barret Zoph and Quoc V Le. "Neural Architecture Search with Reinforcement Learning". In: *arXiv preprint arXiv:1611.01578* (2016) (cit. on p. 19).

List of Figures

List of Tables

List of Listings

Declaration

I hereby affirm that I have independently composed this work and have not used any sources or aids other than those indicated.

Paderborn, February 15, 2023

Vikrant

Vikrant Singh