UNIVERSITY OF CAPE TOWN

MASTERS THESIS

From GNNs to Sparse Transformers: Graph-based architectures for Multi-hop Question Answering

---

Author:
Shane ACTON

Supervisor:
Dr. Jan BUYS

University of Cape Town

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

in the faculty of Science

January 20, 2023

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Declaration of Authorship

I, Shane ACTON, declare that this thesis titled, "From GNNs to Sparse Transformers: Graph-based architectures for Multi-hop Question Answering" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Signed by candidate

Date: January 20, 2023

Acknowledgements

I would like to thank my Supervisor Dr. Jan Buys for his consistent interest and advice regarding this work. His guidance lead me to this project, and his experience lead me through it.

Next I would like to thank my peer, Sasha Abramowitz, for helping to keep me sane along the journey. Without at all meaning to deride the Masters program, I will say that misery loves company.

Finally, I would like to thank The Centre for Artificial Intelligence Research (CAIR) for funding my studies, thereby providing me with the opportunity to participate in this project. The CAIR is primarily funded by the Department of Science and Innovation (DSI), a South African governmental division. Thus all thanks naturally extend to the DSI and the Republic of South Africa.

UNIVERSITY OF CAPE TOWN

Abstract

Science Faculty
Department of Computer Science

Masters in Computer Science

From GNNs to Sparse Transformers: Graph-based architectures for Multi-hop Question Answering

by Shane ACTON

Multi-hop Question Answering (MHQA) is a challenging task in NLP which typically involves processing very long sequences of context information. Sparse Transformers [114] have surpassed Graph Neural Networks (GNNs) as the state-of-the-art architecture for MHQA. Noting that the Transformer [101] is a particular message passing GNN, in this work we perform an architectural analysis and evaluation to investigate why the Transformer outperforms other GNNs on MHQA. In particular, we compare attention- and non-attention-based GNNs, and compare the Transformer's Scaled Dot Product (SDP) attention to the Graph Attention Network [102] (GAT)'s Additive Attention [6]. We simplify existing GNN-based MHQA models and leverage this system to compare GNN architectures in a lower compute setting than token-level models. We evaluate all of our model variations on the challenging MHQA task Wikihop [106]. Our results support the superiority of the Transformer architecture as a GNN in MHQA. However, we find that problem-specific graph structuring rules can outperform the random connections used in Sparse Transformers. We demonstrate that the Transformer benefits greatly from its use of residual connections [39], Layer Normalisation [5], and element-wise feed forward Neural Networks, and show that all tested GNNs benefit from this too. We find that SDP attention can achieve higher task performance than Additive Attention. Finally, we also show that utilising edge type information alleviates performance losses introduced by sparsity.

Contents

Declaration of Authorship i

Acknowledgements ii

Abstract iii

List of Abbreviations vii

Table of Symbols viii

1 Introduction 1
- 1.1 Research Goals . . . 4
- 1.2 Findings and Contributions . . . 5
- 1.3 Overview of Thesis Structure . . . 5

2 Background and Related Work 7
- 2.1 Question Answering . . . 7
- 2.2 Multihop Question Answering Datasets . . . 7
  - 2.2.1 The Stanford Question Answering Dataset . . . 8
  - 2.2.2 TriviaQA . . . 8
  - 2.2.3 Wikihop Dataset . . . 8
  - 2.2.4 HotpotQA Dataset . . . 10
- 2.3 Recurrent Neural Networks . . . 10
- 2.4 Transformers . . . 11
  - 2.4.1 Scaled Dot-Product Attention . . . 12
  - 2.4.2 Multi-Headed Attention . . . 14
  - 2.4.3 Transformer Encoder . . . 14
  - 2.4.4 Sparse Transformers . . . 16
  - 2.4.5 Relative Positional Embeddings . . . 17
- 2.5 Graph Neural Networks . . . 19
  - 2.5.1 Message Passing Graph Neural Networks . . . 20
  - 2.5.2 GNN architectures . . . 22
    - 2.5.2.1 GraphSAGE . . . 23
    - 2.5.2.2 Graph Attention Neural Network . . . 24
    - 2.5.2.3 Transformer as a GNN . . . 25
    - 2.5.2.4 Gating . . . 26
    - 2.5.2.5 Relational GNNs . . . 27
- 2.6 GNN-based MHQA models . . . 28
  - 2.6.1 Graph Construction . . . 29
    - 2.6.1.1 GSPR . . . 29
    - 2.6.1.2 EGCN . . . 30
    - 2.6.1.3 HDE . . . 30

2.6.1.4 HGN . . . 31
2.6.1.5 Summary . . . 31
2.6.2 Graph Embedding . . . 32
2.6.2.1 Token Embedding . . . 32
2.6.2.2 C-GRU . . . 34
2.6.2.3 GSPR . . . 34
2.6.2.4 EGCN . . . 35
2.6.2.5 HDE . . . 35
2.6.2.6 HGN . . . 35
2.6.3 GNN Architectures . . . 36
2.6.3.1 GSPR GNN . . . 36
2.6.3.2 EGCN and HDE GNN . . . 37
2.6.3.3 HGN GNN . . . 37
2.6.4 Output models . . . 38
2.6.5 Summary . . . 39

3 Model **40**
3.1 Graph Construction . . . 40
3.2 Graph Embedding . . . 43
3.2.1 Token Embedding . . . 43
3.2.2 Context-Query Coattention . . . 44
3.2.3 Summariser . . . 46
3.2.4 Switch-Summariser . . . 47
3.3 GNN Graph Encoding . . . 47
3.3.1 GNN Cores . . . 48
3.3.1.1 Switch-Core . . . 48
3.3.1.2 Edge-Core . . . 48
3.3.1.3 SAGE-Core . . . 48
3.3.1.4 GAT-Core . . . 48
3.3.1.5 SDP-Core . . . 49
3.3.1.6 GNN-Core summary . . . 49
3.3.2 Update Functions and In-Out Asymmetries . . . 49
3.3.2.1 No Asymmetry . . . 50
3.3.2.2 SAGE Asymmetry . . . 50
3.3.2.3 MLP Asymmetry . . . 50
3.3.2.4 Gate Asymmetry . . . 50
3.3.2.5 Transformer Update Function Asymmetry . . . 50
3.3.2.6 Composing Asymmetries . . . 50
3.4 Candidate Selection Output Model . . . 51

4 Experimental Method **52**
4.1 Implementation . . . 52
4.1.1 Data Processing . . . 53
4.1.2 Sparse and Dense GNN Implementations . . . 55
4.2 Hyperparameters . . . 56
4.3 Experimental Procedure . . . 57
4.4 Benchmark Models . . . 58
4.4.1 HDE-Base . . . 59
4.4.2 Att-Base . . . 59
4.4.3 BERT-Base . . . 60
4.5 Evaluation Method . . . 60

4.6 Model Evaluation Variation . . . 61
4.7 Test Fairness . . . 61
4.8 Data Analysis . . . 62
4.8.1 Wikihop Data Points . . . 62
4.8.2 Constructed Graphs . . . 64

5 Experiment Results 67
5.1 Main Results . . . 67
5.2 GNN Architecture . . . 67
5.2.1 GNN Cores and In-Out Asymmetries . . . 68
5.2.1.1 GNN Core . . . 68
5.2.1.2 In-Out Asymmetries . . . 69
5.2.2 Edge Information . . . 71
5.2.3 Weight Sharing . . . 71
5.3 Graph Construction . . . 72
5.3.1 Graph Structure . . . 73
5.3.2 Special Entities vs Detected Entities . . . 73
5.3.3 Sentence Nodes . . . 74
5.3.4 Bidirectional Edges . . . 75
5.3.5 Edges Type Inclusion . . . 75
5.4 Graph Embedding . . . 75
5.4.1 Token Embedder . . . 76
5.4.2 Switch Summariser . . . 76
5.4.3 GRU encoders . . . 76
5.5 Sparse and Dense GNN Implementations Results . . . 77
5.6 Model Parameters . . . 80
5.7 Results Summary . . . 80
5.7.1 GNN architecture . . . 81
5.7.2 Edge Information . . . 81
5.7.3 Graph Structure . . . 81
5.8 Research Questions . . . 82

6 Discussion 83
6.1 Limitations . . . 84
6.2 Generalisability . . . 84
6.3 Future work . . . 86
6.4 Final Thoughts on the Long Context Problem . . . 86

7 Conclusion 87

A Appendix 88

Bibliography 90

List of Abbreviations

NLP Natural Language Processing
QA Question Answering
MHQA Multihop Question Answering

MLP Multilayer Perceptron
RBM Restricted Boltzmann Machine
DNN Deep Neural Network
SGD Stochastic Gradient Decent

GNN Graph Neural Network
GAT Graph Attention Neural Network
HDE Heterogeneous Document Entity system
GSPR Graph Structured Passage Representation
EGCN Entity Graph Convolutional Network
HGN Hierarchical Graph Network

RNN Recurrent Neural Network
GRU Gated Recurrent Unit
LSTM Long Short-Term Memory

GloVe Global Vectors for Word Representation
BERT Bidirectional Encoder Representations from Transformers
BPE Byte-Pair Encoding

RPE Relative Positional Embedding
APE Absolute Positional Embedding
DAG Directed Acyclic Graph
SOTA State of the art

Table of Symbols

Symbol	Description
$\boldsymbol{q}$	A textual question
$\boldsymbol{S_q}$	A set of supporting textual documents to help answer question q
$\boldsymbol{C_q}$	A set of potential answers to q
$\boldsymbol{l}$	The length of a sequence
$\boldsymbol{f}$	The number of features of a vector/ model
$\boldsymbol{X}$	An l by f Matrix storing a sequence of token vectors
$\boldsymbol{x_i}$	A token vector with f features
$\boldsymbol{Y}$	A query-aware Matrix storing a sequence of token vectors
$\boldsymbol{n}$	The number of nodes in a graph
$\boldsymbol{L}$	The number of layers in a GNN
$\boldsymbol{H}$	An n by f Matrix storing a sequence of encoded node state vectors
$\boldsymbol{h_i^k}$	The state vector for node i after k layers of GNN encoding
$\boldsymbol{\phi^k}$	The message function of a GNN at layer k
$\boldsymbol{\omega^k}$	The aggregate function of a GNN at layer k
$\boldsymbol{\gamma^k}$	The update function of a GNN at layer k
$\boldsymbol{A_i^k}$	The output of the aggregate function at layer k for node i. This represents the aggregate message being passed to node i, from all neighbours
$\boldsymbol{M_{i,j}^k}$	Partial output of the message function at layer k. This represents the message passed from node i to j.
$\boldsymbol{N(i)}$	The node indices of the neighbours to node i
$\boldsymbol{E}$	An adjacency matrix/ edge matrix

TABLE 1: Symbols table

Chapter 1

Introduction

Question Answering (QA) [94] is a supervised task in Natural Language Processing (NLP) that tests a computer system's ability to interpret and reason about natural language text. QA tasks are typically structured to include a plain text query which contains a question often in natural language. Accompanying the query is a collection of context information which is represented by natural language. Finally, the QA data point will contain a ground-truth answer to the query. Typically, the context and query are used as inputs to a model, and the answer is the output the model is trained to produce.

In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under **gravity**. The main forms of precipitation include drizzle, rain, sleet, snow, **graupel** and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals **within a cloud**. Short, intense periods of rain in scattered locations are called "showers".

What causes precipitation to fall?
gravity

What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?
graupel

Where do water droplets collide with ice crystals to form precipitation?
within a cloud

FIGURE 1.1: An example figure from SQuAD [79] where each answer is a span of text found in the context.

The Stanford Question Answering Dataset (SQuAD) [79] is a a notable task in NLP whereby a model must find the answers to questions in a given context. An answer is represented by a start and end word/token in the context. Figure 1.1 shows a SQuAD example with one context document, and three question and answer pairs. Models which successfully answer SQuAD questions demonstrate a basic understanding of natural language; however, it is unclear how deep this understanding is. From this SQuAD example, we can see that the information required to answer any of the questions is found within a single statement. Furthermore, we can see that many of the question words are found in the statement containing the answer. This may allow models to exploit simple strategies to achieve high scores [112].

a) Context:
1) Table Mountain is 1086m high
Query:
How tall is Table Mountain?

b) Context:
1) Table Mountain is in Cape Town
2) Cape Town is a city in South Africa
Query:
What country is Table Mountain in?

FIGURE 1.2: A single hop question (a), compared to a multihop question (b)

Multihop question answering (MHQA) problems such as Wikihop [106] or HotpotQA [112] require the combination of multiple pieces of evidence to solve [106]. Figure 1.2 contrasts a multihop question (b) with a single-hop question (a). A single hop question is one where a single statement is sufficient to answer the question. Here, the 'hop' refers to reasoning steps which are required. A single hop problem requires a single step of reasoning to go from the question to the statement containing the answer. A multihop problem requires reasoning from the question, through multiple pieces of evidence, and finally to the answer. From figure 1.2, we can see that the multihop question (b) requires the combination of two pieces of evidence, which are linked via the common entity 'Cape Town'. Multihop problems become difficult when the important evidence pieces are scattered among long sequences of irrelevant context. Machine Learning (ML) has recently been applied to MHQA [99, 114] and QA more generally [82]. Recurrent Neural Networks (RNN) were an early approach to applying Deep Neural Networks (DNN) to the NLP domain [82]. However, these RNN solutions suffered problems when processing long text sequences [30].

In this work, we discuss and compare two newer families of DNN solutions to the challenging MHQA task:

- Transformer-based solutions, which use attention mechanisms [101, 6], and typically operate using words/tokens to represent context information.
- Graph Neural Network (GNN)-based solutions that use rules to convert text into graphs, which are then processed. Here, graph nodes represent multiple words/tokens each. Nodes could represent named entities [62, 99], sentences [31], or even whole documents [99, 31].

Both GNNs [99, 26, 31, 95] and Transformers [8, 114] have been separately shown to be effective at answering Wikihop's multihop questions. Central to this work is the observation that the general nature of GNNs means that Transformer-based solutions can be considered a subset of GNN-based solutions [92, 110, 7]. In practice however, there are a number of key differences between these two families. In this thesis, we aim to evaluate some of these differences in the context of the multihop QA dataset Wikihop [106].

Transformers. In recent years, Transformer-based sequence models have been applied to virtually every area of ML [11, 88, 36, 115, 50]. Most relevant here is that Transformers have been central to a wave of innovations in NLP with models like Bidirectional Encoder Representations from Transformers (BERT) [27], and Generative Pretrained Transformer (GPT)

3 [11] demonstrating the potential power of large pretrained Transformer-based models in NLP. Pretraining refers to the pretrain/fine-tune paradigm whereby large amounts of data are used to train a model in a self-supervised manner, before a supervised task is trained on. For Transformers in NLP, text is broken up into tokens [27], which represent words, or pieces of words [90]. The Transformer model can then be used to encode and decode vector representations of these token sequences, for various tasks [101, 27, 77, 11].

At the core of the Transformer is the self-attention module [101] that computes the value of a compatibility function for pairs of tokens. Full self-attention Transformers, like BERT, compare/attend all n^2 pairs of tokens, resulting in memory requirements which scale quadratically with the length of the input sequence [101, 114, 8]. This issue has given rise to various avenues of inquiry, all centered around reducing the Transformer's memory requirements for long sequences. Sparse Transformers work by only attending a subset of sequence element pairs [8, 114]. However, it has been shown that sparsity reduces the modeling capacity of a Transformer model [114].

Graph Neural Networks. GNNs are a diverse family of Neural Networks (NN) architectures which all aim to process graphs made up of nodes and edges [108]. Nodes are represented by f-dimensional vectors, and edges may or may not have features [108]. Some GNN classes involve restrictions in the topology of graphs they are able to process [108]. However, we exclusively consider a class of GNN called Message Passing GNNs (MP-GNN), which are able to process arbitrary graph topologies. This is achieved by decoupling the GNN's parameters from graph properties such as nodes and edges. When thinking of the Transformer as a GNN, it follows that Transformers operate on fully connected graphs, since all element pairs communicate. Sparse Transformers transcend this, being able to communicate information between elements with arbitrary topology. Despite the similarity between Sparse Transformers and GNNs, Transformer and GNN approaches to MHQA differ substantially.

In multihop question answering (MHQA), GNN-based systems encode coarse-grained nodes which represent multiple words. In contrast, Sparse Transformers encode fine-grained tokens, contributing to high memory requirements even with sparsity [114]. Practically, this should allow for much larger collections of text to be processed via a GNN-based MHQA system than with Transformers. In MHQA, GNN node representations vary depending on the model, but a common node type is named-entities [62, 99, 26, 95, 13, 31]. GNN-based MHQA systems can also include other node types such as answer candidate [99], query [31], sentence [31], or whole document nodes [99, 31]. The graph-construction process is typically done via a hand-built heuristic process [99, 31, 26, 95]. It is important to note what GNN's could be used to encode fine-grained tokens, as the Transformer does. However, by convention, GNN-based MHQA models encode coarse-grained multi-word nodes [99, 26, 95, 13, 31].

GNN-based MHQA systems typically make use of multilayer perceptrons (MLP) and linear transformations to communicate information around graph nodes [99, 26, 95]. Attention, as is used in the Transformer, can be incorporated into GNNs. The Graph Attention Network (GAT) [102] uses additive self-attention to communicate between nodes. Additive attention is a variation of the scaled dot-product attention common in Transformers [101]. The GAT has been applied in the MHQA context [31], however, the authors did not evaluate on Wikihop. Thus, attention-based GNNs have not yet been evaluated on Wikihop.

The Transformer can be substituted in for the GNN in existing GNN-based MHQA models. Thus, the work done in the GNN-based MHQA literature could be repurposed to help alleviate the long sequence problem in Transformer-based NLP. This is because the GNN-based

MHQA models reduce their compute requirements by using coarse-grained node representations instead of the token-level representation common in Transformer-based MHQA [114]

Sparse Transformer models commonly make use of a locality bias, where nearby tokens are connected to each other [8, 114]. Some GNN-based systems incorporate locality by connecting entities which are nearby in text [26], others represent text in a hierarchical form by encapsulating nodes representing entities, sentences, and/or whole documents [99, 31]. Our work focuses on a particular hierarchical GNN-based MHQA system called the Heterogeneous Document-Entity (HDE) [99] system.

There is precedence for modeling edge type information in both the GNN and Transformer literatures. GNNs typically make use of distinct linear transformations for each edge type [86, 45, 99]. Transformers however make use of edge type embedding vectors which are summed into the model [92].

1.1 Research Goals

We aim to use Wikihop to train and evaluate various GNN-based MHQA models in order to discover which common methods in GNN-based and Transformer-based MHQA models work synergistically and which methods render others redundant. More specifically, our aim is to study how model performance is affected by different methods of communicating information between nodes, as well as methods of incorporating edge information. Since the Transformer-based approaches are more successful[1] than current GNN-based approaches, our default hypothesis is that wherever there is an architectural difference between the two, the method used by the Transformer is superior. Finally, we also provide an investigation into the use of different forms of graph structure in our GNN models.

More succinctly, our **goals** are as follows:

1. Evaluate the usage of attention-based GNNs in the context of the Wikihop MHQA dataset.
2. Evaluate the set of differences between a GAT-based and a Transformer-based MHQA model in the context of the Wikihop dataset. Primarily the use of different attention formulas, as well as the Transformer's use of residual connections [39] and Layer Normalisation [5].
3. Investigate the role that graph structure and sparsity has on GNN-based MHQA models.

Some **research questions** that naturally arise are:

1. Which components of the HDE MHQA model might improve Transformer-based MHQA models?
2. Do attention-based GNNs outperform non attention-based GNNs in the GNN-based MHQA setting?
3. Does edge type information improve GNN-based MHQA model performance, and if so does the Transformer-based approach to edge information improve over the GNN-based approach?

[1] WikihopLeaderboard:https://qangaroo.cs.ucl.ac.uk/leaderboard.html

4. Out of the many existing examples of graph structuring rules in the GNN literature, which common methods are important for improving model performance?

5. Can hand-crafted graph sparsity increase model performance when compared to fully connected graphs?

1.2 Findings and Contributions

Our contributions are as follows:

1. We offer a configurable open source GNN-based MHQA system for the Wikihop [106] and Medhop [106] datasets.

2. We offer a case study on the use of various architectural features of GNN within the context of MHQA. Here, we find that gating, as is common in GNN-based MHQA models, may not be required; GNNs perform better with distinct parameters per layer; edge type information is vitally important; and attention boosts performance in GNNs.

3. We offer a case study and analysis of the differences between the Transformer and various GNNs. We also provide motivation for the use of more Transformer-like GNNs. This includes the use of residual connections [39], Layer Normalisation [5], and Scaled Dot Product (SDP) [101] attention. This also includes the use of Transformer-style edge information instead of the prevailing GNN-style of edge information.

4. We provide an empirical analysis of the memory consumption of a prominent GNN implementation paradigm, and contrast it to the implementation approach of the Transformer. Our findings motivate the use of the Transformer implementation in most realistic use cases (see Sec 4.1.2).

5. We provide a recommendation for future Sparse Transformer development which includes the addition of edge type information and problem-specific graph structure.

1.3 Overview of Thesis Structure

Here we briefly discuss the structure of the remainder of the document. We begin by detailing the relevant background information (Chapter 2), primarily related to GNNs, Transformers and the problem of MHQA. We introduce the message passing notion as a universal way to describe any message passing GNN. We then conclude our background by describing some notable GNN-based MHQA models.

Next, we introduce our GNN-based MHQA system in the model chapter (Chapter 3). There we formalise each of the component variations we consider at each phase of the model pipeline. We then detail our experimental setup (Chapter 4), providing details on our implementation approach. There we offer an analysis and comparison of two prominent implementation approaches. We also describe our evaluation procedure.

From there we go on to provide the results of our experiments (Chapter 5). We offer an analysis of said results in an attempt to discover which features perform best. We provide a summary of the most important results before attempting to answer our research questions. Next, in our discussion (Chapter 6) we candidly explore the implications of our results as they relate to GNNs; Transformers; and MHQA models. We then attempt to answer our research questions and discuss directions for future work in the MHQA task setting.

Finally, we conclude our work by providing a concise and high level summary of our work, our findings and their implications (Chapter 7).

Chapter 2

Background and Related Work

In this chapter, we begin by describing some notable Multihop Question Answering (MHQA) datasets including the Wikihop [106] dataset, on which we focus. We provide a brief introduction to Recurrent Neural Networks [30] (RNN), which were used prominently in Natural Language Processing (NLP). We then introduce and detail the Transformer [101] and Graph Neural Network [108] (GNN) architectures. Finally, we discuss how RNN-based MHQA models lead naturally into GNN-based MHQA models, and provide details on some prominent GNN-based MHQA models.

2.1 Question Answering

Question Answering (QA) [94] is a supervised task which lies at the intersection of Information Retrieval (IR) and Natural Language Processing (NLP). A QA model is required to answer questions presented in either natural language, or possibly a structured query language. To do this, a QA model must use an understanding of natural language to process given context information, such as articles from Wikipedia. The model is required to locate information relevant to the query within the given context and use this information to compute an answer, typically in natural language. Multiple choice QA is a form of QA in which a set of possible answers are provided to the model, and the model must simply select the correct one.

QA is a large and diverse area of research. Recently, the number of new datasets created to train and test QA systems has increased dramatically [81]. In this work, we focus on a particular QA dataset called Wikihop. This was chosen due to its popularity in GNN-based QA systems. As such, our background information will focus on concepts and datasets which help the reader understand Wikihop and its related tasks. For a more comprehensive taxonomy of QA tasks, the interested reader could refer to [81]. Under this taxonomy, Wikihop would be considered a *complex multi step QA task* [81]. Beyond this point however, Wikihop will instead be referred to as a *multi-hop* QA task.

2.2 Multihop Question Answering Datasets

In recent years a number of multi-hop question answering (MHQA) datasets have been produced based off of the observation [106] that many previous QA datasets contained questions which could be answered using a single piece of information [79, 78], called single-hop questions. These single-hop questions can often be answered by simply finding the statement which related the question directly to the answer. Instead, MHQA requires the integration of multiple distinct pieces of information, sometimes contained in different documents. This is to say that for a multi-hop question, no single sentence is sufficient to

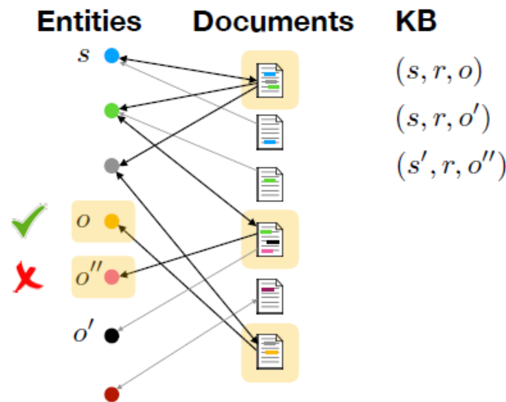

FIGURE 2.1: A figure from [106] showing the construction of a Wikihop question example via a graph traversal, using an external Knowledge Base (KB). The process yields a set of documents, answer candidates, a question, and a correct answer.

answer the question. Here, we will introduce four QA datasets, with a specific focus on the MHQA Wikihop dataset.

2.2.1 The Stanford Question Answering Dataset

The Stanford Question Answering Dataset (SQuAD) [79, 78] was a popular QA task to benchmark QA models in NLP. Each of the 100,000+ data points contain a passage from a Wikipedia page, as well as a set of questions whose answers can be found in the passage [79]. Analysis of a sample of SQuAD questions found that 13.6% of SQuAD questions require multi-sentence/ multi-hop reasoning [79]. Thus, the majority of SQuAD questions are single-hop.

2.2.2 TriviaQA

The TriviaQA dataset was constructed by gathering 650,000 question-answer pairs, some of which were created by human trivia enthusiasts [52]. The context used to answer these questions is gathered using Information Retrieval methods [9, 52]. Some of the TriviaQA examples require cross sentence reasoning [52]; however, due to the information retrieval process, there are no strong guarantees of the nature of the reasoning required [112].

2.2.3 Wikihop Dataset

The QAngaroo Question Answering (QA) dataset[2] [106] is made up of Wikihop and Medhop datasets. The Wikihop dataset contains questions and answers related to general knowledge and is generated from Wikipedia data [106]. Medhop is specifically based off of medical knowledge and focuses on drug and protein interactions[106]. We primarily discuss and evaluate models on the Wikihop dataset. The QAngaroo datasets are both multiple choice QA tasks, which means that a set of possible answer candidates is provided, where only one candidate is the correct answer [106]. The QAngaroo datasets are comprised of data points, each of which is defined by a question q, a set of supporting documents S_q, a set of possible answer candidates C_q, and finally an answer to the question a_q. The task of a QA model evaluated on the Wikihop dataset is thus a multiclass classification problem, whereby the

[2] QAngaroo Dataset: https://qangaroo.cs.ucl.ac.uk/

model must predict which answer candidate is the correct answer, differentiating it from the incorrect answers.

The QAngaroo MHQA datasets construction procedure requires a pre-constructed set of documents D, and a knowledge base (KB) containing fact triplets defined as (s, r_{so}, o) for a subject entity s, object entity o and a relation r_{so} [106]. A question can be generated by leaving an entity out of a triplet as such: $q = (s, r_{so}, ?)$ where $a_q = o$. MHQA naturally involves the concept of a reasoning chain [15, 111], which is a series of fact triplets which link via common entities. A reasoning chain of length 3 would be defined by the triplets: $F_1 = (s, r_{so^1}, o^1)$, $F_2 = (o^1, r_{o^1o^2}, o^2)$, and $F_3 = (o^2, r_{o^2o^3}, o^3)$. Thus a reasoning chain of length 3 requires 2 'hops' to get from F_1 to F_3, and 1 hop to relate the question to this chain. Thus, one can define a 3-hop QA example as (s, r_{so^3}, o^3) for $q = (s, r_{so^3}, ?)$ where $a_q = o^3$.

When constructing the QAngaroo MHQA datasets, the documents found in D are matched up to all the entities found in the KB such that there is a link between a document from D and an entity, if the entity is mentioned in the document [106]. This mapping is many to many, since any entity could be found in multiple documents, and any document contains multiple distinct entities. This mapping naturally defines a graph G_D where nodes are made up of documents and entities, and whereby traversing the graph is similar to following hyperlinks found in documents [106]. This graph is illustrated in figure 2.1. To generate a single QAngaroo QA example, a fact triplet (s, r_{sa}, a_q) is selected from the KB. The graph G_D is then traversed via a breadth-first search using s as the starting point, and a_q as one of the many end points [106]. Answer candidates C_q are selected by filtering for all of the traversal endpoints which are type-consistent with a_q [106]. Two entities o^1 and o^2 are defined as type-consistent if there exists some type of relation r which is found to apply to both o^1 and o^2. This is to say that if there exists two fact triplets $F_1 = (*, r, o^1)$ and $F_2 = (*, r, o^2)$, then o^1 and o^2 are type-consistent with respect to relation r [106]. Ensuring that all candidate answers are type-consistent with respect to the query relation means that a model could not choose correct answers based on grammar and syntax-based information [106]. G_D is traversed from s up to a maximum depth of 3 document nodes [106]. Once answer candidate filtering has been done on this traversal, the set of document nodes which were collected along the traversal are then the supporting documents S_q used to answer the question. Only one of the paths of this traversal directly connects the subject s and the answer a_q, the rest of the paths lead to incorrect answer candidates. The documents gathered along the incorrect paths are thus irrelevant to the QA example and act as distractors to any QA model attempting to solve the task. Due to the traversal process, each answer candidate is guaranteed to be mentioned at least once in S_q.

From the process described thus far, there is a noteworthy issue. Since documents are selected by traversing G_D, and G_D is constructed via the co-occurrence of entities in documents, it is not guaranteed that the traversal paths correspond to logical reasoning chains. This is to say that given a fact triplet (s, r_{sa}, a_q), the traversal process selects documents S_q which connect s and a_q, however, it may not be logically deducible that (s, r_{sa}, a_q) is true given only these documents. This issue is related to the distant supervision assumption which states that if two entities are linked via a relation, that any piece of text which contains those entities may contain that same relation [97]. To investigate this and other issues, the creators of the QAngaroo datasets made use of human annotators to read a subset of the data points produced and classify attributes of each data point. The annotators found that 9% of the QA data points could be answered using only a single document, 26% of the data points contained ambiguity in which answer is correct, and 45% of questions could be logically answered from multiple of the given documents [106]. They also found that 12% of the questions could not be answered from the documents at all, meaning these documents

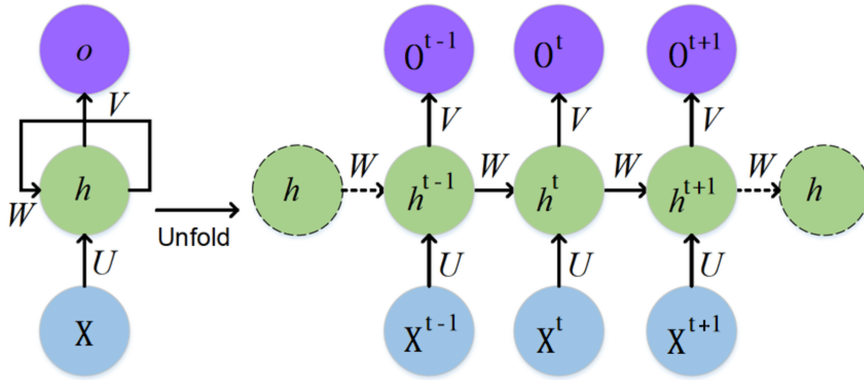

FIGURE 2.2: A diagram from [33] showing the general form of an RNN. Here x denotes input sequence elements, O denotes the corresponding output elements, and h denotes the state vectors, used to communicate between elements.

did not describe any valid reasoning chain capable of answering the question. If this sampling is assumed to be representative of the full dataset, it follows that 12% of the Wikihop questions are not answerable using reasoning [106], possibly resulting in a maximum score of 88%. For reference, human performance on Wikihop is 85% [106], and the current state-of-the-art model achieves 84.4% [41]. However, it should be noted that access to general knowledge may allow human or computer systems to surpass this limit by incorporating information not directly contained in the given context.

2.2.4 HotpotQA Dataset

Another prominent MHQA dataset is the HotpotQA dataset [112]. Similarly to Wikihop, it is constructed by traversing a graph of Wikipedia documents [112]. Unlike Wikihop however, no external KB is used, instead the HotpotQA construction process relies solely on Wikipedia hyperlinks [112]. Furthermore, HotpotQA dataset is not multiple choice, and instead requires that a model select a correct answer via span prediction. Despite being a span prediction task, the dataset is still constructed in a way that requires multihop reasoning to solve [112]. HotpotQA also contains a subset of yes/no questions [112], meaning the dataset is a multitask dataset, requiring a complex model output. HotpotQA was created in collaboration with Google, and due to the resource availability this brings, the dataset construction involved more human annotation than Wikihop. For example, alongside the answer to questions, each data point contains annotations indicating which sentences contain the information needed to answer the question [112], thus a model being trained on HotpotQA can use these annotations as an additional training signal. The current best exact match accuracy by a model on HotpotQA is 71.9%[3], compared to human performance of 83.6% [112].

2.3 Recurrent Neural Networks

Prior to the widespread adoption of attention-based sequence models, Recurrent Neural Network (RNN)-based [33, 83] models such as the LSTM [43] and GRU [21] were the de facto approach to NN-based sequence modeling. These models operate by encoding sequence elements one at a time, usually from left to right, and sometimes in both directions [43, 21]. When encoding a sequence element, RNNs create an encoding which is dependent

[3] HotpotQA leaderboard: `https://hotpotqa.github.io/`

on the features of the element, but also the encoding of the previous elements. They achieve this by maintaining a state vector or vectors which is updated and passed forward during the encoding of each element. Thus, separate elements in the sequence could exchange information via these state vectors, albeit only in one direction at a time, which gives them a strong sequential bias. This process is illustrated in Figure 2.2. In practice, these models performed poorly on long sequences, as the burden on the state vectors was too great to allow rich communication between distant elements [29]. Gating as used in the GRU and LSTM could reduce, but not eliminate this problem. Bi-directional RNNs were also common, allowing the RNN to read the sequence in both directions.

The RNN's limited ability to communicate information across long sequences results in a strong recency bias when encoding a sequence. This bias may have been responsible for its initial success in NLP [30], since text is principally a sequence of words, with many local word interactions. The sequential manner of encoding seemed to match the way a human might read text, i.e., from front to back, one word at a time. However, this strong recency bias also imposes certain limitations on the RNN, namely that it biases the model towards short-range dependencies [29, 30], contributing to the long sequence problem in RNNs. The sequential nature of RNNs also precluded the use of model parallelisation [101], since the encoding of each element depended on the encoding of the previous elements. Multihop Question Answering datasets present a challenge for RNNs [29]. This is because MHQA problems such as Wikihop require the integration of information which may be far apart in the text, or even in separate documents [106, 112, 52].

2.4 Transformers

The Transformer was first proposed as an encoder-decoder model for tasks such as machine translation or summarisation [101] (see Fig 2.3). Here, an encoder is a model which inputs data, and outputs a vector or matrix encoding of the data. Conversely, a decoder is a model which inputs an encoding, and outputs data. The Transformer is also used either as an encoder-only [27, 8, 114] or a decoder-only [77, 11] model. The transformer is also commonly used with a paradigm called pretraining and fine-tuning [27]. In this paradigm, the Transformer model is first pretrained on a large (unlabelled) text corpus using a self-supervised training task [27, 40, 37, 17]. Two common self-supervised training tasks in NLP are Language Modeling [77, 11], and Masked Language Modeling [27, 59, 114]. Language Modeling involves the prediction of the next element in a sequence, and thus can be used as a self-supervised training task by providing a model with the first few elements of a sequence and training it to complete the sequence [77, 11]. Masked Language Modeling involves masking out a randomly selected subset of elements in a sequence, and training the model to fill in these blanks [27]. Once a model has been pretrained on these or other self-supervised training tasks, the model can then be fine-tuned to perform a downstream NLP task such as question answering [114] or text classification [71]. The pretraining and fine-tuning paradigm has lead to large improvements in the state-of-the-art in many NLP tasks [27, 114, 11]. A particular advantage of this approach is that models are less reliant on large amounts of supervised training data than previous approaches [27]. The availability of open-source implementations of Transformer neural networks and publicly availability pretrained models[4] has further contributed to the wide-spread adaptation of Transformers in NLP.

The paradigm of pretraining and fine-tuning pre-dates the Transformer in the field of Computer Vision [107]. Convolutional Neural Networks (CNNs) have long been the dominant

[4] https://huggingface.co/

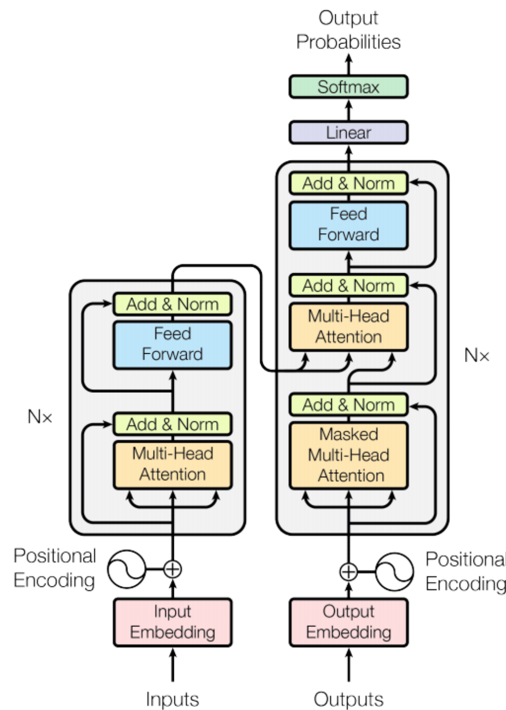

FIGURE 2.3: A figure from [101] of the encoder-decoder stack used in the Transformer model. Left is the encoder, making use of only self-attention. Right is the decoder which also makes use of cross-attention.

approach in computer vision. However, models based on the Transformer architecture have since been shown to improve the state of the art in tasks such as image classification (evaluated on the ImageNet benchmark) [115]. Transformers have also been used successfully in a number of other domains involving sequence modelling or structured data, including speech recognition [36], audio synthesis [28, 46], and protein structure prediction [80]. Transformer-like models can also be used as multi-modal models, for example encoding both text and images with the same network [49]. Finally, it has been shown that Reinforcement Learning (RL) can be reformulated as a sequence modelling problem, and promising results have been obtained using Transformers for both offline [16] and online [50] Reinforcement Learning problems.

2.4.1 Scaled Dot-Product Attention

At the core of the Transformer model is the multi-headed scaled dot-product self-attention mechanism [101]. Here we describe the scaled dot-product self-attention mechanism in the case of only one attention head, we then further describe how to perform multi-headed attention. The self-attention function involves computing a query, key and value vector for each of the inputted sequence elements. Queries and keys are used to compute the compatibility scores between element pairs, while value vectors are the representations which are passed forward through the model. The self-attention function outputs weighted sums of value vectors, where the weights are determined by normalised compatibility scores [101]. This process is illustrated in Figure 2.4 (left). Given a sequence of vectors stored in a matrix

$$X \in \mathbb{R}^{l \times f} = \text{Concat}_s(x_1, .., x_i, .., x_l), \tag{2.1}$$

where l is the length of the sequence, and f is the number of features of the vectors $x_i \in \mathbb{R}^f$ in the sequence. Concat_s represents concatenation along the sequence dimension. We formally describe the self-attention function where

$$Q = X\text{W}_\text{q}, \tag{2.2}$$

$$K = X\text{W}_\text{k}, \tag{2.3}$$

$$V = X\text{W}_\text{v}. \tag{2.4}$$

Here, $Q, K, V \in \mathbb{R}^{l \times f}$ are matrices of queries, keys and values respectively, and W_q, W_k, W_v $\in \mathbb{R}^{f \times f}$ are learned matrices used as linear transformations. In practice, the Transformer operates using batched matrix multiplications whereby the transformed states of all input elements are computed in parallel [101]. For our purposes however, it is useful to describe the algorithm from the point of view of a single input element $\boldsymbol{x_i} \in \mathbb{R}^f$. Formally, given $\boldsymbol{x_i} \mid i \in [1, l]$, we compute the scaled dot-product (SDP) between linear transformations of $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$ $\forall$ $j \in [1, l]$. This yields

$$\text{Dot}(\boldsymbol{x_i}, \boldsymbol{x_j}) = \frac{(\boldsymbol{x_i}\text{W}_\text{q})(\boldsymbol{x_j}\text{W}_\text{k})^T}{\sqrt{f}}, \tag{2.5}$$

where Dot(.) is a function which calculates the SDP between two vectors $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$. The values of these dot-product scores are scaled by a factor of $\frac{1}{\sqrt{f}}$ in order to improve model performance by preserving gradient signals [101].

Next, the softmax function normalises the dot-product scalars. This follows:

$$\alpha_T(\boldsymbol{x_i}, \boldsymbol{x_j}) = \text{softmax}_j(\text{Dot}(\boldsymbol{x_i}, \boldsymbol{x_j})), \tag{2.6}$$

where $\alpha_T(.)$ (read *Transformer*-style *Attention*) is a function which calculates the normalised compatibility score between two elements of a sequence of vectors. We refer to normalised compatibility scores as *attention scores*. The softmax_j function is defined as:

$$\text{softmax}_j(\text{Func}(\boldsymbol{x_i}, \boldsymbol{x_j})) = \frac{exp(\text{Func}(\boldsymbol{x_i}, \boldsymbol{x_j}))}{\sum_{k=1}^{l} exp(\text{Func}(\boldsymbol{x_i}, \boldsymbol{x_k}))}, \tag{2.7}$$

where Func is any binary function which computes the compatibility score of two input vectors. For SDP attention, Func would be the Dot(.) function, but the same softmax procedure is also used in Additive Attention [6]. Once we have our attention scores, we are ready to complete the self-attention step. Formally,

$$SelfAttention(\boldsymbol{x_i}) = \sum_{j=1}^{l} \alpha_T(\boldsymbol{x_i}, \boldsymbol{x_j})\boldsymbol{x_j}\text{W}_\text{v}. \tag{2.8}$$

Given these equations, we can see that the attention scores calculated by the function $\alpha_T(.)$ are responsible for routing information around the sequence. If $\alpha_T(\boldsymbol{x_i}, \boldsymbol{x_j})$ is large, then element j will have a large influence over element i, however if $\alpha_T(\boldsymbol{x_i}, \boldsymbol{x_j})$ is zero, then element j will have no influence on element i.

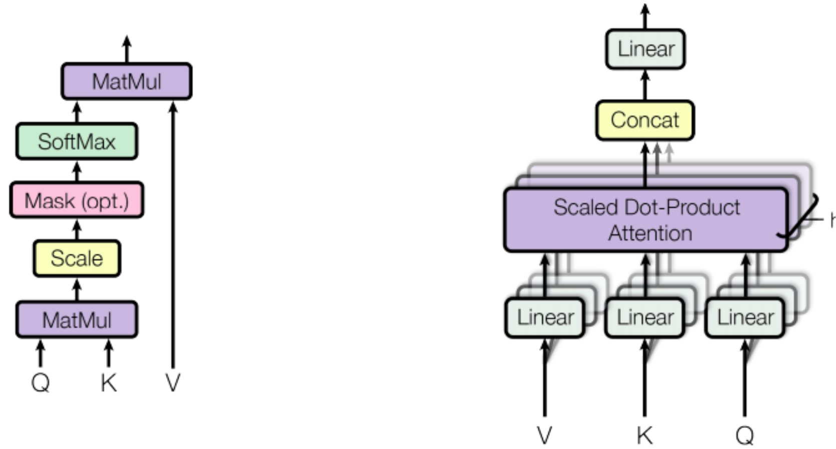

FIGURE 2.4: A figure from [101]. (Left) Shows how encoded states are generated from queries, keys and values using Scaled Dot-Product Attention. (Right) shows the entire Multi-headed attention function for h heads.

2.4.2 Multi-Headed Attention

Practically, the Transformer model computes multiple parallel channels of the self-attention mechanism described above, every layer. Each one of these channels is referred to as an attention head. For a Transformer model with m attention heads, m distinct Q^i, K^i, $V^i \;\; i \in [1, m]$ matrices are computed every layer [101]. This results in m distinct attention scores being calculated for each sequence element pair, and m separate weighted sums performed per element. Notably, the dimensionality of each query, key, and value vector is mapped down by a factor of m when performing multi-headed attention [101]. This keeps the computational requirements roughly the same as the number of heads increases, since each head operates at a lower dimensionality [101]. These m attention outputs are then concatenated together, and passed through a further linear transformation to bring the final output back to the model dimension f. More formally:

$$\text{MultiheadedSelfAttention}(X) = \text{Concat}_f(P^1, ..., P^m)\text{W}_\text{h}, \tag{2.9}$$

$$P^i = \text{SelfAttention}^i(X), \tag{2.10}$$

where Concat_f represents concatenation along the feature dimension. Here, each of the m SelfAttention^i functions are modified to reduce the dimensionality of the Q^i, K^i, V^i matrices to $\mathbb{R}^{l \times \frac{f}{m}}$. This is done by changing the dimensionality of weight matrices W_q^i, W_k^i, W_v^i to $\mathbb{R}^{f \times \frac{f}{m}}$. Concatenating all m $P^i \in \mathbb{R}^{l \times \frac{f}{m}}$ matrices results in a single $\mathbb{R}^{l \times f}$ matrix, which is then linearly transformed by the weight matrix W_h.

Making use of multiple attention heads serves to increase the expressiveness of the attention mechanism by allowing different heads to generate different compatibility scores for the same element pair, in the same layer [101]. This allows each attention head to potentially focus on different criteria when comparing sequence elements [23, 103].

2.4.3 Transformer Encoder

Despite first being proposed as part of an encoder-decoder model as shown in figure 2.3, the Transformer encoder has since been used in isolation by a number of models [27, 59,

114, 8, 55, 51, 25, 103]. The Transformer encoder is a sequence model, meaning that it inputs a sequence of vectors, and outputs a same-sized sequence of encoded vectors [27]. The Transformer decoder is generally used to generate new sequences, and can be used in combination with an encoder[101], or in isolation [77, 11]. Our work is only concerned with the Transformer encoder. The Transformer encoder is made up of a repeating stack of identical blocks, each of which contains a distinctly parameterised multi-headed SDP attention module [101], as well as a multilayer perceptron (MLP) [58] with a single hidden layer [101]. These blocks also make use of Layer Normalisation [5] and residual connections [96, 39] (see Fig 2.3). We will refer to the use of an MLP, residual connections and Layer Normalisations as the Transformer Update Function (TUF). Formally:

$$\text{TUF}(X_1, X_2) = \text{Norm}(X' + \text{MLP}_l(X')), \tag{2.11}$$

where

$$X' = \text{Norm}(X_1 + X_2). \tag{2.12}$$

Here $X \in \mathbb{R}^{l \times f}$ is a sequence of vectors stored in a matrix, l is the sequence length, and f is the number of features representing each sequence element. Norm is the LayerNorm [5] function, and MLP_l is an MLP with a single hidden layer containing a ReLU nonlinearity [2, 109]. LayerNorm [5] is an adaptation to the popular Batch Normalisation [47]. LayerNorm shifts the mean and standard deviation of its inputted matrix for each training example independently. This is in contrast to BatchNorm where the number of examples in a batch alters the behaviour of the normalisation [5]. LayerNorm thereby operates the same at test and train time, by ignoring the batch dimension. The original implementation of LayerNorm includes 2 trainable parameters, which are used to shift the mean and standard deviation of its input.

The Transformer encoder uses the TUF in combination with a self attention function [101]. It is worth noting that while the self attention function allows for rich communication between elements in the sequence, the TUF does not. This is because the MLPs in the TUF operate on each sequence element independently [101], and the LayerNorms allow very little information to be communicated across elements, specifically the mean and standard deviation of the sequence [101, 5].

The equation for the Transformer's MLP follows:

$$\text{MLP}_1(X) = \text{ReLU}(X\text{W}_l^1 + b_l^1)\text{W}_l^2 + b_l^2, \tag{2.13}$$

where $\text{W}_l^1, \text{W}_l^2$ are parameterised weight matrices used as linear transformations, b_l^1, b_l^2 are parameterised bias terms, and $\text{ReLU}(.)$ is simply the function $max(x, 0)$.

The full Transformer block equation for the encoder then simply follows:

$$X^{t+1} = \text{TUF}(X^t, \text{MultiheadedSelfAttention}(X^t)), \tag{2.14}$$

where MultiheadedSelfAttention is a multi-headed SDP attention function which we describe in section 2.4.2 above. This encoder block can be seen in figure 2.3, on the left of the decoder block. This notation is useful to us since we evaluate the TUF in combination with functions other than scaled dot-product attention.

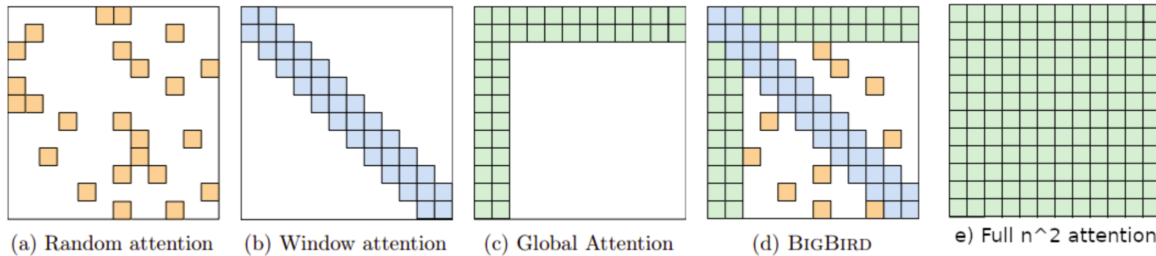

FIGURE 2.5: A figure from [114] showing a collection of sparse attention connection rules used in NLP (a-c), their composition (d), and a full attention map (e). Each row i and column j represents an element of the sequence, and each cell ij represents the interaction between elements i and j. Here a coloured block means that elements i and j can attend to each other, while an empty space means no communication.

2.4.4 Sparse Transformers

Full $O(n^2)$ self-attention mechanisms involve computing an attention score for every pair of the n sequence elements, for every attention head. Sparse transformers [114, 8, 18] aim to address this long sequence problem by avoiding the full $O(n^2)$ attention calculation, and thus improving how the compute requirements scale with the length of the sequence. We describe two prominent Sparse Transformer models, namely the Longformer [8], and the BigBird model [114]. These models use heuristic rules to decide which subset of token pairs should be attended with each other.

Models like BERT [27] are built on a full self-attention Transformer encoder [101], as such they suffer from the long sequence problem, resulting in a practical limit of around 512 input tokens. Sparse transformers like the Longformer and Bigbird can be pretrained in the same way as BERT, allowing for masked language modeling of sequences of up to 8192 tokens [114]. These models use carefully selected rules to connect tokens, resulting in a number of connections which is linearly proportional to the number of input context tokens [114, 8]. The configuring of these connection rules offers sparse transformers a trade-off between memory usage and modeling capacity. All else equal, the more sparse a Transformer model is, the lower its modeling capacity would be [114], and the better its compute requirements would scale with sequence length [8, 114].

As shown in Figure 2.5 (b), these models connect tokens within a sliding window, ensuring nearby tokens are always connected, and able to directly communicate. Here, a sliding window simply means that every token is connected to a fixed number of tokens ahead and behind in the sequence. This sliding window rule imposes locality on the model, which is beneficial to language models due to the importance of nearby words [114, 8]. The number of connections made by the sliding window rule is roughly $w \times n$ for a window size of w. The BigBird model adds a second connection rule which connects each token to a fixed number of randomly selected tokens [114] (Fig 2.5 (a)). This was done to decrease the average degrees of separation between tokens. Furthermore, randomly connected graphs are able to approximate fully connected graphs [114]. The number of these random connections is $c \times n$ for c random connections per token.

Finally, both Bigbird and the Longformer provide global attention (Fig 2.5 (c)) to a subset of the token sequence [114, 8]. This is to say that g tokens are chosen to be global, and are thus connected to every other token. The tokens which are chosen to be global are decided upon

based off of the task that the Sparse Transformer is performing. For classification tasks, these models provide global attention to the "[CLS]" token, from which the classification output is read [8, 114]. For Question Answering (QA), these models provide global attention to all query tokens [8, 114]. And for the multiple choice QA such as Wikihop, these models provide global attention to all query and candidate tokens [8, 114]. In any case, the number of connections added by the global rule is $g \times n$ for an upper limit of g global tokens. Overall, the number of connections made in the Longformer model is $(w + g) \times n$, and $(w + g + c) \times n$ for Bigbird. Since w, g, and c are all constants, the total number of connections is linearly proportional to n, which leads to the linear memory complexity of these models of $O(n)$.

Sparse Transformers are an appealing solution to the long sequence problem in Transformers. It has however been demonstrated theoretically that Sparse Transformers with $O(n)$ edges require a factor of n times as many attention layers to recover the full mathematical expressivity of a fully connected self attention mechanism [114]. These extra layers which are required thus offset the memory complexity gains given by the Sparse Transformer. In practice however, the full theoretical expressivity of the self attention mechanism may not be required for every task, and at every layer. Thus the Longformer and Bigbird were both able to outperform full self attention models on a range of NLP tasks [8, 114], including the Wikihop dataset. However it should be noted that Sparse Transformers cannot be considered a complete solution to the long sequence problem in Transformers, since for any fixed value of $(w + g + c)$, the model sparsity will grow with the number of sequence elements, possibly reducing modeling capacity for longer sequences.

2.4.5 Relative Positional Embeddings

The multi-headed scaled dot product attention as described in the original Transformer paper [101] does not take the position of tokens into account during computation. This is to say that the multi-headed attention module is permutation invariant [101]. In natural language however, the order of words carries some important information [30, 76, 73]. In order to incorporate word and token positions into the Transformer model, it is common to include absolute positional embeddings (APE) [101, 114, 27]. APEs map each ordinal position in the input sequence to a distinct vector. These APEs are summed with the initial token embeddings before they are encoded by the model, thus imbuing the token embeddings with positional information [101]. Various methods are used to generate these positional embeddings such as using sin waves of various wavelengths [101], or just learning each positional vector during the training process [27, 101].

Relative positional embeddings (RPE) work by utilising the relative difference in position of token pairs [92]. In contrast to absolute positional embeddings, RPEs operate inside of every multi-headed attention layer [92], instead of being added at the beginning of an encoder stack [101]. The RPE module replaces the standard scaled dot product attention [101] mechanism with a modified version. This is such that the relative ordinal position of every sequence element pair is included into the attention mechanism itself [92]. This means that instead of positional information being maintained in the sequence vectors throughout the model's layers [101], the RPE attention mechanism itself is made to incorporate this positional information during every layer. RPEs involve the usage of element pair information. In graph terminology, element/node pair information would be labelled as edge features.

Here we describe the original modifications to the standard scaled dot product attention mechanism proposed by [92]. To include relative positional information in the self attention function, equations 2.5 and 2.8 are modified to include learned RPE vectors. Formally equation 2.5 becomes:

$$\text{RelDot}(\boldsymbol{x_i}, \boldsymbol{x_j}, K) = \frac{(\boldsymbol{x_i}\text{W}_\text{q})(\boldsymbol{x_j}\text{W}_\text{k} + K_{ij})^T}{\sqrt{f}}, \tag{2.15}$$

resulting in a new equation for calculating attention scores which follows:

$$\alpha_r(\boldsymbol{x_i}, \boldsymbol{x_j}, K) = \frac{exp(\text{RelDot}(\boldsymbol{x_i}, \boldsymbol{x_j}, K_{ij}))}{\sum_{k=1}^{l} exp(\text{RelDot}(\boldsymbol{x_i}, \boldsymbol{x_k}, K_{ik}))}. \tag{2.16}$$

Here, α_r (read *relational Attention*), is a Transformer-style SDP attention function which incorporates an edge feature matrix K. Next, equation 2.8 becomes

$$\text{RelSelfAttention}(\boldsymbol{x_i}, V, K) = \sum_{j=1}^{l} \alpha_r(\boldsymbol{x_i}, \boldsymbol{x_j}, K)\boldsymbol{x_j}\text{W}_\text{v} + V_{ij}, \tag{2.17}$$

where K $\in \mathbb{R}^{n \times n \times f}$ for keys and $V_{ij} \in \mathbb{R}^{n \times n \times f}$ for values, are both learned embedding matrices. Each cell $\text{K}_{ij} \in \mathbb{R}^f$ or $V_{ij} \in \mathbb{R}^f$ in these matrices represents a learned embedding vector. When using relative positional embeddings, the cells will have distinct values depending on the relative ordinal position of i and j. This is to say that

$$K_{ij} = K_{ab} \quad \forall \quad Abs(i-j) = Abs(a-b). \tag{2.18}$$

From the equations, we can see that these two modifications (K and V) are independent, and as such they can be used in combination. Alternatively, either one can be used without the other.

The authors of the first RPE Transformer paper [92] performed ablation tests for both modifications, namely the addition of K_{ij} and V_{ij}, which we refer to as Key Type Embeddings (K-Type) and Value Type Embeddings (V-Type) respectively. From the equations, we see that K-Type embeddings influence the attention coefficients, while the V-Types directly influence the value vectors which are passed forward through the Transformer. Their ablations found no significant improvement from including V-Type embeddings, however K-Type embeddings improved model performance [92]. It is also worth noting that the RPE algorithm introduces a memory and speed overhead to the self-attention algorithm [92, 46]. A second RPE algorithm [46] was introduced in order to overcome the memory constraints introduced by the original RPE algorithm. The authors describe a modification to the RPE algorithm which reduces the quadratic memory complexity of RPEs to linear complexity [46]. The modification exploits the fact that the relative ordinal positions between element pairs contains implicit structure. Specifically, for element pairs i and j, the relative position difference increases by 1 for every increase in j. Beyond this modification, the authors elect to remove the usage of V-Type embeddings in equation 2.17 [46], following the ablations performed by [92].

Relative Positional Embeddings are of interest to us due to their ability to inject edge information into the self-attention mechanism [92]. For the canonical usage of the RPE algorithm, the edge information contained in every element pair is simply the pairs relative ordinal position. However, the original RPE algorithm is general enough that any arbitrary edge information could be used [92]. Since Absolute Positional Embeddings produce a unique vector per input element, they are fundamentally incapable of modeling edge information. RPEs however offer the ability to include unique vectors for every element pair, making them ideal for modeling edge information. It is worth noting that the RPE

modification offered by [46] is not applicable when using the algorithm for general edge information. This is because the modification exploits the structure contained in the relative ordinal positions of elements, and no such structure exists for the general case of a graph with edges of varying types.

2.5 Graph Neural Networks

A graph is a highly general data structure, with most other data structures being describable as graphs. A sequence can be represented by a chain of connected nodes, a set could be thought of as a group of unconnected or fully connected nodes, an image could be considered a 2-dimensional mesh of nodes with adjacent pixels being connected. Beyond this, graphs can easily represent non-euclidean data such as knowledge bases[108]. It is thus desirable to be able to model graphs with modern Deep Neural Networks (DNN). This area is sometimes referred to as *geometric deep learning* [108], and any DNN capable of modeling graph data is referred to as a Graph Neural Network (GNN). Following the success of the Convolutional Neural Network in computer vision, early approaches to GNNs attempted to generalise the 2-dimensional euclidean convolution operator to the general graph case [108]. From this work two approaches emerged, namely spectral and spatial GNNs. Spectral GNNs [12, 54] were inspired by the mathematical study of graphs called the spectral graph theory [20]. They suffer from two key issues, namely they are computationally expensive [108], and crucially, they fail to generalise to unseen graph structures [108], making them incapable of solving many practical graph modeling tasks. Spatial GNNs involve the concept of *message passing* [85, 69], whereby nodes send each other messages along edges. A comprehensive review of the development of GNNs shows the trend for GNN research to increasingly focus on Spatial GNNs [108].

GNNs have been used in various ways in NLP. Our work focuses on GNNs which encode entity graphs [26, 99, 95, 31]. However, there has also been work on using GNNs to encode Abstract Meaning Representation (AMR) graphs [104]. Various works have also used GNNs to encode Knowledge Graphs (KG), for example to select relevant context passages [113] for downstream QA tasks, or to generate text from the KG [56]. It is also interesting to note that GNN encoding does not preclude any decoding/output strategies. Thus, GNN encoding can be followed by node-level classification [26, 99], graph-level classification such as span prediction [31], or even generative outputs [56].

In this section, we formally describe message passing spatial GNNs, introducing a simple notation for the message passing paradigm, and use it to describe a few key GNN architectures. We also describe some key works in using GNNs in NLP, specifically to address the problem of MHQA.

A graph G is made up of a set of nodes and a set of edges. Each node has a corresponding set of features stored in a vector $\boldsymbol{h} \in \mathbb{R}^f$, called the node's state. Edges may or may not have features. GNNs encode graphs by inputting both a set of node states stored in a matrix $H \in \mathbb{R}^{n \times f} = \text{Concat}_s(\boldsymbol{h}_1, .., \boldsymbol{h}_n)$ and an adjacency-matrix $E \in \mathbb{Z}^{n \times n}$, where n is the number of nodes, and f is the number of features per node.

For a graph which has edges of differing types, we first define the set of e unique edge types as $T = \{r_1 ... r_e\}$. We then generalise the adjacency matrix E to account for both the edge connectivity, as well as the edge types. We instead refer to this generalised matrix as the *edge matrix* E such that $E_{ij} = 0$ implies that nodes i and j are unconnected, while $E_{ij} = c$ implies that nodes i and j are connected via edge type r_c. Thus, a graph without edge types is simply a graph where $e = 1$, returning the matrix E to a standard adjacency matrix. The

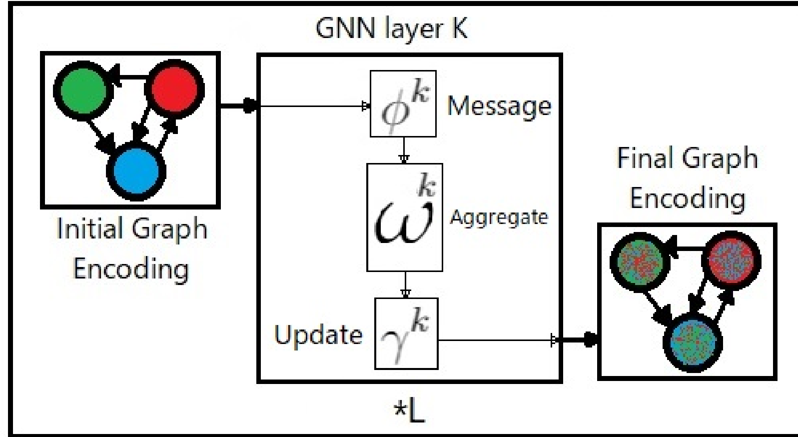

FIGURE 2.6: Shows the process of encoding a graph via a message passing GNN made up of L layers. Each layer k is composed of a message function ϕ, an aggregate function ω, and an update function γ.

number of edges in the graph G is represented by $|E|$, which is the number of elements where $E_{ij} > 0$. The edge matrix thus defines the connectivity of the graph G, as well as the edge types if they are present. E_{ij} is a single cell in the edge matrix and represents a single edge, which connects two nodes. These edges are directed, thus edge E_{ij} emanates from node j, which we call the *out-node*, and points to node i, which we call the *in-node*. To rephrase, edges point from out-node j to in-node i.

A GNN with L layers will output an encoded set of node states $H^L \in \mathbb{R}^{n \times f}$. The output of a GNN is dependent on both the node feature matrix H, as well as the edge matrix E. More formally, given initial node states $H^1 \in \mathbb{R}^{n \times f}$ and an edge matrix E, a GNN with L layers will calculate encoded node states H^L via:

$$H^L = \text{GNN}^{*L}(H^1, E), \tag{2.19}$$

where GNN^{*L} is L GNN layers stacked on top of each other.

2.5.1 Message Passing Graph Neural Networks

A message passing GNN (MP-GNN) [35] is a GNN which conceptually sends messages along edges between its nodes. These messages are then collected for each receiving in-node and used to update the node states. MP-GNNs make use of operations such as mean or sum that are able to accept varying numbers of inputs [108]. Using these operations allows message passing GNNs to aggregate neighbour node messages regardless of the number of neighbour nodes. This feature is the key to allowing MP-GNNs to generalise to all possible graph topologies. An MP-GNN is made up of multiple GNN layers, where each layer performs message passing between all connected nodes once (see Alg 1). Formally:

$$H^{k+1} = \text{GNN}^k(H^k, E), \tag{2.20}$$

where GNN^k is the kth GNN layer, $H^k \in \mathbb{R}^{n \times f}$ is the intermediate set of node states inputted to layer k, and H^{k+1} is the corresponding encoded set of node states outputted by GNN^k. Typically, GNN layers share parameters [99, 31], in which case they are referred to as recurrent GNNs[108], however they may also have distinct parameters per layer. This process is

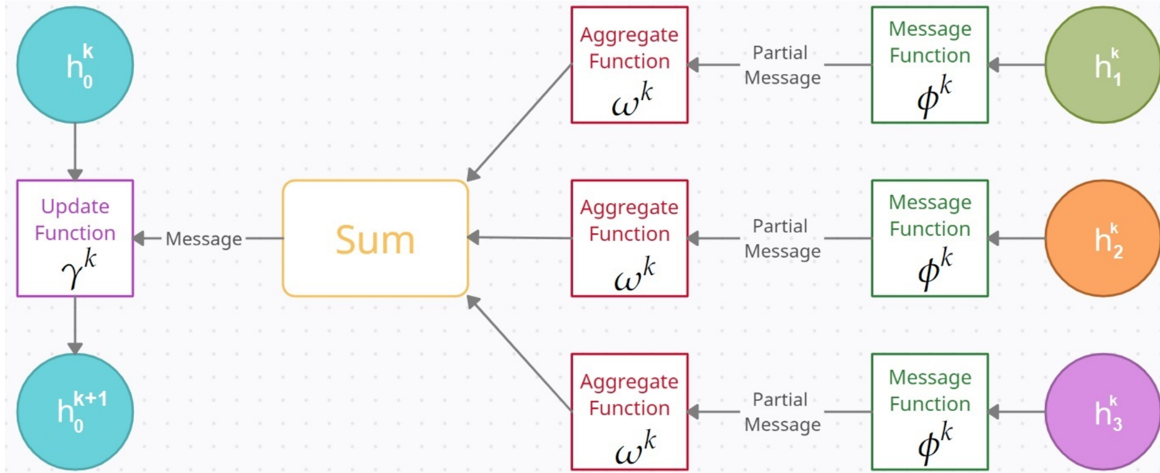

FIGURE 2.7: A visualisation of message passing being performed once to update the state of node h_0 via messages from its neighbour nodes $N(0) = \{1, 2, 3\}$. Not shown here is the fact that the message and aggregate function have access to the receiving nodes current state h_0^k

visually represented by figure 2.6.

The message passing pattern [35] is a general framework which describes how MP-GNNs update their node states. It assumes a directed graph, however undirected graphs can be emulated by replacing each undirected edge with two oppositely directed edges with the same edge features. Each directed edge has an out-node, from which the edge emanates, and an in-node, to which the edge points. Formally, given edge E_{ij}, the in-node is node i while the out-node is node j. The message passing pattern follows:

1. **Message**. For every edge in the given graph, the node state of the edge's out-node is transformed into a partial message vector via a learned transformation, usually a simple MLP [99, 65], or a parameterised linear transformation [102]. This transformation can optionally incorporate edge features [99, 26]. The trivial message function simply returns the out-node state without transforming it.

2. **Aggregate**. A variable number of partial-message vectors inbound for each in-node are aggregated into a single, fixed-size message vector. A simple example is the mean function. The aggregate step is the step which allows for information to be communicated between nodes by combining the messages of many neighbour nodes into a single message per in-node.

3. **Update**. Each in-node's state is updated as a function of its current state, as well as the aggregated message vector from its neighbours. A common trivial update function just returns the aggregated message vector, without incorporating the current state [38, 102]. Various forms of gating [96] can also be computed in the update step [99, 95].

We use the following notation to describe the general form of the message passing pattern, from the perspective of a single node state ($\boldsymbol{h}_i \in \mathbb{R}^f$) being updated once, by the kth GNN layer:

Message:

$$M_{ij}^k = \phi^k(\boldsymbol{h}_i^k, \boldsymbol{h}_j^k, E_{ij}) \tag{2.21}$$

Algorithm 1 Message Passing GNN Node State Update Procedure

Require:
Initial node features $\{\boldsymbol{h}_1^1, .., \boldsymbol{h}_n^1\}$
Edge matrix $E \in \mathbb{Z}^{n \times n}$

1: **for all** $k \in \{1, .., L\}$ **do** ▷ For each GNN layer
2: **for all** $i \in \{1, .., n\}$ **do** ▷ For each in-node
3: $A_i^k \leftarrow Zeros(f) \in \mathbb{R}^f$
4: **for all** $j \in N(i)$ **do** ▷ For each out-node pointing to node i
5: $M_{ij}^k \leftarrow \phi^k(\boldsymbol{h}_i^k, \boldsymbol{h}_j^k, E_{ij})$ ▷ Message
6: $A_i^k \leftarrow A_i^k + \omega^k(M_{ij}^k, \boldsymbol{h}_i^k, \boldsymbol{h}_j^k)$ ▷ Aggregate
7: **end for**
8: $\boldsymbol{h}_i^{k+1} = \gamma^k(\boldsymbol{h}_i^k, A_i^k)$ ▷ Update
9: **end for**
10: **end for**
11: **return** $\{\boldsymbol{h}_1^L, .., \boldsymbol{h}_n^L\}$ ▷ Final Encoded Node States

Aggregate:

$$A_i^k = \sum_{j \epsilon N(i)} \omega^k(M_{ij}^k, \boldsymbol{h}_i^k, \boldsymbol{h}_j^k) \tag{2.22}$$

Update:

$$\boldsymbol{h}_i^{k+1} = \gamma^k(\boldsymbol{h}_i^k, A_i^k) \tag{2.23}$$

Where ϕ^k is the message function at layer k of the GNN, and $M_{ij}^k \in \mathbb{R}^f$ is its output which represents the partial message vector being passed to node i from node j along a single edge E_{ij}. $N(i)$ is the set of neighbour node indices for node i, and is defined entirely by the edge matrix $E \in \mathbb{Z}^{n \times n}$. ω^k is the aggregate function which prepares the message M_{ij}^k to be summed with all messages bound for node i. This summing forms a single aggregated message $A_i^k \in \mathbb{R}^f$ to be passed to node i, which represents information from all of node i's neighbours. γ^k is the update function, which combines the aggregated message A_i^k with the nodes current state $\boldsymbol{h}_i^k$ to create the nodes next state $\boldsymbol{h}_i^{k+1} \in \mathbb{R}^f$. This process is shown in figure 2.7.

We detail the usage of the message, aggregate and update function in algorithm 1, and show how the final node states $H^L \in \mathbb{R}^{n \times f}$ are calculated from the initial node states $H^1 \in \mathbb{R}^{n \times f}$.

2.5.2 GNN architectures

Here we describe three notable MP-GNN architectures using this message passing notation, namely GraphSAGE [38], Relational Graph Convolutional Network (R-GCN) [86], and the Graph Attention Neural Network (GAT) [102]. We also describe the Transformer encoder model using the message passing notation to demonstrate that the Transformer can be easily described as a GNN, and to further familiarise the reader with the notation. Finally we introduce the concept of *in-out symmetry*, and classify each catalogued GNN as either *in-out symmetric* or *in-out asymmetric*.

A GNN is in-out symmetric if it treats in-node and out-node states identically. This is to say that when updating an in-node i, the nodes own state h_i is not given special treatment over any of the neighbouring out-node states h_j. An in-out symmetric GNN thus treats a nodes

current state as just another message from its neighbours $i \in N(i)$. Conversely, an in-out asymmetric GNN will differentiate a nodes current state from its neighbours messages.

2.5.2.1 GraphSAGE

GraphSAGE (SAmple and aggreGatE) [38] is not a single specific model architecture, and instead is a general framework which supports message passing between n^{th} degree node neighbours. However in practice, implementations of this algorithm[5] simply use first degree neighbours, and thus conform to the standard message passing pattern. The word 'Sample' refers to the fact that GraphSAGE GNNs may sample node neighbours instead of aggregating messages from all node neighbours every GNN layer [38, 108]. However, this feature is only required for very large graphs which contain orders of magnitudes more nodes [108] than we are concerned with in this work. Again, implementations of GraphSAGE may simply ignore the sampling and conform rather to the standard message passing pattern.

GraphSAGE describes multiple methods of aggregating neighbour message vectors, all of which are able to support a variable number of neighbours [38]. The simplest of which, and the one we use as a base of comparison, just returns the mean of all the messages. This is the method which is popular in implementations, and is most similar to purpose-built GNNs used in the GNN-based MHQA models discussed in this work (see Sec 2.6.3). Another interesting aggregation function it proposes [38] is using an LSTM [43] to aggregate messages. The authors note that since LSTMs implicitly introduce a sequential bias, and are thus not symmetric about permutations, they must randomly reorder neighbour nodes before every aggregation [38].

In the notation of the message passing pattern GraphSAGE with mean aggregation, single degree node neighbours, and no neighbour sampling is as follows:

Message:

$$\phi^k(\boldsymbol{h}_i^k, \boldsymbol{h}_j^k, E_{ij}) = \boldsymbol{h}_j^k \tag{2.24}$$

Aggregate:

$$\omega^k(M_{ij}^k, \boldsymbol{h}_i, \boldsymbol{h}_j) = \frac{M_{ij}^k}{|N(i)|} \tag{2.25}$$

Update:

$$\gamma^k(\boldsymbol{h}_i^k, A_i^k) = \sigma(\text{Concat}_f(\boldsymbol{h}_i^k, A_i^k)\text{W}_s) \tag{2.26}$$

These equations describe a GNN with the trivial message function and mean aggregation function. The update function concatenates an in-node's state with the message vector from all its neighbouring out-nodes, before applying a learned linear transformation W_s, and a nonlinear activation function σ. Concat_f represents concatenation along the feature dimension.

From the GraphSAGE update function above, we can see that the in-node state $\boldsymbol{h}_i^k$ is concatenated with the neighbour nodes message A_i^k. Thus GraphSAGE is *in-out asymmetric* since the in-node state is not simply considered a message, but is instead treated differently to neighbour node messages $\boldsymbol{h}_j^k$.

[5] GraphSAGE implementation: https://pytorch-geometric.readthedocs.io/en/1.4.3/_modules/torch_geometric/nn/conv/sage_conv.html

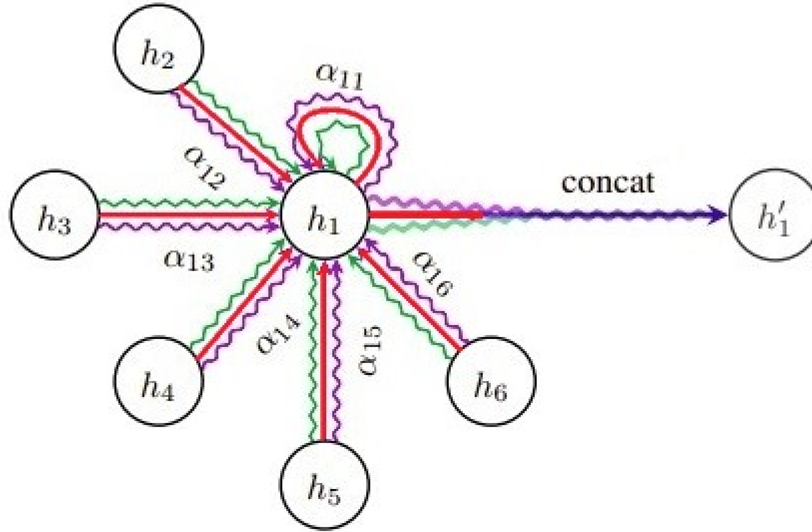

FIGURE 2.8: A figure from [102]. The node state update process for node_1, using 3 attention heads (green, red, magenta). Coefficients α_{ij} are generated for each (node_1, neighbour) pair, and for each head, using the MLP-based additive attention mechanism [102] (See Eq 2.27), and a softmax (See Eq 2.6). These coefficients are such that for each head, the coefficients wrt node_1 sum to 1. The new node_1 state is then generated via coefficient-weighted sums of neighbour states, finally being combined from the 3 heads.

2.5.2.2 *Graph Attention Neural Network*

The Graph Attention Neural Network (GAT) [102] was introduced around the same time as GraphSAGE. Similarly to GraphSAGE, it follows the message passing pattern, and thus makes use of an aggregation function capable of operating on a variable number of node neighbours. Specifically, it makes use of a self-attention mechanism to perform learned weighted sums over node messages [102]. Thus, it is able to give more weight to more important neighbours in a learnable way [102]. The GAT makes use of a different type of attention than does the Transformer. Specifically, the GAT makes use of *Additive Attention* [6], whereas the Transformer uses scaled dot-product (SDP) attention [101]. More formally:

$$\text{Att}(\boldsymbol{h}_i, \boldsymbol{h}_j) = \psi(\text{Concat}_f(\text{W}_v\boldsymbol{h}_i, \text{W}_v\boldsymbol{h}_j)\text{W}_a) \tag{2.27}$$

$$\alpha_G(\boldsymbol{h}_i, \boldsymbol{h}_j) = \text{softmax}_j(\text{Att}(\boldsymbol{h}_i, \boldsymbol{h}_j)). \tag{2.28}$$

Here Att(.) is an MLP-based compatibility scoring function used in additive attention [6]. Here ψ is the LeakyReLU [2] nonlinear activation function. W_a, and W_v, are model weight parameters. $\alpha_G(.)$ (read *GAT*-style *Attention*) is a function which calculates the attention scores between nodes i and j. As in SDP attention, softmax_j (see Eq 2.7) is applied to compatibility scores to produce the normalised attention scores [102, 101].

What follows are the message passing equations for the GAT in the case of a single attention head:

Message:

$$\phi^k(\boldsymbol{h}_i^k, \boldsymbol{h}_j^k, E_{ij}) = \text{W}_\text{v}\boldsymbol{h}_j \tag{2.29}$$

Aggregate:

$$\omega^k(M_{ij}^k, \boldsymbol{h}_i, \boldsymbol{h}_j) = \alpha_G(\boldsymbol{h}_i, \boldsymbol{h}_j) M_{ij}^k \tag{2.30}$$

Update:

$$\gamma^k(\boldsymbol{h}_i^k, A_i^k) = \sigma(A_i^k) \tag{2.31}$$

Figure 2.8 offers a high level view of the node state update process for the GAT for a single node with multiple neighbours. This process is much the same as in the Transformer, whereby $\alpha(.)$ is responsible for routing information between nodes. The main difference between additive attention and a SDP attention is how $\alpha(.)$ is calculated (Dot from Eq 2.5 vs Att from Eq 2.27). There are differences between the GAT and Transformer beyond this, such as the use of the TUF in the Transformer [101].

From the GAT equations, we can see that there is no distinction between an in-node and an out-node in the update function, meaning the GAT is in-out symmetric. This then necessitates the use of self edges in the GAT, meaning it must be so that $i \in N(i)$. If this were not the case, then a nodes current state would not factor into its subsequent state at all.

To boost performance, like in the Transformer literature, the GAT makes use of multi-headed attention. Practically this amounts to performing the above attention mechanism m times per layer, with m distinctly parameterised functions, for each of the m heads. These m outputs are then concatenated along the feature dimension and mapped back down to the dimensionality of the model. This is distinct from how multi-headed attention is performed in the Transformer literature, whereby the dimensionality of the attention mechanism is first mapped down by a factor of m, thus returning to the model dimension after the head outputs are concatenated. Crucially, this means that computational requirements scale linearly with the number of heads in the GAT [102], but not in the Transformer [101].

Given single-headed GAT layers which computes the full message passing equations above such that

$$H' = \text{GAT}^s(H^t, E), \tag{2.32}$$

where GAT^s is the sth GAT head such that $s \in \{1, .., m\}$, $H \in \mathbb{R}^{n \times f}$ is a matrix containing all node state vectors, and $H' \in \mathbb{R}^{n \times f}$ is the same-sized intermediate output of the single head. The multi-headed GAT then calculates updated node states as follows:

$$H^{t+1} = (\mathop{||}_{s=1}^{m} \text{GAT}^s(H^t, E)) W_m, \tag{2.33}$$

where $||$ represents concatenation along the feature dimension over all head outputs. $\mathrm{W}_m \in \mathbb{R}^{mf \times f}$ is a learnable matrix which maps the feature dimension back down to the model dimension f.

2.5.2.3 *Transformer as a GNN*

The Transformer model can be described using the message passing notation as follows:

Message:

$$\phi^k(\boldsymbol{h}_i^k, \boldsymbol{h}_j^k, E_{ij}) = \mathrm{W_v} \boldsymbol{h}_j \tag{2.34}$$

Aggregate:

$$\omega^k(M_{ij}^k, \boldsymbol{h}_i, \boldsymbol{h}_j) = \alpha_T(\boldsymbol{h}_i, \boldsymbol{h}_j) M_{ij}^k \tag{2.35}$$

Update:

$$\gamma^k(\boldsymbol{h}_i^k, A_i^k) = \text{TUF}(\boldsymbol{h}_i^k, A_i^k) \tag{2.36}$$

Here W_v, α_T, and the TUF are defined in section 2.4. In this notation, it is clear that the GAT and the Transformer are indeed very similar, differing only in the way attention scores are calculated, and the usage of the TUF in the update function. Notably, the only thing which distinguishes a Sparse Transformer from its full self-attention counterpart is the edge matrix E, that defines which nodes are connected. The traditional full self-attention Transformer [101] operates on fully connected graphs where $E_{ij} = 1 \; \forall \; i,j \in \{1,..,n\}$, whereas Sparse Transformers use a heuristic-based graph structuring algorithm to decide which nodes are connected.

The use of the argument $\boldsymbol{h}_i^k$ in the TUF here ensures that the Transformer is in-out asymmetric. This is due to the use of residual connections in the TUF, which privileges a nodes current state over other nodes current state when calculating its next state. The self-attention mechanism itself is in fact in-out symmetric (see Eq 2.8), thus it is the TUF alone which ensures that the Transformer is in-out asymmetric. From these observations, and due to the success of the Transformer model, a natural question to ask is can the GNN-based MHQA models benefit from SDP attention and the TUF.

2.5.2.4 *Gating*

Typical Deep Neural Networks (DNN) are built using Multilayer Perceptrons (MLP) [58] as a basic unit of computation [93]. MLPs work by performing weighted sums over neuron activations, where the weights are the learnable parameters of the network [58]. Gating in Neural Networks can be defined as any operation whereby the activations of at least two neurons are directly multiplied together [108]. This can increase the expressiveness of a DNN [68]. Despite the fact that DNNs without gating are universal function approximators [61], and as such can learn to approximate a multiplication operation, it has been shown that the number of neurons required to perform this operation would grow exponentially with the required precision [68].

Gating in Neural Networks takes many forms, such as LSTM [43] or GRU [21] style gating in the RNN literature, or the scalar gating used to introduce soft inductive biases into models [23, 3]. Gating is also used in various ways in GNNs [108, 31, 95, 65]. GNN-based MHQA models commonly use gating in their update function [26, 99, 95, 13]. Using a gating mechanism in a GNN's update function allows for deeper GNNs, and prevents node states from becoming homogeneous [99]. We describe a common gating function used in the GNN-based MHQA literature as follows:

$$\text{Gate}(\boldsymbol{h}^k, A^k) = \tanh(\boldsymbol{u}^k) \odot \boldsymbol{g}^k + \boldsymbol{h}^k \odot (1 - \boldsymbol{g}^k), \tag{2.37}$$

where

$$\boldsymbol{u}^k = \boldsymbol{h}^k \text{W}_\text{u} + A^k, \tag{2.38}$$

$$\boldsymbol{g}^k = \sigma(\text{Concat}_f(\boldsymbol{h}^k, \boldsymbol{u}^k)\text{W}_\text{g}). \tag{2.39}$$

Here Gate is the gating function, σ denotes the sigmoid function, $\odot$ denotes element-wise multiplication. $\text{W}_\text{g} \in \mathbb{R}^{2f \times f}$ and $\text{W}_\text{u} \in \mathbb{R}^{f \times f}$ are learned matrices. Incorporating gating as shown above into a GNN effectively makes the GNN into a highway network [96], offering some additional theoretical backing to the claim that gating allows for deeper GNNs [96]. Using this form of gating in the update function of a GNN will ensure that the GNN is in-out asymmetric.

2.5.2.5 *Relational GNNs*

Graphs such as knowledge graphs [44] contain edges of multiple types. It is thus useful for GNNs to be able to model edge types when encoding graphs. A relational GNN (RGNN) exploits edge type labels during its message passing step [86]. Current RGNN models involve the usage of distinct weight matrices for each unique edge type [86, 45, 63]. This in effect makes these GNNs similar to a Mixture Of Experts [67, 32] model. A notable relational GNN is the Relational Graph Convolutional Network (R-GCN) [86] which we describe here using the message passing notation.

Message:

$$\phi(\boldsymbol{h}_i^k, \boldsymbol{h}_j^k, E_{ij}) = \frac{\mathrm{W}^{\mathrm{r}}\boldsymbol{h}_j^k}{|E_i^r|} \tag{2.40}$$

Aggregate:

$$\omega(M_{ij}^k, \boldsymbol{h}_i, \boldsymbol{h}_j) = \frac{M_{ij}^k}{|N(i)|} \tag{2.41}$$

Update:

$$\gamma(\boldsymbol{h}_i^k, A_i^k) = \sigma(A_i^k + \mathrm{W}_0\boldsymbol{h}_i^k) \tag{2.42}$$

Here $\mathrm{W}_0 \in \mathbb{R}^{f\times f}$ is a learned weight matrix, and $\mathrm{W}^{\mathrm{r}} \in \mathbb{R}^{f\times f}$ refers to a weight matrix which depends on the edge type $r = E_{ij} \in T$. $|E_i^r|$ is the number of edges of type r which point to node i. Here, the number of distinct weight matrices used in the message function is equal to the number of unique edge types $|T|$. From the update equation, we can see that the R-GCN is in-out asymmetric.

There is a variation of the GAT proposed for use in a GNN-based MHQA model [31] which extends the GAT to be a relational GNN. Like the R-GCN, this relational GAT makes use of a distinct weight matrix W^r for different edge types (see Eq 2.60). We refer the strategy of using distinct weight matrices for different types as *parameter switching*, and refer to any GNN which switches parameters based on edge types as a *switch-GNN*.

The way edge information is included in the Transformer [101] literature is via *Edge Embeddings*, which generalise Relative Positional Embeddings (see Sec 2.4.5). This is in contrast to the GNN approach which primarily makes use of distinctly weighted linear transformations per edge type. We attempted to find an example of a GNN making use of edge embeddings in the GNN literature, but to the best of our knowledge this has not been done in published work. There are a few conceptual and practical advantages to using edge embeddings in place of a switch-GNN. Firstly, the number of parameters per edge type is far lower for edge embeddings when compared to switch-GNNs. This is because a parameterised linear transformation is defined by a weight matrix $\mathrm{W}^{\mathrm{r}} \in \mathbb{R}^{f\times f}$, requiring f^2 parameters per edge type. Edge embeddings however only require an f-dimensional vector to be learned per edge type. Using basis matrices can help switch-GNNs scale their parameters better with edge types, however we hypothesise that this may come at the cost of modeling capacity for cases where the number of edge types is far larger than the number of basis matrices. Edge embeddings may also allow for continuous and/or multidimensional edge features to be used, beyond discrete edge types. This could be achieved by learning a single function which maps every edge feature vector to an edge embedding vector. Given these advantages, it is of high relevance to evaluate the use of edge embeddings in a GNN-based system.

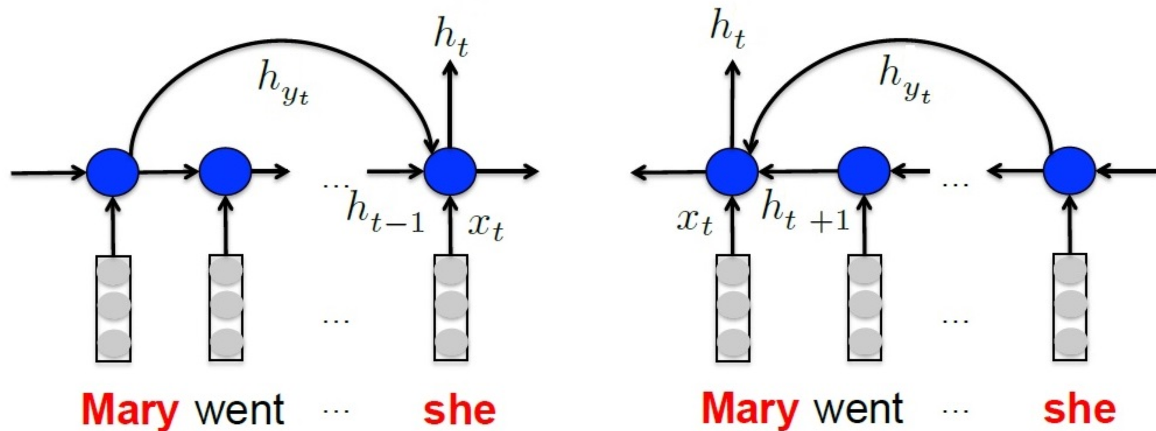

FIGURE 2.9: A diagram from [29] showing both directions of the modified Coref-GRU architecture. Here, h_t represent the usual sequential encoding of a typical RNN, while h_y represent the coreference-based skip connections. The encoding of each direction naturally forms a directed acyclic graph.

2.6 GNN-based MHQA models

Here we will introduce some key GNN-based MHQA models. We will describe these models by looking at three primary components which each GNN-based MHQA model contains. First *Graph Construction*, whereby the QA data point is converted into a graph using heuristic rules. Second *Graph Embedding*, whereby the graph nodes are embedded into a vector space. Third *Graph Encoding*, whereby the embedded graphs are further encoded by a GNN. For graph encoding, we will describe the relevant GNN architectures by using the message passing notation (see Sec 2.5.1). We will begin by describing an RNN-based MHQA model called Coref-GRU, before moving on to describe a set of GNN-based MHQA models. Finally, we will describe how these models used encoded graph states to predict question answers.

Coref-GRU (C-GRU) offered an extension to the typical RNN paradigm. Specifically, it added a new sequential bias to the RNN which exploited entity coreference information [29]. Text often contains entities such as "Mary", which may be mentioned multiple times, or may be referenced via pronouns such as "she" or "her". As an example: "**Mary** read **her** book ... **Mary** fell asleep." . These references to a shared entity are called coreferences.

RNNs encode tokens sequentially. This sequential encoding could be visualised as a directed acyclic graph (DAG) with no branches. C-GRU extends this DAG to include branches by directly connecting entity tokens to their coreference tokens. This effectively amounts to adding in skip connections between mentions of the same entity. Figure 2.9 illustrates this DAG and shows how this modified RNN is able to communicate information directly between entity coreferences. This extension allows the C-GRU to mitigate the RNN's long sequence problem by creating shortcuts across the text, centered around entities. They do this by modifying the GRU [21] equations to incorporate these skip connections, by summing in the state of the most recent entity coreference, where one is available [29].

With the success of the C-GRU model, one can begin to see the appeal of GNN-based MHQA models. Stepping away from RNN-based models towards GNN-based models allows the usage of richer graph structure by allowing for cyclic graphs [108]. GNN literature

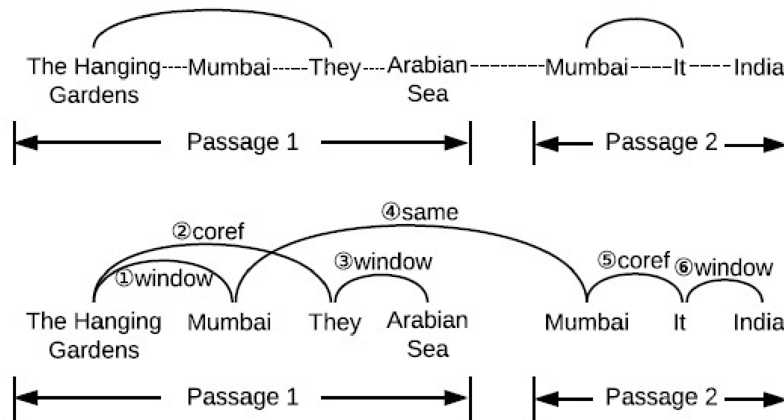

FIGURE 2.10: A figure from [95] contrasting the Coref-GRU (C-GRU) graphs [29] (top) with the GSPR graphs [95] (bottom). Here the dashed lines indicate that the C-GRU model is connecting all intermediate tokens sequentially, as is common in RNN-based NLP. The solid lines indicate a direct connection.

also includes provisions for factoring edge types into computations [86]. Within a short period of time, a host of GNN-based MHQA models were proposed, which all make use of entity graphs and GNNs [26, 95, 99, 13]. In this section, we provide some high level detail on the similarities and differences between these models. Where a particular paper offers multiple options for a single architectural choice, such as two distinct GNNs, we exclusively consider the option which performed best in terms of test accuracy, by the papers own internal results. We describe these GNN-based MHQA models by detailing their architectural choices in constructing entity graphs, embedding these graphs into a vector space, and finally encoding these graph embeddings using a GNN.

2.6.1 Graph Construction

The graphs constructed by the GNN-based MHQA models we catalogue all have at least one thing in common. Namely, they all make use of entities as nodes [99, 31, 26, 95, 13]. Two distinct approaches to extracting entities from text documents exist in GNN literature. The first involves the use of a Named Entity Recognition (NER) [62] module, which automatically detects named entities such as "Google" or "Table Mountain" in text. We will refer to these entities as *Detected Entities*. The second approach only applies to multiple-choice question answering, and involves extracting only those entities which are an exact text match with one of the answer candidates, or any entity mentioned in the question [99, 29]. We refer to these entities which match an answer candidate or question entity as *Special Entities*. We will now detail the diverging graph construction strategies of the various GNN-based MHQA models we consider in this work. Then at the end of this subsection, we will summarise all of the node and edge types used by the models we catalog here.

2.6.1.1 GSPR

Similar to the DAG graphs constructed by the C-GRU model is the Graph-structured Passage Representation (GSPR) model's graph construction. GSPR extends the entity coreference edge found in the C-GRU to include three edge types: Coref, Same, Window [95]. The Coref edge is used exclusively to link entity mentions to their pronoun coreferences [95], eg: "Mary -> Her". The 'Same' edge connects entity mentions whose texts match exactly [95],

eg "Mary -> Mary". Beyond this point, we will refer to the 'Same' edge as the 'Co-mention' edge. Finally the Window edge connects mentions of distinct entities [95]. All three of these edges are subject to a maximum distance window, meaning matching entities which are found far apart in the text will not be directly connected [95]. In this way, the graphs constructed by GSPR still contain a sequential bias, and are unable to shortcut information across large distances in the text. It should also be noted that the GSPR model makes use of detected entities instead of Special entities, and thus a pre-existing NER module is required [95]. Since the GSPR model makes use of a GNN instead of an RNN, graphs with cycles are tenable. As such, GSPR makes use of bidirectional edges. Figure 2.10 illustrates the differences between the C-GRU graphs and the GSPR graphs. Importantly, GSPR, and in fact all of the GNNs we catalog, do not represent tokens directly in their graphs and graph embeddings. This represents a major departure from the conventions of RNN-based and Transformer-based NLP. Instead of reasoning over tokens as in RNNs, GNN-based approaches reason over nodes, which each represent multiple tokens, or rather a token span defined as a start and end token. Thus, if a given sentence contains no nodes, then that sentence is largely omitted from the GNN encoding. The GNN-based approach then serves as a form of dimensionality reduction whereby the number of nodes represented may be significantly fewer than the number of tokens in the full text.

2.6.1.2 EGCN

The model Entity Graph Convolutional Networks (EGCN) offers a set of improvements and simplifications over the GSPR model, resulting in a higher evaluation accuracy on the Wikihop dataset [26, 95]. The most notable change in the EGCN graph structuring approach is the exclusion of the maximum distance window which is found in GSPR. This means that mentions of the same entity can be connected across arbitrarily long sections of text, or even across disparate documents. This decision imbues the model with an architectural bias towards global information correlation. EGCN includes both the 'Co-mention' and Coref edges which are present in the GSPR model [26]. However, instead of the Window edge, EGCN makes use of two further edge types, namely Co-Document, and Compliment. The Co-Document edge connects all entity nodes which are extracted from the same document. Finally the Compliment edge connects all remaining unconnected nodes, thereby creating fully connected graphs [26]. Unlike GSPR, EGCN makes use of Special entities instead of detected entities. Notably, the results of the EGCN evaluations show that the Coref edge does not significantly contribute to a higher test accuracy on Wikihop [26]. This seems to indicate that given the entity graph EGCN constructs, which richly connects entity mentions, local coreferences between entities and pronouns may no longer be required.

2.6.1.3 HDE

The Heterogeneous Document-Entity (HDE) system is the highest scoring GNN-based MHQA model on the Wikihop dataset leaderboard[1] as of this writing. It is similar to the EGCN, using the same GNN, using Special entities, and allowing connections between nodes which are far apart in text [99]. HDE does not include any pronoun coreference information, meaning no Coref edge, and all of the entity nodes are entity mentions, and none for pronouns. Notably, the graphs constructed by HDE contain nodes of different types, and represent information at two different granularities. The model includes nodes for entity mentions, as is common, however it also includes nodes for entire documents, and also for answer candidates [99]. Candidate nodes represent each answer candidate, and are separate from any particular document or entity mention. Document nodes represent an entire supporting document, and are linked to entity nodes which are mentioned in the respective document.

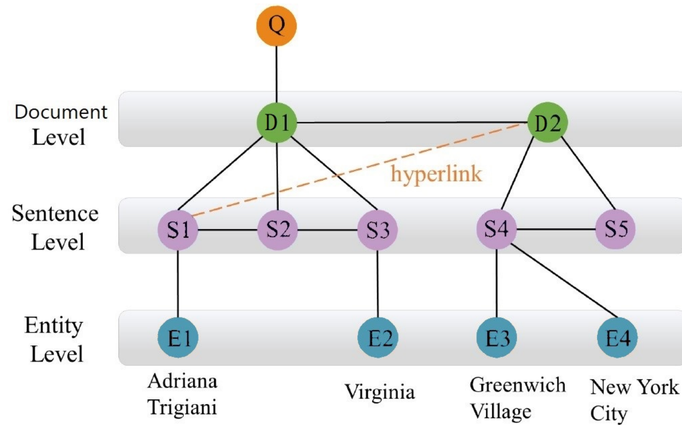

FIGURE 2.11: A visualisation from [31] for the graphs constructed by the HGN system. Here, the hyperlink edge is provided by auxiliary Wikipedia hyperlink data [31]. Beyond this, entity nodes are connected to their naturally containing sentences. Sentence nodes are connected to their containing documents. The "Q" node is a node for the question.

The HDE system makes use of 7 distinct edge types, which we detail in our model section 3.1. Figure 3.1 offers a simplified visual representation for the graphs constructed by HDE.

2.6.1.4 HGN

The Hierarchical Graph Network (HGN) model is a recent MHQA model which produces SOTA results on the challenging HotpotQA MHQA dataset [31]. While the HGN system does not contain candidate nodes, it does contain 2 additional node types which the HDE system does not, namely Sentence nodes and Question nodes [31]. Sentence nodes add an extra level of granularity between document nodes and entity nodes. The HGN system also makes use of a node for the question representation [31] as a way of incorporating question information into the model. To link nodes together, the HGN system makes use of the natural hierarchy of entities, sentences and documents. This is to say that the model links documents to their contained sentences, and links sentences to their contained entities [31].

To incorporate cross document links, the HGN model makes use of auxiliary data from Wikipedia, specifically hyperlink information [31]. This graph construction process is illustrated in Figure 2.11. The authors also provide a recommendation to avoid the need for this auxiliary data. Instead of using Wikipedia hyperlinks, one could connect sentence nodes to document nodes if any of the sentence's entities are found anywhere in the document, even if the sentence itself is not found in the document [31]. There are a few more subtle complexities that separate HDE from HGN, such as the explicit document selection process in the HGN model. HGN employs a hybrid learned and heuristic process to select paragraphs in stages [31]. This is in an attempt to reduce the amount of irrelevant information that enters the constructed graphs, and thereby decrease the graph sizes and therefore memory requirements too.

2.6.1.5 Summary

Here we summarise the graph construction rules used in the four GNN-based MHQA models we catalog. Table 2.1 recalls the node and edge types used in each model. The named

Model Name	Node Types	Edge Types
GSPR	Detected Entities, Pronouns	Entity Coreference, Co-mention, Window
EGCN	Special Entities, Pronouns	Entity Coreference, Co-mention, Co-document, Compliment
HDE	Special Entities, Document, Candidate	Co-mention, Co-document, Candidate-Candidate, Document-Contained_Entity, Document-Contained_Candidate, Candidate-Matching_Entity, Compliment
HGN	Detected Entities, Sentence, Document, Query	Query-Document, Document-Contained_Sentence, Sentence-Contained_Entity, Hyperlink, Sequential Sentence-Sentence

TABLE 2.1: A table summarising the node and edge types used by the four GNN-based MHQA models we catalogue. A-B implies all nodes of type A are connected to all nodes of type B; A-Condition_B implies nodes of type A are connected to all nodes of type B which meet the condition

edges are: Coreference - entities are connected to their pronouns; Co-mention - all matching entities; Window - all entities within a distance; Co-document - all entities found in the same document; Compliment - all as of yet unconnected entities. Overall, as these models evolved over time, there was a move away from locality imposed via distance thresholds towards hierarchical structure with nodes at different granularities. The newer HDE and HGN models also moved away from using entity coreference information.

2.6.2 Graph Embedding

Graph embedding is the process of mapping a graph's nodes into a vector space so that it can be processed by a neural network. To do this, we need to map the variable length token spans in each node to fixed-size vectors. The graph embedding process takes place in two steps:

1. **Token Embedding**, where token spans are converted into a sequence of token vectors [26, 95] via a pretrained token embedding module such as Glove [74] or BERT [27]. This is such that every token in the token span is mapped to a particular token embedding vector.
2. **Node Summarisation**, where the variable length sequence of token vectors for each node is mapped into a fixed size feature vector.

We will begin by describing the process of Token Embedding in a general sense, then move on to describe how various MHQA models incorporate Token Embedding.

2.6.2.1 Token Embedding

Embedding is the process of mapping a set of discrete values onto points in a shared vector space [22]. The position of these points can then be used as input to Neural Networks (NN)

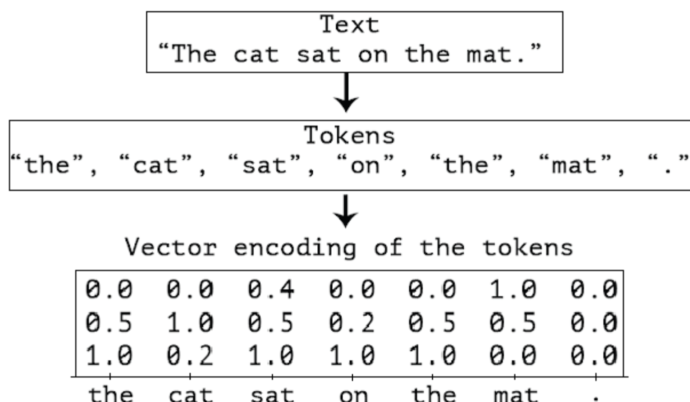

FIGURE 2.12: An example visualisation from [4] of a token embedding process. First the given text is broken up into tokens, then each token is associated with an n-dimensional vector. In this example each token is associated with a 3-dimensional vector.

[22]. Token embedding is then just the mapping of a set of discrete tokens to a set of points in a fixed dimensional vector space. Token embedding can be learned or fixed. A learned token embedder implies that the mapping from tokens to vectors can be updated during training. A fixed token embedder, by contrast, makes use of a fixed mapping.

Token embedders can be further broken down into two types, namely *contextual* and *non-contextual*. A contextual token embedder maps tokens to different vectors, depending on the surrounding tokens/ context. Consider the two phrases: "The boy named James", and "Oh boy, that sucked". Both phrases contain the word "boy", however when considering context, the meaning of the word changes in each phrase. A contextual embedder is in theory able to capture the different meanings of words in different contexts [75]. A non-contextual token embedder produces the same vector given the same token every time, and thus is unable to differentiate the different meanings of words in different contexts [74].

Token embedders such as Glove [74] or BERT [27] make use of massive quantities of text in order to find useful ways to represent natural language [74, 27]. Once these vector representations have been calculated from the large corpus of text, they can be used in other applications, without having access to the source corpus. This is related to the pretraining/fine-tuning paradigm whereby a large general data source is used to train a model on a general task. Token embeddings which are generated from large text corpora can then be fine-tuned for down stream tasks, which then require much less training data [27].

To embed a piece of text into a vector space, a token embedder first makes use of a tokeniser to break text up into small chunks called tokens. The token embedder then produces a unique n-dimensional vector for each token that it recognises. This process is visualised in figure 2.12. The set of all tokens that a token embedder recognises is called its vocabulary. Typically, a token embedder will be coupled to a particular tokeniser, with which it shares this vocabulary. In older token embedding systems such as Glove, these tokens represent whole words [74]. Due to the massive quantity of unique words in any given language, it is not feasible to store every single word in an embedder's vocabulary. Instead models like Glove must select a set of maximally useful tokens for which to find representation vectors. The Glove model simply chooses the set of words which occur most frequently in its source corpus [74]. This approach breaks down for rare words, or certain derivative words such "antiquing" where the word in question is simply not well represented in the source

corpus. Due to these issues, it was common in MHQA models to include character level embeddings to supplement Glove's word level embeddings [99, 29]. Thus, when a particular word was out of Glove's vocabulary, the character-level representation may still be able to capture some of the words meaning. To remedy this problem, modern language models like BERT [27] make use of an algorithm called Byte Pair Encoding (BPE) [27, 66, 77]. Conceptually, BPE is somewhere in-between a word-level and character-level representation. BPE represents common words as whole words, while breaking rare words up into subwords [90] such as {"anti", "qu", "ing"}. Since any given word could be broken up into subwords in many different ways, these subwords are chosen such that their frequency in the source corpus is maximised [90]. BPE is referred to as being open-vocabulary [90], meaning there are no words for which BPE fails to produce representations [90]. This effectively removes the need to use character-level representations while using BPE.

We will now describe the various ways MHQA models make use of token embeddings to generate graph embeddings.

2.6.2.2 C-GRU

Given a textual context document $s \in S_q$, C-GRU uses Glove [74], as well as a 10-dimensional character-level embedder. C-GRU makes use of a Convolutional Neural Network (CNN) to encode characters [29]. To combine the Glove and character embedding vectors produced per word, C-GRU simply concatenates the Glove and character-level representations together along the feature dimension [29]. This is to say that given "the apple" as an input, C-GRU would for example produce two 300-dimensional vectors using Glove [74], as well as two 10-dimensional vectors using its character-level encoder. After concatenation, the final encoding would then be two 310-dimensional vectors, one for each word. Since C-GRU is an RNN, and not a GNN, no Node Summarisation is required.

2.6.2.3 GSPR

The GSPR model makes use of a Glove [74] word embedder, without a supplementary character embedder [95]. GSPR, along with the HDE model [99], further encodes token embeddings with an RNN, before mapping them into node embeddings [99]. Specifically, given a textual document s and query q, GSPR generates initial token embeddings X_s^0 and X_q^0 using a Glove token embedder [95]. The GSPR model then further encodes these token embedding vectors with a bidirectional LSTM [43]. This yields new token embeddings X_s^1 and X_q^1 such that

$$X^1 = \text{biLSTM}(X^0), \tag{2.43}$$

for both the document and query. To generate entity node embeddings from the encoded token embeddings X_s^1 and X_q^1, GSPR concatenates the first and last token representation for every entity token span [95]. An entity ϵ with a token span of (ϵ^i, ϵ^j) indicates that ϵ covers all the tokens from i to j. The two token embedding vectors we need for entity ϵ are thus tokens i and j, and all intermediate vectors between i and j are ignored. Once the first and last token vectors for an entity ϵ have been concatenated, the result is transformed via a learned linear transformation [95]. We refer to this as *head and tail concatenation*, which follows:

$$X_\epsilon = \text{Concat}_f(X_s^1[i], X_s^1[j])W_1 + b_1. \tag{2.44}$$

This yields a fixed size vector representation X_ϵ for every entity present in the document s. Here $X[i]$ represents extracting the ith vector from the matrix X, with respect to the sequence dimension. Similarly, a fixed size query representation is obtained using a distinctly

parameterised head and tail concatenation as follows:

$$X_q = \text{Concat}_f(X_q^1[1], X_q^1[l_q])\text{W}_2 + b_2, \tag{2.45}$$

where l_q is the number of query tokens.

Finally, to combine query and context information, every entity node representation X_ϵ is concatenated with the fixed size query vector X_q, and again transformed via another learned linear transformation. Thus the final query-aware entity node representations Y_ϵ follow:

$$Y_\epsilon = \text{Concat}_f(X_\epsilon, X_q)\text{W}_3 + b_3, \tag{2.46}$$

where $\text{W}_1, \text{W}_2, \text{W}_3, b_1, b_2, b_3$ are all model parameters.

2.6.2.4 EGCN

The EGCN model [26] makes use of the ELMo [75] token embedder. ELMo is a contextual embedder, like BERT, but makes use of an LSTM-based encoding architecture [75] instead of BERT's Transformer-based architecture [27]. EGCN encodes both the context and query using ELMo, however it also further encodes the query representation with an LSTM [26]. Finally, similar to GSPR, it introduces query information into the context by concatenating the query representation with all of the entity node representations, before transforming these concatenated vectors using a learned transformation. EGCN makes use of a two-layer MLP instead of the linear transformation which is used in GSPR.

2.6.2.5 HDE

The HDE system makes use of a comparatively more complex graph embedding process than does GSPR or EGCN. HDE begins by making use of a 300-dimensional GloVe [74] token embedder, alongside a 100-dimensional character-level embedder, resulting in a combined 400-dimensional token embedding process [99]. Next, HDE makes use of three distinct GRU [21] encoders for the query, document, and candidate token embeddings respectively [99]. In GSPR and EGCN, query information is introduced to node vectors after they have been summarised from their respective span of token vectors [95, 26]. In HDE however, query information is injected directly into token embeddings before node summarisation [99]. In theory, this may increase the expressiveness of the context/query interactions since their interaction is computed at the high-dimensional token level, instead of the lower-dimensional node level. Specifically, HDE makes use of a coattention module [116] to produce query-aware token vector sequences. These query-aware token vector sequences are then mapped into fixed size node vectors by a self-attentive pooling module [99]. This effectively amounts to performing a learned weighted sum over all token vectors in the respective span. This is in contrast to the head and tail method, where intermediate tokens are ignored by the summarisation process.

2.6.2.6 HGN

The HGN model makes use of a pretrained RoBERTa [66] model, which is similar to BERT only larger and with a different pretraining objective. The model concatenates all context documents together, and then concatenates that with the query too. This long text sequence of context and query is then tokenised and embedded using the RoBERTa model [31]. This is only possible since HGN performs document selection, and thus does not include all supporting documents [31]. Including all documents would make the resulting token sequence too long for RoBERTa to encode. The resulting token embeddings are then passed through

an LSTM encoder to obtain the final token embeddings [31]. These token embeddings are then used to generate node embeddings by using the head and tail concatenation method. This method of introducing query information is similar to HDE's in that it introduces the query information at the token level. However, HGN introduces the query information during token embedding rather than after as with HDE. Since HGN uses RoBERTa instead of GloVe, this query information is able to affect the initial token embeddings. This is because contextual embedders such as BERT, ELMo, and RoBERTa function as units of computation, not just a static lookup table as in GloVe. If fine-tuned, these contextual token embedders could learn to aid in the question answering process before any other part of the model is used. HGN distinguishes itself from the rest of the GNN-based MHQA models we catalogued in that it also adds a query node to its graph. Thus, HGN in fact introduces the same query information in two different places, once right at the start of the model before token embedding, and a second time during the GNN encoding step.

2.6.3 GNN Architectures

Here we detail the GNNs used in these MHQA models using the message passing notation.

2.6.3.1 GSPR GNN

The GSPR model implements a GNN based off of the popular LSTM RNN [43]. This means passing 2 vectors per node forward through the GNN. In addition to the node states $\boldsymbol{h}_i^k$ for the ith node at the kth GNN layer, the LSTM GNN also maintains *cell vectors* $\boldsymbol{c}_i^k$. In the notation of the message passing pattern, the GSPR GNN has a trivial message function and a sum-based aggregation function. All of the GSPR's computation is thus taking place in its LSTM-based update function. To define this LSTM-based gating function Gate_l we begin by defining a single layer MLP with a bias term b and a nonlinear activation function σ as

$$\text{MLP}^y(\boldsymbol{x}) = \sigma(\text{W}_\text{y}\boldsymbol{x} + b_y). \tag{2.47}$$

Given a vector input of A_i^k, the LSTM gating function first maps the input vector A_i^k to 4 distinct vectors. namely:

$$\boldsymbol{i}_i^k = \text{MLP}^i(A_i^k), \tag{2.48}$$

$$\boldsymbol{o}_i^k = \text{MLP}^o(A_i^k), \tag{2.49}$$

$$\boldsymbol{f}_i^k = \text{MLP}^f(A_i^k), \tag{2.50}$$

$$\boldsymbol{u}_i^k = \text{MLP}^u(A_i^k). \tag{2.51}$$

Then the cell c_i^{k+1} vector is calculated following:

$$\boldsymbol{c}_i^{k+1} = \boldsymbol{f}_i \odot \boldsymbol{c}_i^k + \boldsymbol{i}_i^k \odot \boldsymbol{u}_i^k. \tag{2.52}$$

The full LSTM gating function Gate_l is then:

$$\text{Gate}_l(A_i^k, \boldsymbol{c}_i^k) = \boldsymbol{o}_i^k \odot tanh(\boldsymbol{c}_i^{k+1}) \tag{2.53}$$

Any GNN which uses this LSTM-based gating function must be modified to maintain a cell state vector for every node, for each layer of the GNN. The output of the GNN is however only the transformed input vectors $\boldsymbol{h}_i^{k'}$. Thus, the LSTM-based gating function is a function which maps $\boldsymbol{h}_i^k \rightarrow \boldsymbol{h}_i^{k+1}$, while using the cell state vectors as an auxiliary channel of information. Using the message passing notation modified to include the cell state vectors, the equations of this GNN follow:

Message:

$$\phi(\boldsymbol{h}_i^k, \boldsymbol{h}_j^k, E) = \boldsymbol{h}_j^k \tag{2.54}$$

Aggregate:

$$\omega(M_{ij}^k, \boldsymbol{h}_i, \boldsymbol{h}_j) = M_{ij}^k \tag{2.55}$$

Update:

$$\gamma(\boldsymbol{h}_i^k, A_i^k, \boldsymbol{c}_i^k) = \text{Gate}_l(A_i^k, \boldsymbol{c}_i^k) \tag{2.56}$$

Here, $\boldsymbol{c}_i^k$ is a function of $\boldsymbol{f}_i^k$, $\boldsymbol{i}_i^k$, $\boldsymbol{u}_i^k$, and $\boldsymbol{c}_i^{k-1}$. Since $\boldsymbol{f}_i^k$, $\boldsymbol{i}_i^k$, $\boldsymbol{u}_i^k$ are all functions of A_i^k, there is no distinction between in-nodes and out-nodes, thus this GNN is in-out symmetric.

2.6.3.2 EGCN and HDE GNN

The HDE and EGCN models make use of an identical GNN encoding process. Specifically, they both make use of a gated version of the Relational Graph Convolutional Neural Network (R-GCN). We describe their GNN using the message passing notation:

Message:

$$\phi(\boldsymbol{h}_i^k, \boldsymbol{h}_j^k, E) = \frac{\mathrm{W_r}\boldsymbol{h}_j^k}{|E_i^r|} \tag{2.57}$$

Aggregate:

$$\omega(M_{ij}^k, \boldsymbol{h}_i^k, \boldsymbol{h}_j^k) = \frac{M_{ij}^k}{|N(i)|} \tag{2.58}$$

Update:

$$\gamma(\boldsymbol{h}_i^k, A_i^k) = \text{Gate}(\boldsymbol{h}_i^k, A_i^k + \mathrm{W_0}\boldsymbol{h}_i^k) \tag{2.59}$$

Here $\mathrm{W_r}$ is a different weight matrix for each edge type r, and $|E_i^r|$ is the number of edges of type r which point to node i. Gate is the gating function defined by equation 2.37. From the update equation we can see that $\boldsymbol{h}_i^k$ is distinguished from A_i^k in two places in the update function, thus this GNN is in-out asymmetric. $\mathrm{W_0}$ is a learned weight matrix used to transformer in-node states. This is similar to the R-GCN where the addition of $\mathrm{W_0}\boldsymbol{h}_i^k$ is the only feature making the GNN in-out asymmetric. Here however the gating function fills this role, possibly making the addition of $\mathrm{W_0}\boldsymbol{h}_i^k$ redundant.

2.6.3.3 HGN GNN

Finally the HGN model [99] makes use of a modified version of the GAT [102] GNN (see sec 2.5.2.2). The HGN does make use of gating, but not in its GNN, as is common in the GNN-based MHQA models we have catalogued. Instead the HGN model uses a gate around its entire GNN in order to combine its context token representations with the graph encoding outputted by its GNN [31]. To introduce edge information into the GAT, the HGN model modifies equation 2.27 to make use of a distinct weight matrix per edge type, making the GNN a switch-GNN. Formally, eq 2.27 becomes:

$$Att(\boldsymbol{h}_i, \boldsymbol{h}_j) = \psi(\text{Concat}_f(\boldsymbol{h}_i, \boldsymbol{h}_j)\mathrm{W_r}). \tag{2.60}$$

This is distinct from the way in which Relative Positional Embedding Transformers incorporate edge information, whereby edge embeddings are injected [92] instead of making use of different linear transformations per edge type [31]. This modification to the GAT does

not add an in-out asymmetry as the update function is not changed. Thus this modified GAT is still in-out asymmetric.

2.6.4 Output models

Here we will discuss how these GNN-based MHQA models produce an output which can be used for training model parameters, and performing inference. We will mainly be discussing the models which evaluate on the multiple choice dataset Wikihop [106]. The HGN model [31] is a multi-task model, meaning it needs a sophisticated output model which is capable of outputting different kinds of outputs depending on its input. This is less relevant to our work however which focuses only on multiple choice question answering. Outputting an answer to a multiple choice question generally involves producing a probability distribution over all of the unique entities being modeled [99, 95, 26]. Since any given unique entity may be mentioned multiple times throughout the given context, a common requirement is for these models to aggregate these entity mentions while outputting a prediction.

The GNNs in these models output a graph encoding $H^L \in \mathbb{R}^{n \times f}$ for an L layer GNN. Each element $\mathbf{h}_i^L \in \mathbb{R}^f$ is a vector representing the final encoded state of one of the nodes in the graph. Some or all of these nodes will represent entities, depending on the model. Thus, given H^L, and a single candidate $c \in C_q$, these models must extract all of the entity node vectors which correspond to candidate c. We will represent the set of entity vectors for candidate c as $E_c = \{\mathbf{h}_i^L \ \forall i \mid i \ is \ a \ mention \ of \ c\}$.

The EGCN model [26] concatenates entity vectors with the the fixed-size query representation $\hat{\mathbf{q}}$, then scores the result for each entity mention. The score for a particular candidate is just the maximum score over each of the candidate's mentions. More formally:

$$c_s = \max_{\mathbf{e} \in E_c}(\text{MLP}(\text{Concat}_f(\hat{\mathbf{q}}, \mathbf{e}))), \tag{2.61}$$

where c_s is a scalar score for candidate c. The final probability distribution over all candidates is then obtained by performing a softmax over all candidate scores c_s.

The GSPR model [95] uses an additive attention-based [6] output model where the attention scores themselves are used as the likelihood that a given entity is the correct answer. The model also combines likelihood scores for the representations of entities from every layer of the GNN, not just the final layer. Finally the HDE model [99] makes use of a similar strategy to EGCN, whereby it scores entity mentions, and returns the maximum score. Unlike EGCN, this scoring does not include a query representation. Furthermore, since the HDE model includes candidate nodes, its output model factors their final representation into its calculation. Formally:

$$c_s = \text{MLP}_c(\mathbf{h}_c^L) + \max_{\mathbf{e} \in E_c}(\text{MLP}_\epsilon(\mathbf{e})), \tag{2.62}$$

where MLP_c and MLP_ϵ are MLPs to score candidates and entities respectively. $\mathbf{h}_c^L$ is the final vector representation for candidate c. Here, each MLP has two layers such that the hidden layer size is half that of the model dimension. More formally:

$$MLP_\epsilon(x) = tanh(x\text{W}_\epsilon^1 + b_\epsilon^1)\text{W}_\epsilon^2 + b_\epsilon^2, \tag{2.63}$$

where $x \in \mathbb{R}^f$ is some input, and $\text{W}_\epsilon^1 \in \mathbb{R}^{f \times \frac{f}{2}}$, $\text{W}_\epsilon^2 \in \mathbb{R}^{\frac{f}{2} \times 1}$, $b_\epsilon^1 \in \mathbb{R}^{\frac{f}{2}}$, $b_\epsilon^2 \in \mathbb{R}^1$ are model parameters. *tanh* is the hyperbolic tan activation function.

2.6.5 Summary

Overall, just cataloging this small set of GNN-based MHQA models presents us with many architectural choices at every level of the pipeline. There are different graph construction options, as well as options for graph embedding, and GNN encoding. We are also given options as to how to embed tokens, and how to incorporate query and candidate information into our model. Overall, given the large set of options, and the even larger space of combinations, we decided to primarily base our models off of the HDE model [99]. This decision was mainly due to its high performance on the Wikihop dataset. The HGN model, developed by Microsoft [31], appears to be more sophisticated than the HDE model in its graph construction, but also in some auxiliary processes such as document selection. We also see the use of mixture of experts style relational GNNs (switch-GNN) [31, 86], while the use of Transformer style edge embeddings seems unexplored in GNN-based MHQA models.

Finally we note the distinction between in-out symmetrical and in-out asymmetrical GNNs, and hypothesise that in-out asymmetry may help a GNN learn by preventing node states from becoming homogeneous. This is related to the well documented problem, sometimes called over-smoothing [14, 70], whereby deeper GNNs cause node states to become more similar to each other [108], effectively losing their identities and thereby degrading model performance [108, 70].

Chapter 3

Model

Here, we introduce and describe our Graph Neural Network (GNN)-based Multihop Question Answering (MHQA) models, trained to answer multiple-choice natural-language questions. A multiple choice question comes in the form of a plain text question q, a set of plain text context documents S_q containing information needed to answer the question, a set of plain text answer candidates C_q, and finally a plain text answer to the question $a_q \in C_q$.

We aim to investigate the use of GNNs in Natural Language Processing (NLP), and to compare specific GNN design choices with one another, and with design choices in relevant Transformer literature. To this end we begin by creating a GNN-based MHQA model which is inspired by the Heterogeneous Document Entity (HDE) model [99] (see Sec 2.6.3). This model shares a similar graph structuring, graph embedding and GNN step. It also makes use of an identical output prediction model to HDE [99]. We evaluate this model and all its variants on the Wikihop [106] dataset. Our research methodology involves evaluating this MHQA model with an HDE-like GNN, and gradually making the GNN more Transformer-like by adding in features common in the Transformer literature. We ablate the usage of all such Transformer features and use these ablations as a way to compare some of the typical GNN design choices to Transformer design choices, under our specific conditions. We also evaluate some common GNN features in combination with Transformer features. In this chapter, we detail the various modeling choices we consider in our experiments.

On a high level, our models use heuristic rules to construct graphs from a question's given context. They then use a pretrained token embedder such as Glove [74] or BERT [27] to generate token embeddings. Next, they use parameterised Transformer-encoder layers to create graph embeddings given token embeddings. These graph embeddings are then encoded by a parameterised GNN, before finally being used to predict the correct answer from the given candidates. This chapter is structured to correspond with the GNN-based MHQA model pipeline, which follows: graph structuring; graph embedding; GNN encoding; and finally output model.

3.1 Graph Construction

Similar to HDE [99], our model uses given multiple-choice question-answer examples to construct graphs composed of entities, answer candidates, documents and optionally sentence nodes. We refer to these graphs as *context graphs*, and distinguish them from HDE graphs [99] (see Sec 2.6.1.3), which do not contain sentence nodes. We use the term context graph to refer to the general graph case, not with any one specific construction rule set in mind. Thus, an HDE graph is an instance of the more general context graph.

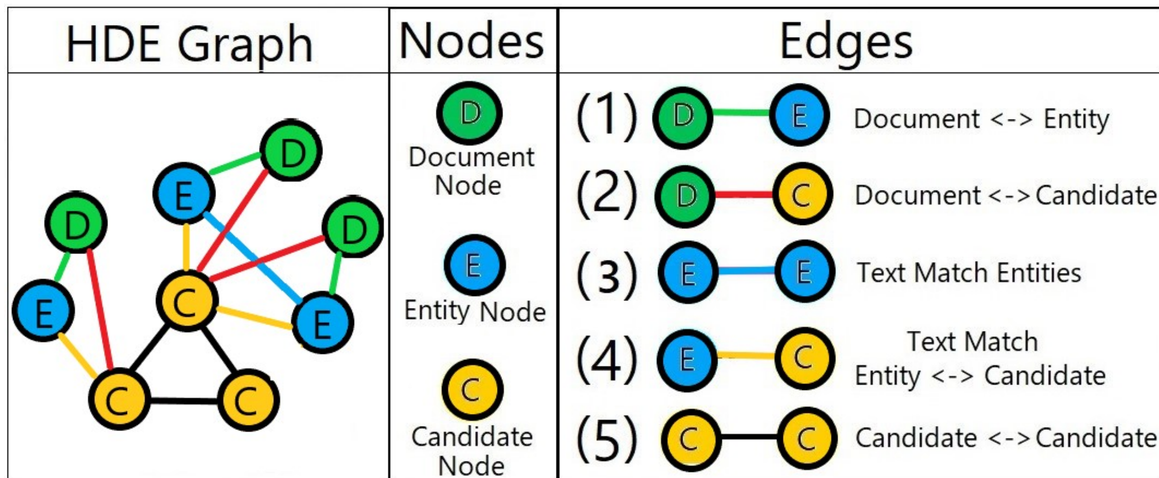

FIGURE 3.1: A visualisation of a graph constructed by our model excluding the optional edges and sentence nodes. These graphs are similar to those constructed by HDE [99] but without the Co-document or Compliment edges.

These context graphs are heterogeneous in that they contain multiple node and edge types. They also include information at multiple levels of granularity, namely the phrase-level entity and candidate nodes, sentence-level nodes, and the coarse-grained document nodes. Since each extracted entity can be mentioned multiple times in the same, or different documents, there may be multiple distinct entity nodes which represent identical text, but with different token spans and/or source documents.

HDE makes use of Special entities instead of Detected entities (see Sec 2.6.1), which means that no NER module is required. Informally, we found that the NER approach failed in many cases where the correct answer was not strictly a Named Entity, such as when the answer is a number like "68" or when the answer is a generic noun like "sports". This may be remedied in future work by also including generic nouns as nodes. The Special entities approach results in notably fewer entity nodes and thus contributes to faster execution times. However, due to the nature of multihop question answering, it is often necessary to 'pivot' around entities (See Fig 1.2, Sec 2.2.3) which may not themselves be in the question or answer candidates. We also found cases where the Special Entity approach failed in Wikihop, such as when the answer candidates are not found in the context via an exact text match. As an example, there exists a Wikihop question: 'participant_of juan rossell?', however in the text Juan is referred to only as 'Juan Miguel Rossell Milanes'. In this case, using only Special Entities would make the question unanswerable using reasoning. These failure cases remain unaddressed in existing Special Entity GNN-based MHQA models. In these cases, including only the Special entities may be insufficient, and thus including both Special and Detected entities may improve the models ability to reason. Regardless of the entity extraction method, the output is a list of token spans denoting the start and end token of each entity contained in each document.

We also optionally include sentence nodes inspired by the HGN model [31]. In this case, sentences must be detected by a sentence detection module (sentence splitting) [87]. We consider two methods of including sentence nodes. One in which all detected sentences are included and one in which only those sentences who contain an extracted entity are included (restricted sentences).

Once entities and sentences have been identified, we are ready to generate context graphs.

Using the given documents and their contained entity/sentence spans, as well as all answer candidates we construct the following node types:

- A Document node for each supporting document.
- An entity node for each extracted entity mention in each document.
- A candidate node for each answer candidate.
- Optionally, a sentence node for each sentence found in each document. We optionally filter these sentences to include only those sentences which contain an extracted entity.

Once the nodes have been created, we use a set of heuristic rules to connect them. Each connection between nodes has an edge type. We make use of a set of fundamental edge types, which are present in all of our models. The following connection rules are used to generate our fundamental edge types in a manner similar to HDE [99]:

1. Document nodes are connected to entity nodes if their represented text is found in the respective document.
2. Document nodes are also connected to candidate nodes if their text is found in the document.
3. *Co-mention Edge*: Each Entity node pair whose texts match is connected. This edge allows mentions of the same entity to connect across long distance in text, and even across distinct documents.
4. Each Entity node is connected to any candidate node whose text is a mention of the entity.
5. All candidate nodes are connected to each other.

We also implement the following optional edge types, which are unused unless otherwise stated:

6. *Co-document Edge*: All entity nodes which are mentioned in the same document are connected.
7. *Compliment edge*: All entity pairs which remain unconnected are connected.
8. *Unconnected edge*: All unconnected nodes are connected. Adding this edge turns any graph into a fully connected graph.

Finally, if sentence nodes are included in our graphs, we make use of the following additional edge types:

9. Sentence nodes are connected to the document node to which they belong
10. Sentence nodes are also connected to any entity nodes whose text is found in the respective sentence. Including entity mentions found in different documents to the sentence.
11. Sentence nodes are connected sequentially to the previous and next sentences.
12. Lastly, sentence nodes are connected to document nodes if any of the document's entities are found in the sentence. This represents another cross-document link.

Figure 3.1 offers an abstract visual representation for the context graphs constructed by our system, and figure 4.1 offers a simplified example of a graph constructed by our system using a real Wikihop data point. Both diagrams omit sentence nodes and their connections.

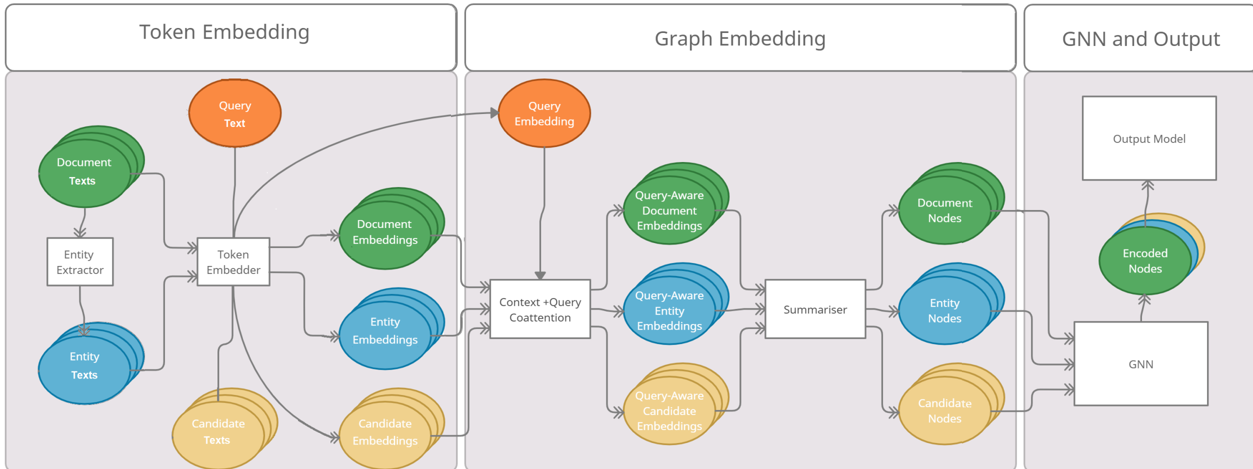

FIGURE 3.2: A diagram depicting the flow of context information through our system. Graph structuring is not included in this diagram for brevity, however this involves the creation of the Edge Matrix E, which is inputted to the GNN along with the node embeddings. The coattention step combines the query embedding with every document, entity, and candidate embedding separately.

3.2 Graph Embedding

Our graph embedding process is similar to HDE's [99] graph embedding process, differing in a few key areas which we detail here. For multiple choice question answering, graph embedding involves generating token embeddings for the context documents, the question, and each of the answer candidates. These token embeddings are then used alongside the constructed context graph to create query-aware node embeddings of fixed size. To do this, query information is injected into token embedding sequences via a coattention module, before being summarised into a single fixed-size vector. Figure 3.2 offers a high-level representation of this process which is used to generate entity, document, and candidate node vectors, which are then encoded with a GNN. The process of creating sentence nodes is functionally identical to that of entity nodes, just making use of different token spans.

3.2.1 Token Embedding

A token embedding (see Sec 2.6.2.1) module consists of a vocabulary made up of recognised tokens, and a mapping of (token, vector) pairs. These vectors can be acquired in various ways for different types of embedders, but conceptually, they should all contain information about the tokens they represent. Formally, we define our token embedder as a black box function $\text{Emb}(text)$ which uses a tokeniser to break text up into tokens with respect to its vocabulary, before converting these tokens into vectors. This process could be learned or fixed. However, we primarily focus on using a 300-dimensional pretrained Glove [74] module, and do not fine-tune the token embeddings, resulting in a fixed token embedding process. It should be noted that HDE [99] uses both a 300-dimensional Glove [74] module as well as a 100-dimensional character n-gram module [98], resulting in a combined 400-dimensional embedding module. Furthermore, HDE makes use of GRUs [21] to further encode their 400-dimensional embeddings, ultimately resulting in a more complex token embedding process than we consider. We do however ablate the usage of these GRUs. We

also experiment with a 512-dimensional BERT-based token embedder. We evaluate the use of BERT with and without fine-tuning.

Given a question q, a set of supporting documents S_q, and a set of answer candidates C_q, we generate token embedding sequences for each document, candidate, and the query, all independently, using the same embedding module. This yields $l \times f$ matrices X_q, X_s^i, and X_c^j for the query, i^{th} document, and j^{th} candidate respectively, where l is the length of the respective sequence, and is equal to the number of tokens the sequence represents. More formally:

$$X_q = \mathrm{Emb}(q) \in \mathbb{R}^{l_q \times f}, \tag{3.1}$$

$$X_s^i = \mathrm{Emb}(s^i) \ \in \mathbb{R}^{l_s^i \times f} \quad \forall \quad s^i \in S_q, \tag{3.2}$$

$$X_c^j = \mathrm{Emb}(c^j) \ \in \mathbb{R}^{l_c^j \times f} \quad \forall \quad c^j \in C_q. \tag{3.3}$$

We refer to these X matrices as *embedding matrices*, and note that there is one embedding matrix for each document and candidate, as well as one embedding matrix for the question. Context graph nodes represent token spans in token sequences. Thus, each node corresponds to a particular subsequence in a given embedding matrix. We refer to these subsequences of embedding matrices as *subsequence matrices*, and denote the general subsequence matrix as X_t. Thus, $X_t \subseteq X$ as the subsequence can cover part of or the entire embedding matrix. Once we have our embedding matrices, we need to extract our subsequence matrices and inject query information into them.

3.2.2 Context-Query Coattention

Similarly to HDE, we make use of Coattention [116] to inject query information into our subsequence matrices. Here coattention is distinct from both self-attention [101] and cross-attention [49]. Self-attention is when a sequence attends over itself; cross-attention is when one sequence attends over another, while coattention is when two sequences attend over each other. HDE makes use of a coattention module [116] which combines a GRU encoder with a custom single-headed attention mechanism. For simplicity, we decided to replace this coattention module with a Transformer encoder [101], which has a native implementation available in Pytorch.[6] This decision also allows our coattention module to make use of multi-headed attention.

Given an embedding matrix X for a document (X_s) or candidate (X_c), we gather all subsequence matrices X_t contained in X. This information is retrieved from our constructed context graphs such that there is one subsequence matrix per node. For candidate nodes, the subsequence matrix $X_t = X_c$. This means that candidate node token spans cover the entire candidate token sequence. Similarly for document nodes $X_t = X_s$, meaning document node token spans cover their full document token sequence. However, for entity and sentence nodes, their token spans cover only part of the full document token sequence, meaning $X_t \subset X_s$.

The task of injecting query information into a subsequence matrix using coattention then follows:

$$Y_t = \mathrm{Coattention}(X_t, X_q), \tag{3.4}$$

where Y_t is a query aware subsequence matrix which is equivalently sized to the X_t, and X_q is the full query embedding matrix. Coattention is some attention-based module which allows two sequences to attend to each other.

[6] Pytorch Transformer: https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html

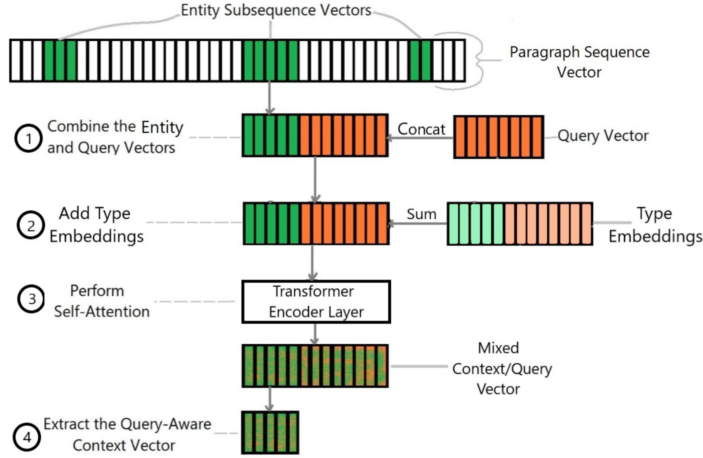

FIGURE 3.3: A diagram showing the process of using a Transformer encoder to perform coattention between the context of a single node and the query representation. Here we show coattention between the query vector (orange) and an entity vector (green), however a similar process applies for document vectors and candidate vectors. Entity vectors are contained inside of document vectors, thus these entity vectors need to be extracted from their containing document vectors. For document and candidate nodes, the entire document or candidate sequence vectors are used. Here the type embeddings help the Transformer differentiate between the context and query vectors.

To perform coattention with a standard Transformer encoder Trans_c, we follow algorithm 2. This involves concatenating each subsequence matrix X_t with the query embedding matrix X_q along the sequence dimension to produce X_{tq}. Given $X_t \in \mathbb{R}^{l_t \times f}$ and $X_q \in \mathbb{R}^{l_q \times f}$, the concatenation produces the matrix $X_{tq} \in \mathbb{R}^{(l_t+l_q)\times f}$. We then use our Transformer encoder Trans_c to perform self-attention over X_{tq}, thus allowing query tokens to attend to context tokens, and context tokens to attend to query tokens. Once we have encoded X_{tq}, we need to extract the elements which correspond to the inputted context tokens, separating them from the query tokens. More formally:

$$Y_{tq} = \text{Trans}_c(\text{Concat}_s(X_t, X_q)), \tag{3.5}$$

$$Y_t = Y_{tq}[1 : l_t]. \tag{3.6}$$

Here, $[a : b]$ refers to the extraction of a subsequence matrix from position a to position b. Concat_s refers to concatenation along the sequence dimension, not the feature dimension. Thus Y_t is same-sized to X_t, while containing query information.

The coattention used by HDE is asymmetric about the context and query representations inputted [99]. This is to say, if you were to swap the context and query representations, the coattention module would output a different result. When using Transformer encoders, however, we need to manually introduce this asymmetry. This is because the self-attention mechanism used in the Transformer is permutations invariant [101]. To allow the Transformer Trans_c to differentiate between the context and query sequence elements, we add a type embedding to each element before it is encoded by Trans_c. Specifically, the model learns two type embedding vectors for 'context' and 'query'. Summing type embeddings

Algorithm 2 Transformer-based Coattention and Summarisation

Require:
A node N with token span (a, b)
A context embedding matrix X in which node N resides
A query token embedding matrix X_q of length l_q
A type embedding vector **c** for 'context' and **q** for 'query'
A Transformer encoder for Coattention Trans_c
A Transformer encoder for Summarisation Trans_s

1: $X_t \leftarrow X[a : b]$ ▷ Extract Subsequence Matrix
2: $l_t \leftarrow (b - a)$
3: $T_c \leftarrow \text{Repeat}(\mathbf{c}, l_t)$ ▷ Get Type Embedding Matrices
4: $T_q \leftarrow \text{Repeat}(\mathbf{q}, l_q)$

5: $X_t \leftarrow X_t + T_c$ ▷ Add Type Embeddings
6: $X_q \leftarrow X_q + T_q$

7: $X_{tq} \leftarrow \text{Concat}_s(X_t, X_q)$
8: $Y_{tq} \leftarrow \text{Trans}_c(X_{tq})$ ▷ Perform Coattention
9: $Y_t \leftarrow Y_{tq}[1 : l_t]$ ▷ Extract Query-Aware Context Matrix

10: $Z_t \leftarrow \text{Trans}_s(Y_t)$ ▷ Encode for Summarisation
11: $\mathbf{h}_t^1 \leftarrow Z_t[1]$ ▷ Extract First Node Embedding
return $\mathbf{h}_t^1$

for these two types into their respective sequence elements provides explicit type information to Trans_c about each token. Factoring type embeddings into our equations yields:

$$Y_{tq} = \text{Trans}_c(\text{Concat}_s(X_t + T_c, X_q + T_q)), \tag{3.7}$$

where T_c is a type embedding matrix for context tokens, while T_q is a type embedding matrix for query tokens. These type embedding matrices are the same size as the matrices with which they are summed, and they are constructed by simply repeating a single embedding vector for the required length. Figure 3.3 depicts this Transformer-based coattention for the case of entity nodes, whose subsequence vectors are nestled within longer document sequence vectors.

3.2.3 Summariser

Once we have produced our query-aware context matrices Y_t for each node in our constructed context graph, we are ready to generate our initial node embeddings. Node embeddings $\mathbf{h}_t^1 \in \mathbb{R}^f$ must be of fixed size, but $Y_t \in \mathbb{R}^{l \times f}$ is variable length, with l being different for each node t. Thus, we follow HDE [99] by employing an attention-based summarisation step where the variable length Y_t representations are mapped to fixed length node embeddings $\mathbf{h}_t^1$. For this, we use another Transformer encoder Trans_s to encode our Y_t matrices. After encoding Y_t with Trans_s, we extract only the first element of the sequence, thus producing a fixed-size node embedding. More formally:

$$Z_t = \text{Trans}_s(Y_t), \tag{3.8}$$

$$h_t^1 = Z_t[1], \tag{3.9}$$

where $[i]$ refers to extracting the first sequence element from a matrix. This process is shown alongside the coattention step in algorithm 2.

3.2.4 Switch-Summariser

HDE does not incorporate node types when performing summarisation on sequences representing documents, entities, and candidates, despite a significant difference in the lengths and content of these sequences. We decided to investigate the incorporation of node types into the summarisation process. We performed a series of preliminary experiments to find a way to incorporate this node information, and thus developed what we refer to as the *Switch-Summariser*. The Switch-Summariser makes use of a separately parameterised Transformer Trans_r depending on the node type of the sequence being summarised. Next, we include a separate Transformer Trans_g, which is used for every operation, regardless of node type. Thus, each node embedding operation includes a type-specific Transformer, and a global Transformer. Finally, we also found it useful to include a type embedding to the global Transformer. Formally, given a node type r, an input vector X, and a learned type embedding matrix T^r, the usage of a Switch-Transformer ST follows:

$$\text{ST}(X, r) = \text{Trans}_\text{r}(X) + \text{Trans}_\text{g}(\text{LayerNorm}(X + T^r)). \tag{3.10}$$

Here Trans_r is a type-specific Transformer, only used wrt to type r, and Trans_g is a global Transformer, used for all node types. $T^r \in \mathbb{R}^{l \times f}$ is a matrix of the same shape as X, which is created by repeating a learned embedding vector t^r l times along the sequence dimension. We used the combination of a type-specific and a global Transformer so that the type-specific Transformer could focus on extracting node type-specific information, while more general information could be extracted by the global Transformer.

3.3 GNN Graph Encoding

This thesis involves evaluating different GNN variants. Primarily we aim to compare GNNs used in typical MHQA systems [99, 31, 26, 13], against the architectural decisions made in the Transformer literature. Here, we formally describe the components that we consider using the message passing notation introduced above (see Eq 2.21, 2.22, 2.23), which describes the node state update process per the ith node.

To aid in the description of our GNN variants, we introduce the term *GNN Core*, which simply refers to the unique combination of a message and aggregate function. Thus, a GNN can be fully described by its core GNN plus its update function. We also leverage the concept of *in-out asymmetries* (see Sec 2.5.2). A GNN is in-out asymmetric if it has at least one in-out asymmetry, otherwise it is in-out symmetric. An in-out asymmetry is any function which allows the GNN to distinguish between sending out-node messages and receiving in-node current states. Practically, an in-out asymmetry can only be found in a GNN's update function, as this is where in-node states are updated. A GNN can have zero in-out asymmetries (GAT Sec 2.5.2.2), a single asymmetry (R-GCN Sec 2.5.2.5), or even multiple asymmetries (HDE GNN Sec 2.6.3.2).

3.3.1 GNN Cores

Here we list and define five distinct GNN cores, which we evaluate on the Wikihop dataset. We use the message passing notation (see Sec 2.5.1), and describe each core by providing the equations for a message and aggregate function.

3.3.1.1 *Switch-Core*

The *Switch-Core* is the core used by the HDE GNN [99] which uses distinct weights for each edge type and is defined by the equations 2.40 and 2.41. These equations describe a GNN which applies a distinct linear transformation to node states depending on the edge type being traversed. Furthermore, they describe how node messages are scaled as a function of a nodes neighbour count, and the distribution of edge types per node.

3.3.1.2 *Edge-Core*

The *Edge-Core* is a GNN core that we propose, which is based on Relative Positional Embeddings (RPE) [92]. Specifically it makes use of V-Type (see Sec 2.4.5) edge embeddings to inject edge information into a GNN. Given the edge matrix $E \in \mathbb{R}^{n \times n}$ such that E_{ij} represents the edge type between nodes i and j, we first define the edge type embeddings as

$$V = \text{EdgeTypeEmb}_V(E) \in \mathbb{R}^{n \times n \times f}, \tag{3.11}$$

where EdgeTypeEmb_V is an embedding function which maps each unique edge type to a distinct learned embedding vector. We then formally define the Edge-Core as follows:
Message:

$$\phi(\boldsymbol{h}_i^k, \boldsymbol{h}_j^k, E) = \mathrm{W_v} \boldsymbol{h}_j^k + V_{ij} \tag{3.12}$$

Aggregate:

$$\omega(M_{ij}^k, \boldsymbol{h}_i, \boldsymbol{h}_j) = \frac{M_{ij}^k}{|N(i)|} \tag{3.13}$$

Here $\mathrm{W_v}$ is a learned weight matrix being used as a linear transformation.

3.3.1.3 *SAGE-Core*

The *SAGE-Core* is another GNN core we propose which makes use of V-Type edge embeddings. Similarly to the Edge-Core, it makes use of an MLP [58] and a mean-based aggregation function. However, this core follows the GraphSAGE GNN (see Sec 2.5.2.1) more closely in its functions. Formally:

Message:

$$\phi(\boldsymbol{h}_i^k, \boldsymbol{h}_j^k, E) = \boldsymbol{h}_j^k + V_{ij} \tag{3.14}$$

Aggregate:

$$\omega(M_{ij}^k, \boldsymbol{h}_i, \boldsymbol{h}_j) = M_{ij}^k \tag{3.15}$$

This core GNN does not include any learned matrix multiplication, and thus acts only to include edge types. Thus, the bulk of the computation for a GNN using the SAGE-Core should take place in the update function.

3.3.1.4 *GAT-Core*

Here we present our modified GAT GNN core, which includes V-Type edge embeddings, which we call the *GAT-Core*. Formally:

Message:

$$\phi(\boldsymbol{h}_i^k, \boldsymbol{h}_j^k, E) = \mathrm{W_v}\boldsymbol{h}_j^k + V_{ij} \tag{3.16}$$

Aggregate:

$$\omega^k(M_{ij}^k, \boldsymbol{h}_i, \boldsymbol{h}_j) = \alpha_G(\boldsymbol{h}_i, \boldsymbol{h}_j)M_{ij}^k, \tag{3.17}$$

where α_G is an additive attention function defined by equation 2.28.

3.3.1.5 SDP-Core

Finally, we present the *SDP-Core*, which is based on Scaled Dot-Product (SDP) attention, and which makes use of K-Type (see Sec 2.4.5) edge embeddings. This GNN core is in fact identical to the RPE Transformer [92], when making use of only K-Type embeddings; however, we describe it here using the message passing notation. First, we calculate the K-Type embedding matrix similarly to the V-Type matrix:

$$K = \mathrm{EdgeTypeEmb}_K(E) \in \mathbb{R}^{n \times n \times f}, \tag{3.18}$$

then the message passing equations:

Message:

$$\phi(\boldsymbol{h}_i^k, \boldsymbol{h}_j^k, E) = \mathrm{W_v}\boldsymbol{h}_j^k \tag{3.19}$$

Aggregate:

$$\omega^k(M_{ij}^k, \boldsymbol{h}_i, \boldsymbol{h}_j) = \alpha_r(\boldsymbol{h}_i, \boldsymbol{h}_j, K)M_{ij}^k \tag{3.20}$$

Here $\alpha_r(.)$ is an SDP attention function making use of K-Type edge embeddings, and is defined by equation 2.16.

3.3.1.6 GNN-Core summary

Above, we present five distinct GNN cores, which are each defined by the combination of a message and aggregate function. Each of our five cores is capable of accepting edge features. The Switch-Core makes use of distinctly parameterised linear transformations per edge type in its message function. The other four cores all make use of edge type embeddings (see Sec 2.4.5). Of our four edge type embedding cores, two make use of attention, and two do not. SAGE-Core serves as the simplest GNN core, making no use of attention, or even any matrix multiplication. Instead SAGE-Core simply adds edge type embeddings to node messages. Edge-Core does not make use of attention either, but does make use of a parameterised matrix multiplication to linearly transform node states before being summed with an edge type embedding. Finally, we present two attention-based GNN-Cores, namely GAT-Core and SDP-Core. These two cores differ in two ways. First, they make use of distinct types of attention. GAT-Core uses additive attention [6], while SDP-Core uses SDP attention [101]. The second difference is in the manner of edge type embedding. SDP-Core uses K-Type edge embeddings by default; however, we do test the use of V-Type edge embeddings in the SDP-Core. GAT-Core, not making use of SDP attention, cannot use K-Type edge embeddings, and thus uses V-Type edge embeddings only.

3.3.2 Update Functions and In-Out Asymmetries

In our review of the GNN and Transformer literature, we have come across various update functions. We have also introduced the concept of an in-out asymmetry (see Sec 2.5.2), which is a type of function which operates in the update function. Here, we list the various

asymmetries which we consider in our GNNs, and describe them by providing an update equation. We also show how some of these asymmetries can be composed together.

3.3.2.1 No Asymmetry

An in-out symmetric GNN is one in which the GNN does not distinguish the in-nodes current state, when calculating its next state. The simplest such update function is the trivial update:

Update:

$$\gamma(\boldsymbol{h}_i^k, A_i^k) = \sigma(A_i^k), \tag{3.21}$$

where σ is a nonlinear activation function. This is the update function used by the GAT [102], making the GAT in-out symmetric.

3.3.2.2 SAGE Asymmetry

The *SAGE Asymmetry* is used in the GraphSAGE GNN [38], and contains its only linear transformation. Formally:

Update:

$$\gamma(\boldsymbol{h}_i^k, A_i^k) = \sigma(\text{Concat}_f(\boldsymbol{h}_i^k, A_i^k)\text{W}_3), \tag{3.22}$$

where $\text{W}_3 \in \mathbb{R}^{2f \times f}$ is a learned weight matrix used as a linear transformation.

3.3.2.3 MLP Asymmetry

The *MLP Asymmetry* is used in both the R-GCN [86], and the HDE GNN [99]. However, the HDE GNN uses this in combination with a second asymmetry. The MLP asymmetry follows:

Update:

$$\gamma(\boldsymbol{h}_i^k, A_i^k) = \sigma(A_i^k + \text{W}_4\boldsymbol{h}_i^k) \tag{3.23}$$

3.3.2.4 Gate Asymmetry

The *Gate Asymmetry* is used in the HDE GNN [99] alongside the MLP asymmetry. Used in isolation, it follows:

Update:

$$\gamma(\boldsymbol{h}_i^k, A_i^k) = \text{Gate}(\boldsymbol{h}_i^k, A_i^k), \tag{3.24}$$

where Gate is HDE GNN's gating function [99] and is defined by equation 2.37.

3.3.2.5 Transformer Update Function Asymmetry

The *TUF Asymmetry* is used by the Transformer model [101], but is named as such by us. It follows:

Update:

$$\gamma(\boldsymbol{h}_i^k, A_i^k) = \text{TUF}(\boldsymbol{h}_i^k, A_i^k) \tag{3.25}$$

Here TUF is the Transformer Update Function (TUF) defined by equation 2.11.

3.3.2.6 Composing Asymmetries

The HDE GNN [99] composes the MLP asymmetry with the Gate asymmetry as follows:

Update:

$$\gamma(\boldsymbol{h}_i^k, A_i^k) = \text{Gate}(\boldsymbol{h}_i^k, A_i^k + \text{W}_4\boldsymbol{h}_i^k) \tag{3.26}$$

This composition can be represented by the notation $Gate(MLP)$. We can also arbitrarily compose any of these defined asymmetries in any order. We do however stick to a single well defined order which follows:

$$Gate(TUF(MLP))$$

when using the MLP asymmetry, and

$$Gate(TUF(SAGE))$$

when using the SAGE asymmetry. We do not always include all asymmetries; however, we respect this order in all cases. Removing the TUF would for example result in

$$Gate(SAGE)$$

when using the SAGE asymmetry. In this work we do not consider composing the MLP asymmetry with the SAGE asymmetry.

Any given in-out asymmetry has the form: $Asym(x, A) = y$. Composing asymmetries together then simply replaces the A input with the output of an inner asymmetry, or more formally:

$$Composition(x, A) = Asym^2(x, Asym^1(x, A)). \tag{3.27}$$

Finally, we take care to remove any intermediate nonlinear activation functions when composing asymmetries.

3.4 Candidate Selection Output Model

Once the GNN has completed the graph encoding step, the outputted node vectors are ready for the answer classification. Note that the graph contains nodes for both candidates and entities, where most candidate nodes have at least one, but sometimes multiple corresponding entity nodes. We follow the HDE model [99] in using two distinct two-layer MLPs labeled MLP_ϵ, and MLP_c to score each of the entity nodes and candidate nodes, respectively. The hidden dimension of these MLPs is half the model dimension f, and the output dimension is 1.

Candidates are scored using equation 2.62, as a function of the final representation of candidate and entity nodes. Once all candidates in C_q have been scored, these candidate scores are normalised using a softmax, producing a final probability distribution over all the candidate nodes. This probability distribution can then be trained to predict the correct candidate by minimising the CrossEntropy loss wrt the correct answer.

Chapter 4

Experimental Method

In this chapter we detail how we implemented our Graph Neural Network (GNN)-based Multihop Question Answering (MHQA) system, including how we preprocess the Wikihop dataset [106] into context graphs. We present an evaluation on the memory usage of two competing approaches to the development of GNN architectures which we refer to as the *Dense* and the *Sparse* implementation approaches. We describe our hyperparameter tuning method for finding reasonable model parameters such as learning rate [89]. We then describe the method by which we conducted our experiments, including detailing three successive benchmark models which we develop. We use these benchmark models to compare various feature variations in GNN architecture, context graph construction, and graph embedding. Finally we detail how we evaluate these models using the Wikihop dataset, and provide a brief investigation into the variation in model performance which is due to random chance.

4.1 Implementation

All of our code is written in python, our configuration files are written in JSON[7], and our DNNs are all implemented in Pytorch.[8] Most of our GNNs make use of Pytorch Geometric[9] [34], a library extending Pytorch for GNNs.

The main aim of our model implementation was to be highly configurable since GNN and Transformer literatures present us with many modeling options at all stages of the model pipeline. To perform GNN operations, we make use of Pytorch Geometric. Pytorch Geometric offers an intuitive implementation of the message passing pattern for GNNs. Furthermore, it also offers a set of GNN implementations from various noteworthy GNN papers such as GraphSAGE [38] and the GAT [102]. The downside of Pytorch Geometric is that is does not readily support modular and swappable components. Our solution was to build a wrapper around Pytorch Geometric which was able to overcome some of these limitations.

We make use of our wrapper system to implement GNN Update functions (see Sec 2.5.1) instead of Pytorch Geometric. This allows us to easily swap and compose various In-Out Asymmetries (see Sec 2.5.2) such as Gating [93] (see Eq 2.37) and the TUF (see Eq 2.11). We implement four of our five GNN Cores (see Sec 3.3) using Pytorch Geometric, while the GAT-Core (see Sec 3.3.1.4) is implemented for us in Pytorch Geometric already.[10] Pytorch

[7] JSON Docs: https://www.json.org/json-en.html
[8] Pytorch Docs: https://pytorch.org/
[9] Pytorch Geometric Docs: https://pytorch-geometric.readthedocs.io/en/latest/
[10] GAT implementation: https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#torch_geometric.nn.conv.GATConv

Geometric does in fact include an implementation for the GraphSAGE [38] GNN, however inspecting their code reveals that it is not a faithful recreation, since it makes use of the MLP-Asymmetry (see Sec 3.3.2.3) instead of the SAGE-Asymmetry (see Sec 3.3.2.2). Thus, we implemented our own SAGE-Core (see Sec 3.3.1.3) and SAGE-Asymmetry using Pytorch Geometric.

We also use our wrapper system to include edge information. Our system can either inject edge embeddings into GNNs, or otherwise it is able to turn any of our GNNs into a switch GNN. The implemented system is able to construct a GNN-based MHQA model which is defined entirely by a single JSON configuration file, and is able to handle almost any reasonable combination of supported features and hyperparameters.

When making use of a Scaled Dot-Product (SDP) attention-based GNN (see Sec 2.5.2.3), instead of using Pytorch Geometric we found it easier to simply modify the Pytorch native implementation of SDP attention, found in its implementation of the Transformer model.[6] Thus our SDP-Core (see Sec 3.3.1.5) does not make use of Pytorch Geometric, instead it is implemented directly in Pytorch. Conceptually, these SDP models still undergo message passing, however the actual implementation we use differs substantially (see Sec 4.1.2). In either case, we are able to make use of our GNN wrapper system to easily include either Pytorch Geometric-based GNNs, or an SDP GNN. Our models make use of different numbers of trainable parameters ranging from 3 to 15 million parameters.

When representing an attention-based GNN in the message passing notation (see Sec 2.5.1), the notation calls for the usage of the attention function $\alpha(.)$ for every pair of connected nodes i and j (see Eq 3.17 and 3.20). However the attention function itself contains a softmax, which loops over all out-nodes j (see Eq 2.7). This appears to result in a time complexity of $O(n^3)$ for n nodes. In practice however, the code implementation of these models make use of caching in order to prevent these redundant operations, maintaining the time complexity of $O(n^2)$. As such, this is more an artifact of the notation than a limitation of an attention-based GNN.

The BERT model we use[11] is a reduced size version of the original BERT which was trained using knowledge distillation [42]. This model can be found on HuggingFace[4], a Language Model repository. Knowledge distillation is a model compression technique whereby the outputs of a large pretrained model are used to train a smaller version of the model on the same data [42, 100]. To distil the knowledge of a large BERT model into a smaller version, the small model was first pretrained using a standard masked language modeling task, and then fine-tuned on the outputs of the large BERT model [100].

4.1.1 Data Processing

To perform GNN encoding on a Wikihop data point, we need to project the data point into a vector space containing nodes with a fixed number of numeric features (see Sec 3.2). Some parts of this process are learned, meaning they involve the use of trainable parameters which change during model training. Other parts of this process are fixed, meaning they do not change as the model trains. To save time during model training, we aim to perform the fixed graph embedding steps only once, saving the results to storage for later retrieval. When a process is performed once and its results are saved, we call the process offline, since the process does not need to be used during model training. Conversely, when a process must happen during model training, we refer to this as an online process. Learned model processes must take place online, since the outputs change as the model parameters

[11] Knowledge Distillation BERT: `https://huggingface.co/prajjwal1/bert-medium`

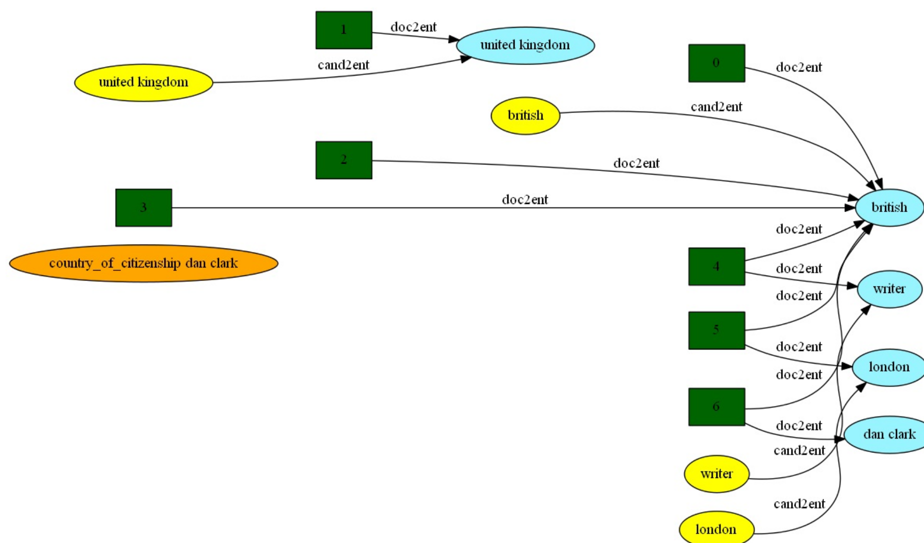

FIGURE 4.1: A reduced visualisation of a graph constructed by our implemented system. Here the orange node is the query, blue nodes are entities, yellow are answer candidates, and green are document nodes. To allow the graph to be readable, we have removed some candidate nodes, as well as aggregated all same-text entities into single nodes. We also omit some edge types, such as candidate to candidate edges, and entity to entity edges. The query node is only included here for the benefit of the reader, and is not actually included in any context graphs.

update. Furthermore, Deep Neural Networks [60] train via backpropagation [84], which requires that the network be differentiable from start to end.

Using the processes described in section 3.1, we use Wikihop data points to construct context graphs in an offline manner, saving the results for late use. Certain model variations result in distinct graph structures, such as the use of Special vs Detected entities ((see Sec 2.6.1)). We thus save a distinct set of graphs for each configuration which describes a different graph construction process. The graphs we save offline consist of nodes and edges. The nodes each represent a token span in a given token sequence. Different token embedders (see Sec 2.6.2) need to tokenise text differently, such as GloVe's word-level tokeniser [74], and BERT's [27] subword-level BPE-based [90] tokeniser. We thus also need to save distinct graphs when making use of token embedders which tokenise text differently. To calculate the token spans that nodes represent, we need to tokenise text using either GloVe [74] or BERT's [27] tokeniser, so that we can align text with tokens. Once we have tokenised the text, and aligned token spans with nodes, we save the tokens and the token spans for later use. The method of aligning token spans with node text differs depending on the tokeniser used. Implementations of BERT's [27] tokeniser which make use of the Huggingface library[4] provide an explicit method for this alignment, while for GloVe, we designed this alignment logic ourselves.

Most of our models evaluated make use of Special entities and do not include sentence nodes. For the variations which require detected entities or sentence nodes, we make use of Spacy[12], an NLP library which offers both sentence splitting and entity detection, among

[12] Spacy Docs: https://spacy.io/

many other natural language features. Spacy can provide us with the start and end character positions for entities and sentences. In other words, Spacy can predict character spans given text as input. However, our models operate on tokens not characters, as such we must convert these character spans into token spans. The method of conversion differs for different tokenisers, but generally involves finding the token spans which overlap with the detected character spans.

The result of the data pre-processing is a set of graphs made of nodes and edges which are saved to storage. The nodes have a type, such as 'Entity', so too do the edges, such as 'Comention'. The nodes also have a token span, and a reference to their source text, such as 'Document_3' or 'Candidate_5'. Figure 4.1 shows a simplified graph created by our system using a real Wikihop data point. To use these graphs in model training, we load them from disk during every training run. The Wikihop text is also loaded and tokenised. The Wikihop tokens are then embedded using either the Glove [74] or BERT [27] token embedder. Finally, the token spans in each graph node are aligned with the token embeddings just created. This allows each node to correspond to a particular subsequence of token embedding vectors. From here, the model continues as described in chapter 3, which generally follows: context-query coattention, node summarisation, GNN encoding, and output prediction.

4.1.2 Sparse and Dense GNN Implementations

In our work we consider two different attention-based GNN Cores, namely the GAT-Core (see Sec 3.3.1.4) and the SDP-Core (see Sec 3.3.1.5). These are two similar algorithms which each make use of an attention mechanism, and could each be implemented in many different ways. The implementations we use for the GAT-Core and the SDP-Core are very different. We name the implementation approach shared by our SDP-Core and the Transformer as the *Dense* approach. The rest of our GNNs are implemented in Pytorch Geometric and share a common implementation approach which we name the *Sparse* approach. Here we describe and contrast the Dense and Sparse approaches by comparing our GAT-Core and SDP-Core, however the distinctions between these two implementation approaches are more general than these specific GNN algorithms. This means that either the GAT-Core or SDP-Core could be implemented in either the Dense or Sparse approach.

Attention-based GNNs are tasked with communicating information around a set of elements, called nodes in GNN literature, and called tokens in Transformer literature. This involves computing a compatibility function between elements of a matrix $X \in \mathbb{R}^{n \times f}$ containing a sequence of n element vectors, with f feature dimensions each. These compatibility values are either calculated via scaled dot products (see eq 2.5) or via an MLP-based scoring function (see eq 2.27). Once calculated, these compatibility values are normalised using a softmax function to yield attention scores. The Dense implementation approach involves the creation of a matrix $D \in \mathbb{R}^{n \times n}$ of compatibility values, representing the interactions of all n^2 element pairs. Applying softmax to D yields the attention scores which our GNNs use to perform self attention. When one does not wish to allow communication between all elements in a sequence, one must mask out the elements of D which represent the non-communicating element pairs. This is to say that all possible element pairs are represented in D, regardless of whether they are connected or not. In this sense, sparsity is a post hoc modification to the Dense attention mechanism.

In contrast to the Dense approach, the Sparse approach taken by Pytorch Geometric avoids the need for the construction of the full matrix D. Instead, the sparse approach operates edgewise, meaning it stores its compatibility values in a matrix $S \in \mathbb{R}^{|E|}$, where $|E|$ is the number of edges in the graph, or rather the number of sequence elements which should communicate. This is to say that for a fully connected graph with n nodes $|E| = n^2$, since

every element pair should communicate. For a sparsely connected graph where $|E| << n^2$ the number of zeros in the full Dense matrix D would be larger than the number of non-zero elements. This means that not only were many compatibility values calculated redundantly, but the matrix D is also unnecessarily consuming memory by storing these zeros. Conversely, to calculate S, the Sparse approach first creates the matrices $S_x^i \in \mathbb{R}^{|E|\times f}$ and $S_x^j \in \mathbb{R}^{|E|\times f}$. S_x^i represents all the in-node (see Sec 2.5.1) vector states for each edge, while S_x^j represents all the out-node states. These matrices are constructed by simply duplicating and rearranging the vector elements of X. Together, S_x^i and S_x^j represent all element state pairs which should communicate, with no masking required. Performing either a scaled dot-product or an MLP-based scoring function between S_x^i and S_x^j yields the compatibility scores matrix $S \in \mathbb{R}^{|E|}$. Performing softmax over the matrix S requires a specially implemented sparse version of the softmax function, which is provided in the Pytorch Geometric library. Overall, the memory requirements of Sparse approach should reduce as the sparsity of the graph increases. However, due to the construction of the two $|E| \times f$ matrices, this approach creates a bottleneck when $|E|$ is large, or when f is large.

4.2 Hyperparameters

Hyperparameter	Value
Num Training Epochs	30
Learning rate (LR)	**0.001** when weight sharing, else **0.01**
LR Schedule Exponential Decay	0.9
Dropout	0.1
Batch Size	1
Num GNN layers	9
Model Dimension	**300** for GloVe models, **512** for BERT models
Num Summariser Heads	6
Num GNN Heads	**4** if attention-based

TABLE 4.1: The numerical hyperparameters used in our models for all of our formal experiments.

Table 4.1 shows the numerical hyperparameters used in all of our formal experiments. Informally we found that using distinct weights in a GNN's layers required/allowed the use of a higher learning rate. We also make use of a batch size of 1, which is not typical of a modern DNN [53, 72], and likely contributes to a lower model performance. This was done due to simplicity of implementation reasons, as implementing true batching is difficult with Pytorch Geometric, and in combination with other modules, would have increased code complexity significantly. We did investigate making use of gradient accumulation in our training process, which allows for the effective batching of model parameter updates without needing to modify model code. Practically, this involves avoiding a model parameter update for a set number of gradient accumulation steps, thus allowing loss gradients to be accumulated over multiple Wikihop examples. However, this did not lead to a model performance increase, and thus we decided to simply retain a batch size of 1 indiscriminately.

We note that the dropout value that we use (0.1) is lower than is typical, with some BERT implementations having a dropout value of 0.5 as default. However our preliminary testing found this to be optimal under our training regime. Both our GAT-based and SDP-based attention GNNs make use of 4 attention heads. Next, our model dimension is determined by the dimensionality of the token embedder used, and thus follows either 300 for GloVe-based

Hyperparameter	Options
GNN Core	Edge-Core, SAGE-Core, Switch-Core, GAT-Core, SDP-Core
Optimiser	**SGD**, Adam, AdamW
Summariser	Transformer, Switch Summariser
Entity Detection	Special, Detected, Both
Bidirectional Edge Types	True, False
Edge Information	Switch GNN, K-Type, V-Type embeddings
HDE Style GRUs [99]	True, False
Gating in Update Function	True, False
TUF in Update Function	True, False
Additional In-Out Asymmetry	MLP-Asymmetry, SAGE-Asymmetry, None
Layer Weight Sharing	True, False
GNN Core Weight Sharing	True, False
Gating Weight Sharing	True, False
TUF Weight Sharing	True, False
Token Embedder Type	Glove, BERT
Fine-Tune BERT Embedder	True, False

TABLE 4.2: The feature options available in our system.

models, and 512 for BERT-based models. It is possible to decouple the token embedding dimension from the model dimension by simply mapping token embedding dimensions up or down with an MLP, however we decided against this. Finally, we make use of a learning rate scheduler which follows an exponential decay. This is to say that the effective learning rate at epoch n is $LR \times 0.9^n$. This results in a learning rate which is 20 times lower by epoch 30 than it is at epoch 1.

Table 4.2 holds the full list of all of the feature options that our system implements. Our preliminary experiments showed that that SGD outperformed Adam in a number of our models, and thus we make use of SGD in all of our formal experiments. K-Type edge embeddings can only be used in combination with an SDP-Core, while V-Type embeddings can be used with any GNN Core. When fine-tuning BERT, we found it useful to freeze the BERT parameters for the first 5 epochs of training. This protects the information contained in the pretrained BERT from being destroyed during the early parameter updates [1], when the GNN model built on top of BERT is still noisy and inaccurate. We test the usage of weight sharing in our GNNs, as such we have the option to share all the GNN weights between layers, or to share individual component parameters between layers, such as only sharing the Gating parameters.

4.3 Experimental Procedure

Our experiments took place in three stages. First prototyping to develop a working model, next an preliminary phase to develop and test features, and finally a formal data collection phase. To begin with, we needed to build our configurable model system, and evaluate it on Wikihop. As a starting point we implemented a simplified version of the HDE model. We then began the preliminary phase where we tested various features and hyperparameters. These preliminary experiments were used to gauge the effectiveness of features such as graph attention, edge embeddings and weight sharing.

By the end of the exploratory phase we had a good indication of which feature combinations were synergistic, and which were redundant or harmful. The main thing distinguishing

Model Name	Type	Granularity	Year	Development Accuracy	Test Accuracy
BiDAF [91]	Attention	Token	2017	-	42.9
Coref-GRU [29]	RNN	Token	2018	56.0	59.3
GSPR [95]	GNN	Node	2018	62.8	65.4
EGCN [26]	GNN	Node	2018	65.3	66.4
CFC [116]	Attention	Token	2019	66.4	70.6
HDE [99]	GNN	Node	2019	68.1	70.9
BERT-Base (ours)	GNN	Node	2022	69	.
Reasoning Chains [15]	Transformer	Token	2021	72.2	76.5
BigBird-ETC [114]	Sparse Transformer	Token	2020	75.9	82.3

TABLE 4.3: Results of various noteworthy models on the Wikihop dataset. Here the Development set is publicly available test data, while the Test set is hidden to the public. We were not able to evaluate our models on the Test set due to technical difficulties with the evaluation system.

our preliminary experiments is the lack of our standardised model benchmarks. While the data was useful to inform design decisions, it was not structured enough to present formally. To present our results formally we defined three successive benchmark models: HDE-Base, Att-Base and BERT-Base, from which we would test feature variations. We detail the motivations behind this exact choice of benchmark configurations, as well as some more details in our Benchmark Model section 4.4.

4.4 Benchmark Models

The primary aim of the paper is to study certain methods used by Transformers and GNNs in the context of NLP. To this end, we use the Wikihop dataset, a challenging MHQA dataset, evaluated on frequently by Transformer and GNN solutions alike. The Transformer variant named the RealFormer[41] is currently SOTA on Wikihop with a test-set accuracy of 84.4%. Table 4.3 shows the performance of various noteworthy models on the Wikihop Dataset. These Transformer-only solutions leverage large compute and few structural biases to obtain high performance by operating on token level features. Separately, GNN-based solutions offer a lower compute alternative, by compressing the high-dimensional token features into node features, which represent text at a higher granularity. HDE [99] is the highest performing GNN-based approach to Wikihop at the time of this writing, scoring 70.9% on the hidden test-set, and 68.1% on the development test set. When ensembled, HDE [99] scores 74.4% on the hidden test-set, which despite being a high score on the challenging dataset, still falls far short of the best Transformer-only solutions, many of which lie in the 80+% range [41, 8, 114]. We thus use HDE as a starting point to discover how GNN performance can be increased by introducing Transformer-inspired techniques into our GNN-based MHQA system, while still retaining the node level processing. In this section, we describe the three successive models we use as benchmarks to evaluate feature variations. We evaluate all our model variations using the development-set accuracy of the Wikihop dataset. We summarise the diverging architectural choices for each of our benchmark models in Table 4.4.

Model Name	HDE [99]	HDE-Base	Att-Base	BERT-Base
Token Embeddings	Glove(300) + n-gram(100)	Glove(300)	Glove(300)	BERT(512)
Coattention	Zhong et al (2019)[116]	Transformer	Transformer	Transformer
Summariser	Zhong et al (2019)[116]	Transformer	Switch Transformer	Transformer
Core GNN	MLP	MLP	GAT	GAT
Edge Info	Switch GNN	V-Type Embeddings	V-Type Embeddings	V-Type Embeddings
Weight Sharing	Yes	Yes	No	No
In-Out Asymmetries	Gating	Gating	Gating + TUF	Gating + TUF

TABLE 4.4: Architectures. A summary of the different models we evaluate against Wikihop, as well as the original HDE. [101]. The parameters shown here are only those which vary between our three benchmark models. It should be noted that our models labeled "SDP" for Scaled Dot-Product make use of K-Type edge embeddings, instead of our other models which make use of V-Type embeddings.

4.4.1 HDE-Base

We begin by describing a model which is similar to HDE, albeit with some key difference which we discuss in our Model section (see Sec 3). We label this model *HDE-Base*, and note that this is the model which we use as a baseline result on Wikihop. HDE-Base makes use of the Transformer-based coattention and summarisation processes described in section 3.2 in order to embed graphs. It also makes use of a simplified version of the HDE GNN [99]. Specifically we remove the MLP-Asymmetry (see Sec 3.3.2.3), and we included edge embeddings instead of using a switch GNN, resulting in the Edge-Core GNN (see Sec 3.3.1.2). The HDE-Base models also make use of the Gate Asymmetry (see Sec 3.3.2.4). As HDE [99] does, our HDE-Base models share all weights across their GNN layers. The purpose of the HDE-Base model is to use as a starting point which resembles existing GNN-based MHQA models, and to test various features which are common in the GNN literature such as the Switch-Core (see Sec 3.3.1.1) proposed by the R-GCN [86]. From here, we describe our two subsequent benchmark models which are used to test different features, and which offer successive improvements over HDE-Base.

4.4.2 Att-Base

We label a second model *Att-Base*, which differs from HDE-Base mainly in its GNN architecture, and its use of distinct model parameters per layer. The purpose of Att-Base is to offer a convenient benchmark to test how Transformer-inspired features work in conjunction with our MHQA system. Our Att-Base model differs from our HDE-Base model by making use of a Graph Attention Neural Network (GAT) [102] by way of the GAT-Core (see Sec 3.3.1.4), instead of the Edge-Core used by HDE-Base. It also contains Transformer Update Function (TUF) Asymmetry (see Sec 3.3.2.5). These two features make the GNN used in Att-Base much more Transformer-like, with an exception being the GAT's use of additive attention

[6] instead of scaled dot product attention [101] as is used in the Transformer. We do however evaluate a version of our Att-Base model with scaled dot-product attention which we label *Att-Base SDP*. Att-Base also includes a gating mechanism, which is not common in Transformer literature [101, 8, 41, 114, 55, 19], however it is common in GNN literature [99, 31, 13, 26, 95]. Finally, our Att-Base model also includes a module which we have named the Switch-Summariser (see Sec 3.2.4). Our preliminary testing indicated that this feature positively affected performance, however our formal testing has shown this feature to have no effect on model performance. Given the time, we would remove the Switch-Summariser entirely from our models and results, however this would require a costly rerun of our Att-Base ablation tests, which is infeasible given time constraints.

4.4.3 BERT-Base

Building on our Att-Base model, we define a third benchmark model called *BERT-Base*, which differs from Att-Base mainly in its use of a BERT-based [27] token embedder instead of Att-Base's Glove-based embedder. Since the BERT embedder is of a higher dimensionality than Glove, the model dimension of our BERT-Base is 512 features, more than the 300 features of our Glove-based models, and more than the 400 features in HDE [99]. We also took the opportunity to remove the Switch-Summariser from this benchmark model. We use this benchmark model to formally compare the usage of Special Entities against Detected Entities, as well as other graph structuring variations such as the inclusion of sentence nodes (see Sec 5.3.1). We also evaluate an SDP version of our BERT-Base called *BERT-Base SDP*

4.5 Evaluation Method

Wikihop is a multiple choice question answering dataset (see Sec 2.2.3), meaning each data point contains a question q, a set of possible answer candidates C_q, and an answer to the question $a_q \in C_q$. Thus the task of any model evaluating on Wikihop is simply to predict which candidate represents the correct answer. This makes it simple to determine the accuracy of such a model. Given a set of questions $Q = \{q_1, ..., q_n\}$ with correct answers $A = \{a_1, ..., a_n\}$, predict an answer to each question $A' = \{a'_1, ..., a'_n\}$. Then simply count the number of correct answer predictions as follows:

$$Correct = \underset{i=1}{\overset{n}{\text{Count}}}(a_i = a'_i), \tag{4.1}$$

$$Accuracy = \frac{Correct}{n} \tag{4.2}$$

The Wikihop dataset is split into three sets, namely the training set, development set and test set. For all of our evaluations and ablations we make use of the development set accuracy. The test set is hidden, meaning the public cannot access the answers, only the questions. Thus to evaluate on the hidden test set, one must submit their model to the Wikihop evaluation portal.[13] We were unable to evaluate our models on the test set due to incompatible software versions used by our system and the outdated evaluation system offered for Wikihop.

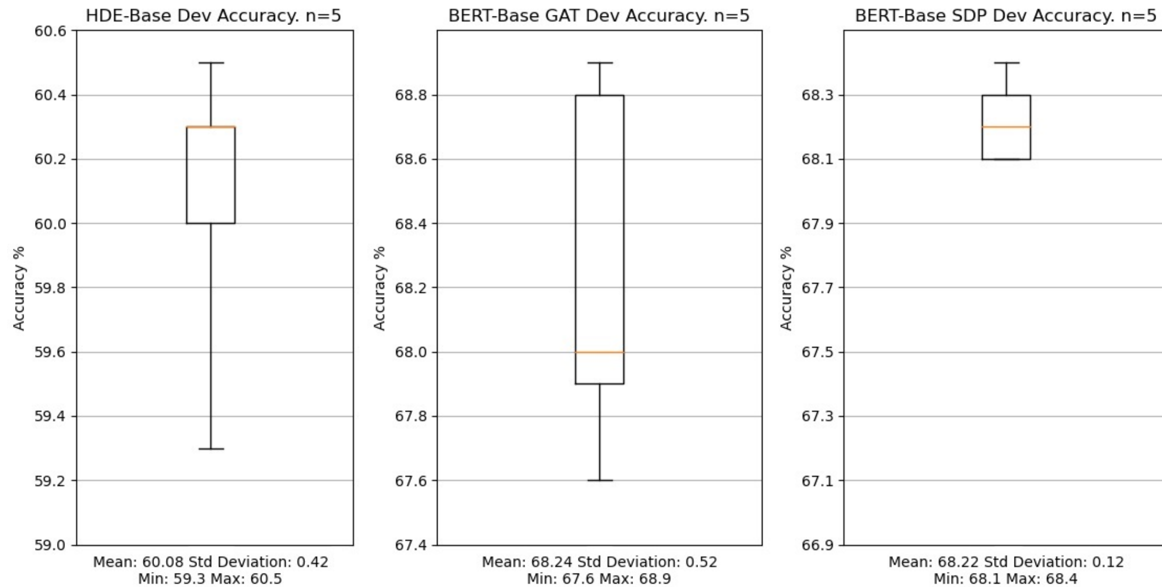

FIGURE 4.2: Box and whisker plots showing the development set accuracy of our HDE-Base and BERT-Base models. Each model was independently trained 5 times on the Wikihop Dataset. The plots show the distribution of development set accuracies for different model runs. We use this to inform our estimate of the standard deviation of our other model configurations. The plots share the same y-axis scale to easily compare variations.

4.6 Model Evaluation Variation

Here, we investigate how the variation of performance in our models is influenced by the random initialisation of their parameters/weights. We begin by choosing three model configurations, and training each configuration 5 times independently using the same training procedure. Namely, we chose our HDE-Base model, and our BERT-Base model with GAT and SDP-based attention. We do this in order to gauge the development set accuracy standard deviation of our model runs. BERT-Base with SDP attention has a very low accuracy standard deviation of 0.12%, while the other two models have similar deviations, each around 0.5%. Figure 4.2 shows the development set accuracy distributions of our three sampled models. We use these variation results to infer the standard deviation of the rest of our model evaluation results, as we cannot directly compute these values due to time constraints. As such, accuracy variations of less than half of a percent may not be significant, while variations of more than 1% may indicate significance.

4.7 Test Fairness

A fair test is one which controls all variables except the variable being tested. Performing a fair test to compare DNN architecture variations is difficult, largely due to the many dependent variables present. These include the number of layers or hidden dimensions in the network, the number of parameters, the number of floating point operations the network requires, the amount of data used to train the model, the number of epochs during training, or even the memory requirements of the model. Beyond this, DNNs are typically randomly instantiated every run, which can lead to different performances of even the exact same

[13] Wikihop Evaluation: `https://worksheets.codalab.org/worksheets/0x9acb78d24d454203ae197439130def65/`

model. Due to time and computational restraints however, we make use of n=1 tests to compare variations. To get an understanding of the degree of variation caused by random initialisation, we picked three model configurations to run 5 times (see Sec 4.6), to get a notion of the standard deviation in model accuracy. Beyond this, we ensured that the order of data loaded into the models was the same between runs.

Performing a fair test where the number of parameters and operations are equal between two model variants is infeasible, as controlling the number of layers and hidden dimension of the model only gives a rudimentary control over parameters and operations. Furthermore, weight sharing can change the number of parameters without changing the number of operations. Thus, in our work, we neglect the influence of model parameters and floating point operations when comparing model variations, however we do investigate the effect of model parameters on model performance (see Sec 5.5). When comparing the use of BERT and Glove, a fair test is unreasonable, as our model's hidden dimensions change significantly, and due to the large size of BERT, the number of operations increases even further. Thus, we make no attempt to formally demonstrate the superiority of BERT over Glove, and instead defer to common knowledge about the general increased performance of BERT over Glove in a range of tasks [27].

Instead of trying to control for parameters and operations, we opted to try and maintain a consistent set of hyperparameters over all of our experiments. As such, the number of model layers, number of attention heads, number of training epochs, and learning rate schedule are constant across all of our experiments. We make use of two learning rates, a low rate for models using weight sharing, and a high rate for models with distinct weights per layer. We also decided to train all of our models for a fixed number of epochs, which was enough for most of our models to converge. However there may be small deviations in our results due to models which plateaued, but may have continued improving given more training time.

As final thought on test fairness, consider the degree of hyperparameter tuning done for different models. If one model has undergone more hyperparameter tuning, it is likely to outperform a counterpart which had less time given to hyperparameters. This is especially relevant when performing ablation studies. All of our hyperparameter tuning was performed during our preliminary testing, before we had defined our benchmark models. This means our benchmark models were not tuned more rigorously than were model variations. Nevertheless, it was not feasible for us to completely control for this variable, thus we cannot rule out the possibility that some of our conclusions are due to our hyperparameters, and not the features in question.

4.8 Data Analysis

Here we offer an analysis into the properties of the Wikihop [106] dataset, as well as the graphs constructed by our system.

4.8.1 Wikihop Data Points

We begin by analysing the size of each Wikihop data point, measured by the number of documents given per data point, as well as the lengths of these documents. We measure the lengths of documents in terms of the number of tokens as tokenised by BPE (see Sec 2.6.2.1) for use in our BERT-Base (see Sec 4.4.3) model. Figure 4.3 demonstrates a key difficulty in modeling the Wikihop dataset - the large variation in the sizes of data points. Even excluding statistical outliers, the total number of tokens per Wikihop data point still ranges

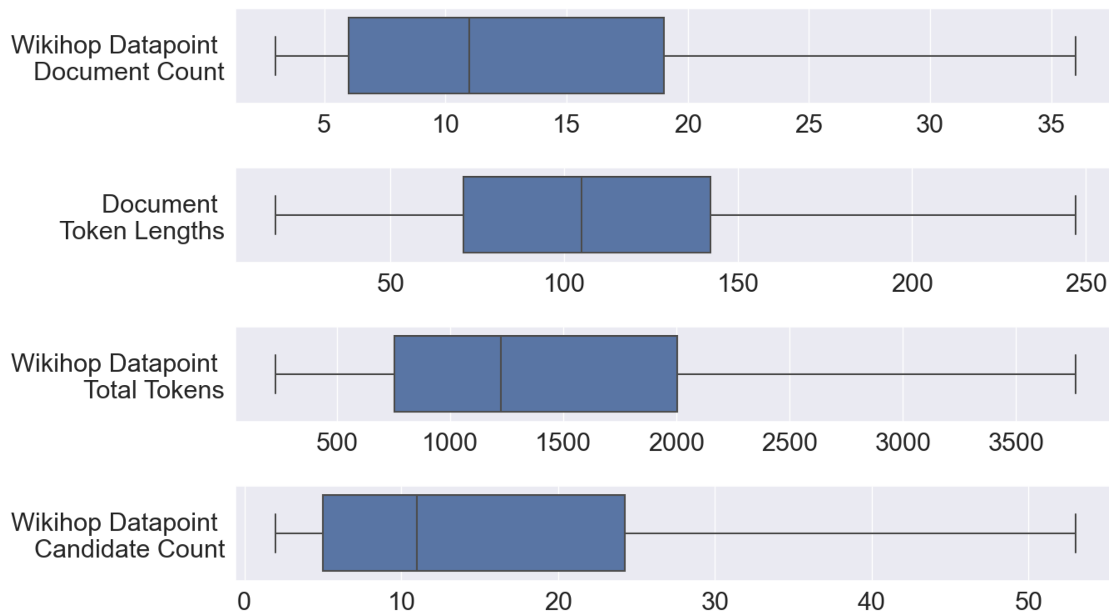

FIGURE 4.3: A figure containing box and whisker plots, demonstrating the distributions of three measures of Wikihop data point size. Statistical outliers have been omitted from these plots. The figure demonstrates: a) The number of documents per Wikihop data point; b) The number of tokens per document over the entire Wikihop dataset; c) The sum number of tokens per Wikihop data point over each document; and d) the number of answer candidates per data point.

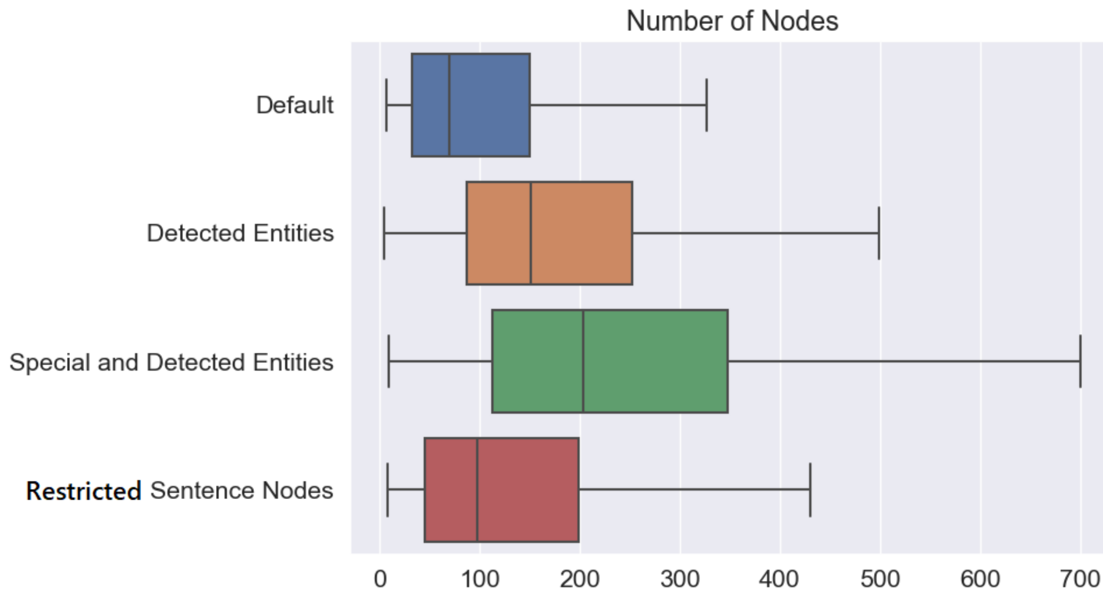

FIGURE 4.4: A figure demonstrating the distributions in the number of nodes in graphs created by the entire Wikihop training set. Included is the distribution for our default graphs, which make use of Special entities. Also included are some key graph structuring variations which affect node counts.

from under 100, to almost 4000. For reference, the largest sequence of tokens that BERT can encode is only 512, below the lower quartile value for the total number of tokens. This means that relying entirely on a full self-attention (see Sec 2.4.1) Transformer such as BERT [27] to model the Wikihop dataset is not feasible. However, Sparse Transformers (see Sec 2.4.4) such as Bigbird [114] or the Longformer [8] are able to process around 8 times as many tokens (4096) [114], thus these Sparse Transformers are able to fully encode even the longer Wikihop data points. However, processing yet longer context sequences may be out of reach of even the Sparse Transformers, motivating the continued study of GNN-based approaches.

Figure 4.3 shows us that the upper bound for the token length of any individual document is under 250. This means that each individual document can easily be encoded by a full self-attention Transformer such as BERT. This makes using pretrained Transformer models as document encoders feasible for Wikihop.

4.8.2 Constructed Graphs

We investigate the properties of our default graph construction process, which is defined by the use of Candidate, Entity, and Document nodes only, as well as the use of our five fundamental edge types (see Sec 3.1). We also investigate the properties of some of our key graph structuring variants.

To investigate these graphs, we use three distinct metrics, namely *Edge Density*, *Cross Document Ratio*, and the number of nodes. Edge density is a standard metric measuring the number of edges present in a graph as compared to the number of possible edges in the graph. For a graph where nodes are allowed to connect to themselves, the number of possible edges is simply n^2 for n nodes. Thus, edge density is simply $\frac{|E|}{n^2}$. Cross Document Ratio

(CDR) is a metric we define as the number of edges which directly connect nodes from distinct documents, divided by the number of distinct documents. Central to this metric is the notion that Entity and Sentence nodes naturally belong to particular documents. Thus, when two entities are connected who each belong to distinct documents, then that edge is considered a cross document edge. Since Candidate nodes do not belong to documents, they are excluded from this metric. Any link between a Document node and a node which it does not naturally contain is also considered a cross document edge. The formula is then just

$$CDR = \frac{\text{Count}(CrossDocumentEdges)}{\text{Count}(DocumentNodes)}. \tag{4.3}$$

Figure 4.4 demonstrates that using Detected entities instead of Special entities will increase the number of entities used.

Figure 4.5 illustrates the distribution of edge density for our default graphs, as well as graph variants which include Co-mention edges, and Co-document edges (see Sec 3.1). The figure also includes statistics on a graph structuring variant which omits the use of the Co-mention edge. Figure 4.5 indicates that the majority of graphs constructed by our default method result in edge densities which are less than $\frac{1}{10}$. We can see from figure 4.5 that the addition of Co-document edges slightly increases edge densities. Removing the Co-mention edge serves to slightly decrease edge densities, resulting in more sparse graphs. Finally figure 4.5 shows us that detected entities are much less connected than special entities, likely due to a lack of exact text matches with answer candidates.

Regarding CDR, we can see from figure 4.6 that the addition of Co-document links has no effect on the ratio. This is expected since the Co-document edge only connects Entity nodes which are found in the same document. Removing the Co-mention edge results in a CDR of zero for all graphs. Removing the Co-mention edge does not result in disconnected graphs however, since all Candidate nodes are connected (see Sec 3.1), thus, distinct Document nodes can communicate indirectly via common Candidate nodes.

Figures 4.5 and 4.6 also demonstrate the effect of using different node types on edge density and CDR. We can see that the inclusion of sentence nodes and their associated edges greatly increases CDR. We can also see from figure 4.4 that using Detected Entities instead of Special Entities (see Sec 2.6) serves to decrease edge densities, which indicates that there are fewer Co-mention edges created when using Detected entities.

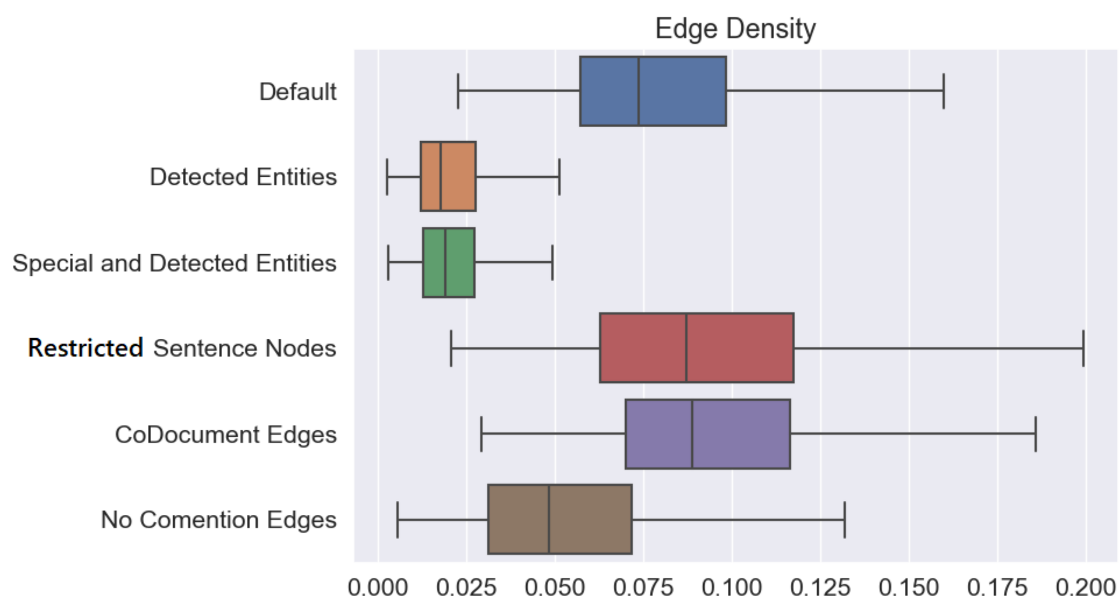

FIGURE 4.5: A figure demonstrating the distributions in edge densities of graphs created by the entire Wikihop training set. Included is the distribution for our default graphs, as well as some key graph structuring variations.

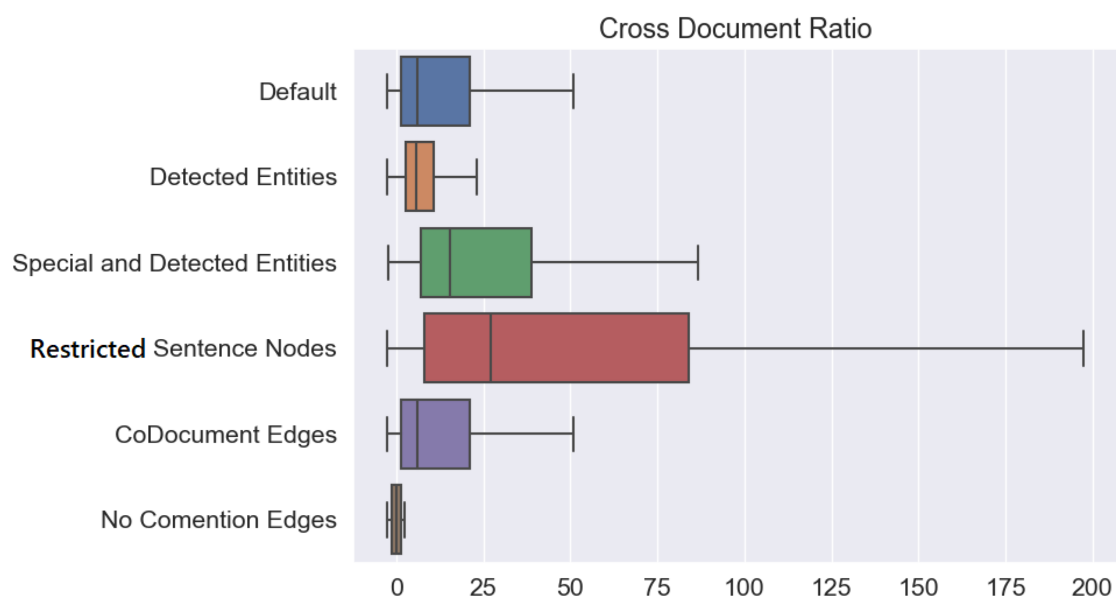

FIGURE 4.6: A figure demonstrating the distributions in CDR for graphs created by the entire Wikihop training set. Included is the distribution for our default graphs, as well as some key graph structuring variations.

Chapter 5

Experiment Results

In this chapter we structure the model performance data we have collected. First we present our main results for our benchmark models, then we go on to provide more detailed results for our ablation tests. We use these ablation results to compare various features at the graph structuring, graph embedding, and Graph Neural Network (GNN) encoding phases of the GNN-based Multihop Question Answering (MHQA) model pipeline. We offer analysis of these results in tandem with presenting them. Finally, after stepping through each feature variation we consider, we summarise our findings and discuss how these findings affect our research goals and questions.

5.1 Main Results

Table 5.1 shows the results for our benchmark models, as well as the HDE model [99] from which our model was primarily inspired. The results show that our HDE-Base model is far from reaching the performance of the HDE model, despite being our most faithful implementation of the HDE model. Our BERT-Base model manages to match the performance of the HDE model, however we would expect the inclusion of BERT [27] to significantly improve model performance over the use of GloVe [74]. Therefore our failure to match the performance of the HDE model without the use of BERT is a notable limitation of our work. From here, we will begin presenting our ablation results, testing the performance effect of various features in our models on the Wikihop dataset. In this chapter, we use the term *Dev* as shorthand for 'Development set'.

Model	Dev Accuracy
HDE [99]	68.1
HDE-Base	60.5
Att-Base	64
BERT-Base	68.2

TABLE 5.1: The development set accuracies of our three benchmark models alongside the HDE [99] model from which are models are inspired.

5.2 GNN Architecture

Here we detail our experiments involving the GNN encoding phase of our model pipeline. We experiment with different message, update and aggregate functions, as well as investigate methods to inject edge information into our GNNs. We also test the usage of weight sharing in our GNN layers, which is common in GNN literature [26, 95, 99].

5.2.1 GNN Cores and In-Out Asymmetries

We begin by providing a set of experiment results which test our GNN-based MHQA models with various combinations of GNN cores (see Sec 3.3.1) and In-Out asymmetries (see Sec 3.3.2). These results are presented in the appendix in tables A.1 and A.2, however they are difficult to interpret due to the large quantity of information. To present our results more legibly here, we step through each feature and highlight the tests involving that feature. Here, each test is simply a comparison between two or more model configurations which differ in only one feature setting.

What follows are our GNN Core and in-out asymmetry test results, organised to highlight one feature at a time. The tables all contain both 'Control Variable' and 'Independent Variable' columns. The tables intend to compare each of the options in the 'Independent Variable' set to one another, under multiple conditions. The 'Control Variables' are thus shown to provide information as to the conditions that the 'Independent Variables' are being compared under. Each row in these tables represents a different experimental setting, using a different set of control variables.

5.2.1.1 GNN Core

Here we present the results relevant to the choice of GNN core, defined in section 3.3.1. The results in table 5.2 show that our Edge-Core GNNs outperform the HDE's Switch-Core when gating is used in the update function, however when no gating is used, the Edge-core performance suffers more than the Switch-Core. We can also see that the Edge-Core outperforms the Sage-Core. These two cores differ in their use of a linear transformation in the message function, thus these results indicate that a learned message transformation is likely useful for our GNNs.

Control Variables			GNN Core		
Gating	MLP Asym	SAGE Asym	Edge	Switch	Sage
Yes	No	No	60.5	58.9	58.9
Yes	Yes	No	60.3	58.0	59.6
No	Yes	No	58.2	58.6	NA

TABLE 5.2: Tests comparing Our Edge-Core to the SAGE-Core and HDE's [99] Switch-Core. All hyperparameters not specified are the same as those used in our HDE-Base model.

Table 5.3 shows the results for various GNN-Cores in our Att-Base models. The results show that the attention-based GNNs (SDP, GAT) outperform the non attention-based GNN Edge-Core in all settings. However, the effectiveness of the two attention cores is strongly influenced by the choice of the gating and TUF in-out asymmetries (see Sec 2.5.2). We will further investigate the effect of these asymmetries in section 5.2.1.2. However, we will note here that the GAT [102] does not include any in-out asymmetries, and performs poorly, achieving only 29%. This is likely since our GNNs are deep, containing 9 layers. For reference, the HGN model [31] uses the GAT without an in-out asymmetry, however it only makes use of a single layer. Between the two attention-based cores, the SDP-Core outperforms the GAT-Core in most settings, and in its maximum performance.

Finally, table 5.4 briefly shows the results of our comparison between the two attention-based cores in the context of our BERT-Base models. The results show that the SDP-Core either matches or outperforms the GAT-Core in both settings. The results again allude to the

Control Variables		GNN Core		
Gating	TUF	GAT	SDP	Edge
Yes	Yes	64.0	64.7	60.2
No	Yes	60.4	66.0	62.5
Yes	No	62.9	61.4	58.5
No	No	29.0	Failed to Train	NA

TABLE 5.3: Tests comparing Our Edge-Core to the GAT-Core and the Transformer's [101] SDP-Core with K-Type edge embeddings. All hyperparameters not specified are the same as those used in our Att-Base model.

effect that the choice of asymmetries have on performance, with gating having a different effect on the two cores.

Control Variables	GNN Core	
Gating	GAT	SDP
Yes	68.2	68.2
No	68.3	71.4

TABLE 5.4: Tests comparing the GAT-Core and the SDP-Core. All hyperparameters not specified are the same as those used in our BERT-Base model.

5.2.1.2 *In-Out Asymmetries*

Here we present the results relevant to the choice of in-out asymmetries defined in section 3.3.2.

Control Variables				Independent Variable	
GNN Core	TUF	MLP Asym	SAGE Asym	Gating	No Gating
Edge	No	No	Yes	61.9	62.1
	Yes	No	No	61.5	64.1
	No	Yes	No	60.3	58.2
	No	No	No	60.5	52.7
Switch	No	Yes	No	58.0	58.6
SAGE	No	No	Yes	58.0	Failed to Train

TABLE 5.5: Tests comparing the use of GNN Gating in our HDE-Base models.

Gating. We begin by examining the role of gating in our models as defined in section 2.5.2.4 by equation 2.37. Table 5.5 shows the results of adding/removing gating in our HDE-Base models, under various conditions. The results are mixed, with gating having a positive or negative effect in different configurations. When considering the Edge-Core, the maximum performance is without gating, while using the TUF. The minimum performance is when no in-out asymmetry is used at all.

Table 5.6 shows the same mixed results shown above, but for our Att-Base models. As in Table 5.5, the highest performing model uses only the TUF, while the worst performing

Control Variables		Independent Variable	
GNN Core	TUF	Gating	No Gating
SDP	Yes	64.7	66.0
GAT	Yes	64.0	60.4
GAT	No	62.9	29.0
SDP	No	61.4	Failed to Train
Edge	Yes	60.2	62.5

TABLE 5.6: Tests comparing the use of GNN Gating in our Att-Base models.

models use neither the TUF nor gating. Overall, these two tables of results demonstrate that there is no consistent advantage to the use of gating in our GNNs. Instead, gating seems to be somewhat interchangeable with other in-out asymmetries. The results also show that gating seems to be more beneficial to our GAT-Core models which use additive attention, and not beneficial to our SDP-Core models.

Control Variables	Independent Variable	
Gating	TUF	No TUF
No	64.1	52.7
Yes	61.5	60.5

TABLE 5.7: Tests comparing the use of the TUF in our HDE-Base models.

Transformer Update Function We provide a set of experiments aimed to test the efficacy of using the Transformer Update Function (TUF) in our GNN models. The results from tables 5.7 and 5.8 clearly show that in all tested cases, the inclusion of the TUF improves model performance. This is most pronounced when the GNN does not include gating. These results strongly support the usage of the TUF in GNNs, which outside of the Transformer literature, is not used in any published GNNs to the best of our knowledge.

Control Variables		Independent Variable	
GNN Core	Gating	TUF	No TUF
SDP	No	66.0	Failed to Train
SDP	Yes	64.7	61.4
GAT	Yes	64.0	62.9
GAT	No	60.4	29.0
Edge	Yes	60.2	58.5

TABLE 5.8: Tests comparing the use of the TUF in our Att-Base models.

MLP and SAGE Asymmetries Table 5.9 shows mixed results for the use of the MLP-Asymmetry in our HDE-Base models. The worst result being the model variant which does not use any in-out asymmetry, further supporting the notion that having at least one in-out asymmetry is useful to our GNNs. Table 5.10 shows that the SAGE-Asymmetry has a positive effect on model performance in all of the few cases tested.

Control Variables		Independent Variable	
GNN Core	Gating	MLP-Asymmetry	No MLP-Asymmetry
Edge	Yes	60.3	60.5
Edge	No	58.2	52.7
SAGE	Yes	59.6	57.4
Switch	Yes	58.0	58.9

TABLE 5.9: Tests comparing the use of the MLP Asymmetry in our HDE-Base models.

Control Variables		Independent Variable	
GNN Core	Gating	SAGE Asymmetry	No SAGE Asymmetry
Edge	Yes	61.9	60.5
Edge	No	62.1	52.7
SAGE	Yes	58.0	57.4

TABLE 5.10: Tests comparing the use of the SAGE Asymmetry in our HDE-Base models.

5.2.2 Edge Information

We consider three distinct ways of incorporating edge information into our GNNs. These include two methods of injecting edge type embeddings, and one method of using distinctly parameterised linear transformations per edge type. These methods are K-Type and V-Type edge embeddings (see Sec 2.4.5), and the R-GCN [86] inspired Switch-Core (see Sec 3.3.1.1).

We found that using a Switch-Core increased training time by just over 30%, while delivering decreased model performance. Table 5.11 shows a 1.6% decrease in model performance when using a Switch GNN when compared to using V-Type edge embeddings on our HDE-Base model. We also compare the performance of K-Type and V-Type embeddings in our Att-Base SDP model. The results show a small improvement in performance when using V-Type embeddings instead of K-Type embeddings, although the difference of 0.5% is too small to be considered significant. Nevertheless this result is notable since the authors of the Relative Positional Transformer [92] paper, who first proposed these two edge type embeddings, found the opposite result. The authors own ablations found that K-Type embeddings were preferable to V-Type embeddings, while our results contradict this. V-Type embeddings are more generally applicable than K-Type embeddings which can only be used in combination with SDP attention. It is thus noteworthy that our model results show that V-Type embeddings may work as well in GNNs as do K-Type embeddings.

Table 5.11 also indicates that removing edge type embeddings entirely from our Att-Base and HDE-Base models has a large negative influence on model performance, indicating that models which do not make use of edge type information perform worse.

5.2.3 Weight Sharing

Finally, with regards to our GNN architecture, we evaluate the usage of weight sharing in our GNNs. Table 5.12 demonstrates a large negative influence from weight sharing on our HDE-Base model, and a small negative influence on our Att-Base model. These results corroborate our preliminary findings which indicate that using distinct weights per GNN layer is preferable to using shared weights, in the context of our MHQA models.

Benchmark Model	Edge Information Type	Dev Accuracy	Δ
HDE-Base	V-Type Embeddings	60.5	.
HDE-Base	Switch-Core	58.9	-1.6
HDE-Base	None	58.1	-2.4
Att-Base SDP	K-Type Embeddings	64.7	.
Att-Base SDP	V-Type Embeddings	65.2	+ 0.5
Att-Base SDP	V-Type + K-Type Embeddings	64.8	+ 0.1
Att-Base SDP	None	60.2	-4.5

TABLE 5.11: The results of different edge information methods in our HDE-Base and Att-Base models. The Att-Base model here is modified to use Scaled Dot-Product (SDP) attention instead of Additive Attention.

Benchmark Model	Layer Weight Sharing	Dev Accuracy	Δ
HDE-Base	No	62.0	.
HDE-Base	Yes	60.5	-1.5
Att-Base	No	64	.
Att-Base	Yes	63.6	-0.4

TABLE 5.12: The results of removing weight sharing from our HDE-Base model, and adding weight sharing to our Att-Base model.

We further investigate whether different components of a GNN respond differently to being distinctly parameterised, as compared to when sharing weights. Table 5.13 shows that in all but one case, including any form of weight sharing serves to degrade model performance. The results do show a small improvement of 0.3% when sharing the GNN-Core and gating weights, however this difference is not large enough to be considered significant.

Benchmark Model	Share GNN-Core	Share Gate	Share TUF	Dev Accuracy	Δ
Att-Base	No	No	No	64	.
Att-Base	Yes	No	No	63.6	-0.4
Att-Base	No	Yes	No	62.6	-1.4
Att-Base	No	No	Yes	63.3	-0.7
Att-Base	Yes	Yes	No	64.3	+ 0.3
Att-Base	No	Yes	Yes	61.1	-2.9
Att-Base	Yes	Yes	Yes	63.6	-0.4

TABLE 5.13: The results of sharing the weights of various individual components in isolation, or in combination.

5.3 Graph Construction

Here we investigate a few key variations in the construction of our context graphs. Namely, we investigate the utility of sparsely connected graphs in contrast to fully connected graphs. We compare the usage of Special and Detected entities (see Sec 2.6.1), as well as evaluate the inclusion of sentence nodes. We also ablate the usage of HDE's [99] two optional edges, the Co-document and Compliment edges. Lastly, we evaluate the usage of bidirectional edges in comparison to unidirectional edges.

5.3.1 Graph Structure

We ablate the usage of sparse graph structure in our Att-Base SDP model, meaning we test the usefulness of respecting HDE's [99] edge connectivity as opposed to making use of fully connected graphs. For this test, we make use of K-Type edge embeddings with sparse graph structure and compare it with fully connected graphs. Making fully connected graphs involves the addition of the *Unconnected Edge* (see Sec 3.1). All previous edge types are still present, and the model is able to differentiate between them, and the Unconnected Edges.

Table 5.14 shows the results of this test, and demonstrate that the highest performing model makes use of both graph structure and edge types, however the influence of sparse graph structure is small (0.3%) and may be insignificant. We see that when edge types are not present, making use of sparse graph structure serves to decrease model performance by 1.3%. The results in table 5.14 seem to indicate that edge type information is more important than sparse graph structure, and that using fully connected graphs, as is common in Transformer literature, may be sufficient for modeling graph data in NLP. These results also indicate that when sparse graph structure is used, edge information is especially important.

Benchmark Model	Sparse Graph Structure	Edge Embeddings	Dev Accuracy	Δ
Att-Base SDP	Yes	Yes	64.7	.
Att-Base SDP	Yes	No	60.2	-4.5
Att-Base SDP	No	Yes	64.4	-0.3
Att-Base SDP	No	No	61.5	-3.2

TABLE 5.14: A table showing the performance of our Att-Base SDP models with and without graph structure. Here the graph structure is imposed via masking, hence removing the masking step amounts to ignoring graph structure.

We perform a second experiment investigating the importance of graph structure and edge information, this time comparing HDE-style [99] hand crafted graph structure with random graph structure, similar to that which is used in Bigbird [114], the Sparse Transformer. Table 5.15 shows that hand crafted structure and edge embeddings significantly outperform both settings without edge information. The results also show that when no edge information is available, there is essentially no difference in performance between random and hand crafted graph structure. It should be noted that we retained the fully connected set of candidate nodes for our random graphs. Finally, we evaluate a variation which attempts to use edge information with random graph structure. The edge information here is essentially only the two-tuple of the node types which the edge connects, such as 'Entity-Candidate'. The results show that edge information without hand crafted graph structure is not useful. Thus the combination of hand crafted graph structure and edge information is especially important to model performance.

5.3.2 Special Entities vs Detected Entities

Further investigating graph composition, we experiment with the usage of Special and Detected entities (see Sec 2.6). Table 5.16 shows that our BERT-Base model with only Special entities performs best, while using only Detected entities performs significantly worse. We also test a model which uses the combination of Special and Detected entities and found that its performance was worse than the model using only Special entities, but only by 0.6%. This seems to indicate that when answer candidate information is available, it is

Benchmark Model	Graph Structure Type	Edge Embeddings	Dev Accuracy	Δ
BERT-Base SDP	Hand Crafted	Yes	68.2	.
BERT-Base SDP	Hand Crafted	No	64.4	-3.8
BERT-Base SDP	Random	No	64.5	-3.7
BERT-Base SDP	Random	Yes	63.8	-4.4

TABLE 5.15: A table showing the performance of our BERT-Base SDP models with hand crafted and random graph structure.

Benchmark Model	Entity Extraction Method	Dev Accuracy	Δ
BERT-Base SDP	Special	68.2	.
BERT-Base SDP	Special + Detected	67.4	-0.8
BERT-Base SDP	Detected	64.3	-3.9

TABLE 5.16: A table showing the performance of the BERT-Base SDP model, which makes use of Special entities only. Also included are the results of a model variation which uses Detected entities only, and a variation which uses Detected and Special entities.

preferable over Detected entities. It is also possible that making use of Detected entities opens the model up to issues of error propagation [57], whereby any errors made by the entity detection algorithm necessarily hurt the performance of our model. Making use of a more modern RoBERTa-based entity recognition [62, 105] algorithm[14] may improve the performance of the Detected entities in a MHQA model.

5.3.3 Sentence Nodes

Inspired by the graph construction process which is used in Microsoft's HGN model [31], we also evaluated the usage of sentence nodes in our HDE-inspired system. Functionally, sentences can be processed exactly as entity nodes are. This is because both entities and sentences are defined by token spans in a specific document's token sequence. We investigate two variations of sentence nodes. In one variation we include sentence nodes for only those Detected sentences which contain an entity (Restricted). In the second variation, we include all Detected sentences in our graphs. Table 5.17 shows that the inclusion of sentence nodes has a positive influence on model performance. Restricted sentence nodes provide the greatest performance improvement (1.2%), while the improvement caused by the inclusion of all sentence nodes may not be significant (0.4%). These results support the inclusion of HGN style sentence nodes into the HDE model's graph construction.

Benchmark Model	Sentence Nodes	Dev Accuracy	Δ
BERT-Base SDP	None	68.2	.
BERT-Base SDP	Restricted Sentence Nodes	69.4	+ 1.2
BERT-Base SDP	All Sentence Nodes	68.6	+ 0.4

TABLE 5.17

[14]Spacy NER: `https://spacy.io/usage/facts-figures`

5.3.4 Bidirectional Edges

We decided to test if edge direction information is useful to our models, allowing information to flow differently in the forwards and backwards direction of every edge. Representing edge direction information requires learning an edge embedding for the forward and backwards direction of every edge type, thereby doubling the number of edge embedding parameters.

Benchmark Model	Bidirectional Edge Embeddings	Dev Accuracy	Δ
BERT-Base SDP	No	68.2	.
BERT-Base SDP	Yes	68.8	+0.6

TABLE 5.18

Table 5.18 shows the effect of adding bidirectional edge embeddings into our BERT-Base model. The results show a small positive improvement from bidirectional edge information.

5.3.5 Edges Type Inclusion

Benchmark Model	Ablations	Dev Accuracy	Δ
BERT-Base SDP	.	68.2	.
BERT-Base SDP	+ Compliment Edge	67.3	-0.9
BERT-Base SDP	+ Co-document Edge	69.4	+ 1.2
BERT-Base SDP	- Co-mention Edge	70.0	+ 1.8
BERT-Base SDP	- Co-mention Edge + Co-document Edge	67.9	-0.3

TABLE 5.19

As a final test of graph structuring, we investigate the inclusion of our two optional edge types defined in section 3.1. The results show that the compliment edge has a negative effect on model performance, while the Co-document edge has a positive effect on performance. We further evaluated our BERT-Base model with the Co-mention edge removed. The results show that model performance improves when the Co-mention edge is removed, which is highly surprising. The Co-mention edge connects mentions of the same entity across text and different documents. This function seems intuitively important to multihop reasoning. Next, to see if these graph structuring improvements are synergistic, we evaluate a model variant which adds the Co-document edge and removes the Co-mention edge. The results show that this model scores lower than our BERT-Base model by 0.3%. It is difficult to interpret these results, however some of the difference may be down to random chance. It is also possible that there is a highly nonlinear and unintuitive relationship between graph structuring methods and model performance.

5.4 Graph Embedding

Here we present the results of a small investigation into graph embedding techniques, namely the use of different token embedders, and the use of a Switch Transformer as a summariser.

5.4.1 Token Embedder

We evaluate two different token embedders in our models, namely GloVe, BERT. The dimensionality of these embedders are 300, 512 respectively. Due to our coupling of embedder dimension and model dimension, it is impossible to perform a fair test between these token embedders. Nevertheless, Table 5.20 demonstrates that our BERT-based model performs better than our GloVe-based model. We also tested the utility of fine-tuning our BERT-based embedder during our model training. Fine-tuning yielded a 0.2% improvement over keeping BERT's weights frozen. This difference in performance is likely insignificant, and given that fine-tuning BERT increased the training time by a factor of 1.5, it seems preferable to leave BERT's weights frozen in our models.

Benchmark Model	Token Embedder	Modification	Dev Accuracy	Δ
BERT-Base	BERT	.	68.8	.
BERT-Base	BERT	Fine-tuning	69	+ 0.2
BERT-Base	GloVe	.	65.0	-3.8

TABLE 5.20: Our BERT-Base model with various token embedder configurations.

5.4.2 Switch Summariser

The Switch Summariser (see Sec 3.2.4) was a feature developed early in the project which we erroneously identified as having a positive effect on model performance. Table 5.21 shows that the Switch Summariser has essentially no effect on performance, despite increasing code complexity and execution times.

Benchmark Model	Ablations	Dev Accuracy	Δ
Att-Base	.	64	.
Att-Base	- Switch Transformer-based Summariser	64.1	+0.1

TABLE 5.21: Att-Base with and without the Switch-Summariser

5.4.3 GRU encoders

Benchmark Model	Ablations	Dev Accuracy	Δ
HDE-Base	.	60.5	.
HDE-Base	+ HDE-system's GRUs	57.6	-2.9

TABLE 5.22: HDE-Base with and without GRUs

HDE [99] includes the usage of GRU [21] encoders in its graph embedding step [99]. We removed this feature after our preliminary experiments showed us that the GRUs did not improve model performance, and significantly increased training time. Table 5.22 shows that our formal tests confirmed the negative affect of these GRUs on model performance. Furthermore, the usage of these GRUs more than doubled our models training time.

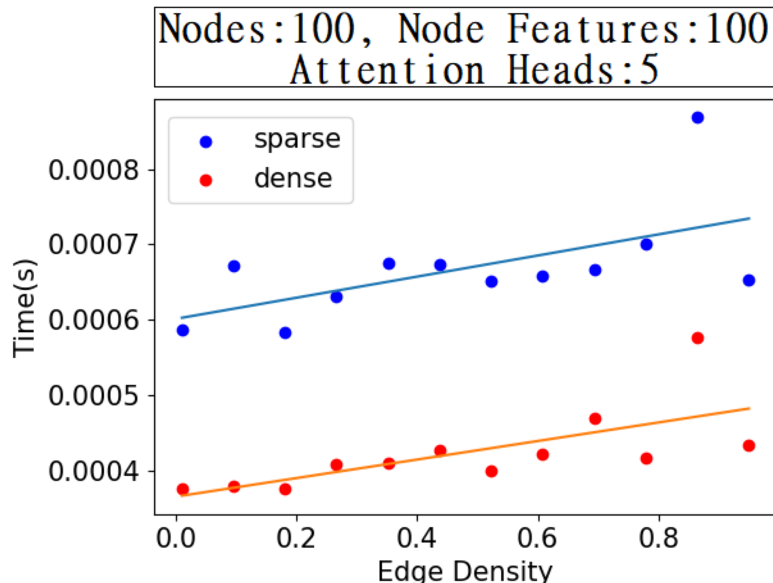

FIGURE 5.1: Average execution times of the Sparse and Dense implementation approaches to a single layer of SDP attention.

5.5 Sparse and Dense GNN Implementations Results

We briefly investigate the role of graph sparsity on the execution times of the two competing implementation approaches. To test this and memory requirements, we implemented our own SDP attention mechanism in Pytorch Geometric, and generated random graphs at different sparsity levels. The measure *Edge Density* is a common metric used to evaluate graph topology and is simply the number of edges in a graph divided by the maximum possible number of edges in that graph, or rather $\frac{|E|}{n^2}$. Thus, edge density is inversely proportional to graph sparsity. Figure 5.1 shows a small positive correlation between edge density and execution times for both implementation approaches. More notably, figure 5.1 shows that the execution times for the Transformer-style Dense approach are consistently faster than those of the GNN-style Sparse approach.

We also compared the memory usages of the Dense and Sparse approach, and confirmed the observation that the Sparse approach should scale better with sparse graphs. Figure 5.2 shows the relationship between graph sparsity and the memory required to perform SDP attention in both the Dense and Sparse approach. From figure 5.2, we can see that the memory requirements of the Dense approach do not change as the sparsity of the graph changes, while the memory requirements of the Sparse approach decrease as sparsity increases. Figure 5.2 illustrates that for graphs with very low edge density, ie very sparse graphs, the Sparse approach taken by Pytorch Geometric may be preferable to the Dense approach taken by the Transformer in terms of memory requirements.

In Figure 5.2, we see that the two approaches intersect at an edge density of 0.033, meaning for graphs where more than 3.3% of node pairs are connected, in this particular case, it is preferable to use the Dense approach in terms of memory requirements. We refer to this critical edge density as the *Critical Density*, the level of sparsity at which the optimal implementation approach changes. A higher Critical Density means the Sparse approach uses less memory than the Dense approach for more dense graphs. Conversely, a low Critical Density means the cost of implementing the sparse approach outweighs the scaling benefits, for more sparse graphs.

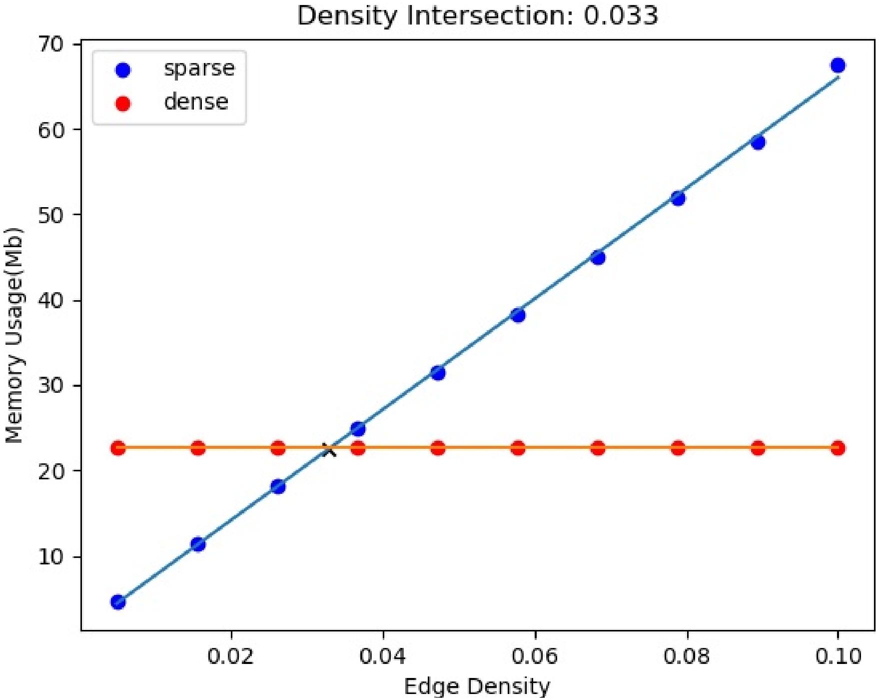

FIGURE 5.2: A figure showing the relationship between Edge density, and the memory requirements for both the Dense and Sparse attention implementations. Edge density is a metric which is inverse to graph sparsity. This graph shows the trade-off under one specific condition, where the number of graph nodes is 500, the number of features per node is 100, and the number of attention heads is 10

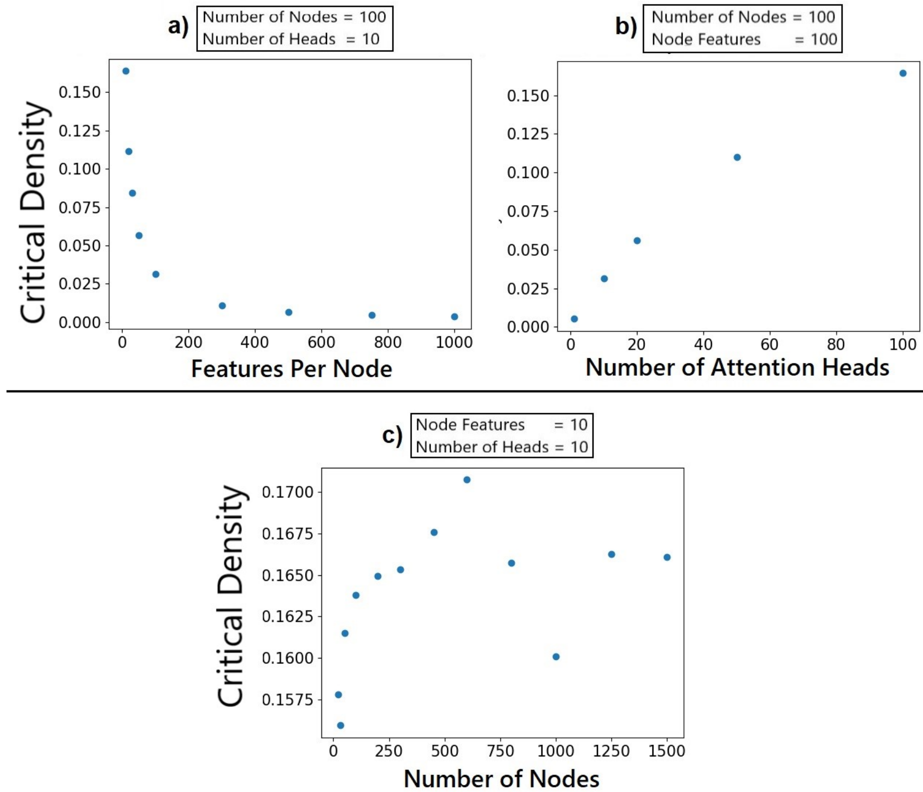

FIGURE 5.3: A figure showing how the metric Critical Density is influenced by the number of attention heads, number of node features, and the number of nodes in a graph. Critical Density informs the decision between the Dense and Sparse implementations, and represents the maximum edge density for which the Sparse approach requires less memory.

We further investigate the relationship between the Critical Density and properties of both the attention mechanism, and the inputted graph. Namely we evaluate the effect of the number of attention heads, the number of nodes n, and the hidden dimension of the attention mechanism f. Figure 5.3(c) shows that the number of nodes in a graph has only a small affect the choice between a Dense and Sparse implementation. Figure 5.3(a and b) shows that when f is small, or the number of heads is large, the Sparse approach may be preferable, but only for very sparse graphs. Overall, these results show that in all but the most extreme cases, the memory requirements of the Dense approach will be lower than that of the Sparse approach. Specifically, one may consider the Sparse approach if one expects very sparse graphs made of nodes with few features.

Sparse Attention models such as the Longformer [8] and Bigbird [114] make use of what we are referring to here as the Dense approach, however they make use of a specific set of optimisations to exploit GPU hardware and improve how memory requirements scale with sparsity. As an example, the Bigbird model found it useful to *blockify* the attention matrix, whereby square blocks are connected instead of individual tokens [114]. This blockification is shown in figure 5.4.

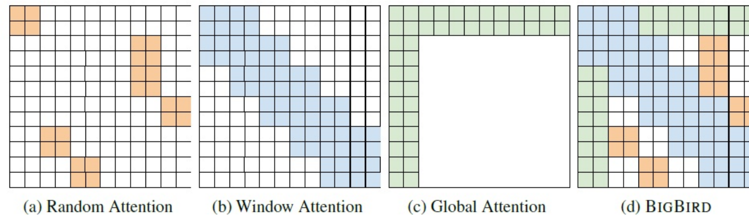

FIGURE 5.4: A figure from [114] showing the blockification used by the BigBird Sparse Transformer to improve how memory requirements scale with graph sparsity.

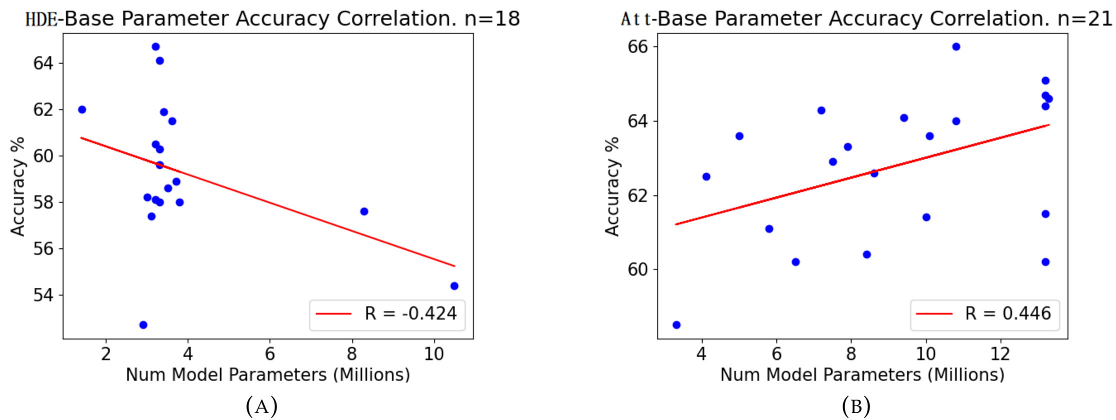

FIGURE 5.5: The correlation between the development set accuracy after training, and the number of parameters for out HDE-Base (A) and Att-Base (B) models.

5.6 Model Parameters

Here we demonstrate the relationship between the number of parameters in our models, and the final development set accuracy of these models. Figure 5.5 shows a negative correlation for our HDE-Base models, and a positive correlation for our Att-Base models. This disagreement may be caused by the weak natural relationship between the number of parameters and the model performance when considering different architecture choices. For example, in HDE-Base the model variation which includes GRU encoding (Rightmost point in Fig 5.5 (a)) has significantly more parameters, despite being notably worse in development set accuracy. In Att-Base, the many shared parameters ablation tests could be partly responsible for the positive relationship, since shared parameters reduces total parameters, and tends to decrease performance. It is worth noting however that SDP-based GNNs have significantly more parameters than do GAT-based GNNs. Thus, the higher task performance of the SDP GNNs is likely contributing to this trend. Despite the increase of parameters in SDP attention, the mechanism is in fact more memory efficient than additive attention [101].

5.7 Results Summary

Here we provide a high level summary of the many ablation tests presented above.

5.7.1 GNN architecture

Our results clearly indicate the benefits of using attention-based GNNs over non attention-based GNNs. Our results also show that the inclusion of the TUF is beneficial in all tested cases, while the use of gating has a mixed effect on performance. More specifically, we found that the GAT-core has a high affinity to gating. Our GAT-Core models perform best when using both gating and the TUF. Our SDP-Core models, however, have a negative affinity to gating, and instead perform best when using the TUF only. The only difference between the GAT-Core and SDP-Core is the method of calculating attention scores - via Additive Attention, or via Scaled Dot-Products. Our highest performing models make use of the SDP-Core, with the TUF as the only in-out asymmetry. This is the configuration used by the canonical Transformer model [101], with the exception of our use of edge type embeddings, which are used by the Relative Positional Transformer [92]. Beyond this, our best performing models make use of distinct weights per GNN layer, as does the Transformer. We also see that models which do not include any in-out asymmetries suffer greatly in performance. This seems to indicate that while some in-out asymmetries like the TUF perform better than others, having at least one asymmetry is necessary for deep GNNs.

5.7.2 Edge Information

We evaluated the use of various methods of including edge information in our GNN models. Our results unambiguously show that all tested methods of including HDE's [99] edge information are beneficial to our models (see Tab 5.11). Beyond this, our results show that using edge type embeddings yields higher performance than using switch-GNNs (see Sec 2.5.2.5). Combined with the conceptual benefits of edge embeddings over switch-GNNs, and the computational speedup, these performance benefits create a strong case for using edge embeddings in GNNs. Finally, we provide a comparison of two types of edge type embedding approaches, namely K-Type and V-Type (see Sec 2.4.5). Our results serve as weak evidence in favour of the usage of V-Type edge embeddings alone. This finding directly contradicts the findings of the original paper proposing these edge embedding methods in the Transformer literature [92], which found that using K-Type embeddings alone performed best. This discrepancy may be due to the different contexts of evaluation between our work and theirs [92]. The original paper's work [92] focused on machine translation [24], while ours focused on Question Answering. It may also be a statistical fluke on our side. In any event, V-Type edge embeddings are appealing since they are capable of being paired with any type of GNN, whereas K-Type embeddings can only be used in combination with SDP attention (see Sec 2.4.1) GNNs.

5.7.3 Graph Structure

We find that graph structure-based sparse attention has little effect on model performance when edge information is used. However, when no edge information is used, we find that this sparsity drastically reduces model performance. This is to say that fully connected graphs/ full self attention is superior to sparse attention when no edge information is available. Edge information can recover this performance which is lost due to sparsity. We also found that when no edge information is available, there is no difference between task-specific hand crafted structure and randomised structure.

We found that the inclusion of HGN style sentence nodes increases model performance, while the addition of detected entity nodes serves to decrease model performance. Finally, our analysis into the effect of different edge types in our models was inconclusive.

5.8 Research Questions

Here we revisit and answer our research questions using the data we have presented.

1. Which features in the HDE MHQA model might improve Transformer-based MHQA models?

    - While the GNN architecture of the Transformer is the best performing GNN that we evaluated, we did find the use of HDE style edge information to be additive to our Transformer GNN-based MHQA models.

2. Do attention-based GNNs outperform non attention-based GNNs in the GNN-based MHQA setting?

    - Yes, our results show that our attention-based GNNs outperform the non attention-based GNNs in almost all tested settings. Furthermore the attention-based GNNs yield our highest model performances.

3. Does edge type information improve GNN-based MHQA model performance, and if so does the Transformer-based approach to edge information improve over the GNN-based approach?

    - Yes, edge type information is highly additive to model performance. We also found that the Relative Positional Transformer [92] style of including edge embeddings to be superior to the typical GNN approach of using distinctly parameterised transformations per edge type.

4. Out of the many existing examples of graph structuring rules in GNN literature, which common methods are important for improving model performance?

    - The results of our brief study into graph structure were inconclusive.

5. Can hand-crafted graph sparsity increase model performance when compared to fully connected graphs?

    - We found that our hand-crafted sparsity did not significantly improve model performance in any tested setting. However, we did find that including edge information can alleviate the performance loss induced by said sparsity. We also found that hand crafted structure without edge information has no advantage over random graph structure.

Chapter 6

Discussion

Attention in Neural Networks has been increasingly relevant to Deep Learning research, largely driven by the success of the Transformer model [101]. While we are not the first to acknowledge that the Transformer is in fact a message passing Graph Neural Network (GNN) (see Sec 2.5.2.3), it is not commonly considered as such. Instead, the GNN and Transformer literatures are separate, and contain divergent methodologies and best practices. In this work, we began by defining a generalised GNN-based Multihop Question Answering (MHQA) system which we evaluate on the Wikihop dataset [106]. We evaluated various architectural choices for each of the three components of a Message Passing GNN (MP-GNN), namely the message, aggregate and update functions. Our empirical results systematically lead us to the conclusion that the best practices in the Transformer literature are superior to those that we tested from the GNN literature, within the context of our evaluations. Specifically, we found that Scaled Dot Product (SDP) attention in combination with the Transformer Update Function (TUF) (see Sec 2.4.3) outperforms Additive Attention used by the Graph Attention Network (GAT), a prominent attention-based GNN. We found that while GNN-style gating (see Sec 2.5.2.4) is useful in combination with the GAT, it reduces the performance of SDP attention, and thus would likely not benefit the Transformer literature. Conversely, the TUF does improve the performance of the Additive Attention mechanism, as well as other tested GNNs. We introduced the concept of an in-out asymmetry in MP-GNNs (see Sec 2.5.2), which is essentially any function used in a GNN's update function that is able to differentiate between a nodes current state, and the messages it receives from its neighbours. The popular GAT [102] GNN does not include any in-out asymmetry, and as such seems to suffer in our deep GNN use case. In-Out asymmetries likely help to reduce the over-smoothing problem whereby node states can become more homogeneous after each successive GNN layer. Making use of at least one in-out asymmetry may help the GNN to preserve node identities by biasing the GNN to treat current states differently to neighbour node states.

The approach to including edge information is distinct in the GNN and Transformer literatures. To the best of our knowledge, there are no published Relational GNNs (see Sec 2.5.2.5) which make use of Transformer-style edge embeddings. Despite this, we found that edge embeddings outperformed Switch-GNNs (see Sec 2.5.2.5) in all tested settings. This may be due to the nature of hard and soft inductive model biases [23]. This is to say that since edge type embeddings add few learnable parameters to a model, it may be easy for the model to learn to 'ignore' this edge type information. This could be trivially achieved by learning to zero out the edge type embeddings, removing their influence entirely. Thus, using edge type embeddings allows a model to utilise edge information only if and when it is necessary, while not serving as an obstruction to the model when unnecessary. By contrast, using distinct linear transformations per edge type (Switch-GNN), as is common in GNN literature [31, 86, 45, 63, 99], may prove more difficult for a model to learn to ignore.

Furthermore, edge type embeddings reduce problems related to uneven edge type distributions. Overall, our work serves as a motivation to use more Transformer-like GNNs, while we were not able to find any GNN techniques which may aid in the Transformer literature.

Sparse Transformers are currently a prominent method of increasing the maximum sequence length of Transformer models [114, 8]. The BigBird paper offered a theoretical proof that sparsity decreases Transformer model expressiveness [114]. The bigbird model does not natively include edge type information, despite its use of hand crafted graph connectivity rules. Our work suggests that including the natural edge type information available to BigBird into its attention mechanism may help to alleviate some of the performance which is lost due to its use of sparsity. This insight may have implications for the current state-of-the-art (SOTA) Sparse Transformers in certain domains such as question answering (QA).

We investigated the memory usage scaling of two prominent implementation approaches for MP-GNNs. Namely the Transformer-style Dense approach and the GNN-style Sparse Approach (see Sec 4.1). We found that unless a very high degree of sparsity is required, it is preferable to use the Transformer-style implementation. This result, together with the Transformers model architectural superiority, indicate that the Transformer may be the best option for modeling graph data in terms of both computational requirements and model performance, in all but the most extreme cases.

6.1 Limitations

Our work involves evaluating GNN-based MHQA models on the Wikihop dataset. Due to this highly constrained context, it is unclear how well our findings would generalise to other MHQA tasks or QA tasks in languages other than English. We also attempt to offer insights into GNN architectures in a more general sense than QA or even Natural Language Processing (NLP), however since we did not evaluate our GNN variations outside of NLP, we cannot verify our findings. Due to the general success of the Transformer model, our main finding that the Transformer is the best GNN of the GNNs we evaluated seems particularly plausible. Beyond this, our model evaluations contain a degree of stochasticity, and thus some of our results may be skewed by randomness. Given more time and compute, we would repeat our experiments multiple times to control for this noise. Another limitation of our evaluations is the lack of dedicated hyperparameter tuning for certain features. It is possible that certain model variations we consider would benefit from vastly different hyperparameters than our benchmark models use. Thus we may not be comparing features at their optimal performance, which would introduce another source of noise into our results.

6.2 Generalisability

Despite our use of what is essentially the Transformer in some of our GNN-based MHQA models, we were unable to achieve performance anywhere near that of the Sparse Transformer approach. One likely explanation is the lack of end-to-end pretraining in GNN-based QA models. This is due to the lack of a unified method to pretrain GNN-based NLP models. Another striking difference between these two model approaches is the GNN-based models' use of node level elements, compared to the Transformer models' use of token level elements. Tokens represent information at a much finer granularity, and thus at a much higher dimensionality. Using node representations thus serves as a form of dimensionality reduction, massively reducing the compute requirements of an attention-based QA model. In problems where the maximum sequence length is within the limits of a Sparse Transformer (around 4000), token level representations are both feasible and likely

preferable to node level representations. This includes the Wikihop dataset on which we evaluated. QA problems which involve yet longer context sequences, such as the Question Answering with Long Input Texts (QuALITY) dataset [10], may still require and benefit from node level representations.

Another point to consider here is the distribution of important information in the source context sequences. The Wikihop and HotpotQA [112] datasets both include many text documents which serve only as distractors. This is to say that many given documents are irrelevant to the QA process. As such, models like the Hierarchical Graph Network (HGN) [31] employ a document selection step, where documents are classified as either distractors or relevant. They then exclude all distractor documents, vastly decreasing the total amount of context information. This is a promising approach to long context QA problems, however it may not always be possible. In real world scenarios where relevant information may be scattered across multiple documents, models which process all given information may still be required. Furthermore, in scenarios where there are many relevant documents, token level models may still be infeasible even after document selection.

Another similar approach involves extracting reasoning chains (see Sec 2.2.3) from given context [15]. This amounts to sentence selection, a finer grained version of document selection. This is an appealing solution to the long context problem in QA, however the process of selecting valid reasoning chains may be difficult and brittle. Consider a case where the full context information is required in order to recognise the importance of a sentence. Both document and sentence selection methods attempt to avoid the long context problem without truly modeling global context information. In contrast both Sparse Transformers and GNN-based approaches are able to model global context information.

GNN-based QA models can implicitly act as sentence selection models, since there is no guarantee that every sentence in the source context will be represented by a node. Document level nodes are technically capable of representing all contained sentences, however this information is very coarse grained, possibly missing important information. Our best performing models made use of special entity nodes (see Sec 2.6.1), which directly exploit answer candidate information. Not only is this not applicable to non multiple choice QA, this also means that sentences which do not contain any candidate entities will not be directly modeled by the GNN. We also evaluated variations which included Detected Entities as extracted by a Named Entity Recognition (NER) module. Detected Entities would allow our models to be used in any QA task beyond multiple choice. Our results showed that Detected Entities underperformed when compared to Special Entities. This is expected, since exploiting answer candidate information when it is available should intuitively aid in QA.

As a final thought on model generalisability, consider that the two prominent MHQA datasets that we consider, Wikihop [106] and HotpotQA [112], both involve highly contrived questions and context. Due to the nature of their dataset construction processes (see Sec 2.2), the distribution of relevant facts across documents contains implicit structure. Any model which directly exploits this structure, such as the Heterogeneous Document Entity (HDE) model, would likely achieve higher task performance. However, in a real world QA setting, the distribution of relevant facts across documents may be unknown and unstructured. Thus, models which represent information in an unbiased manner would likely perform better in more generalised real world scenarios. Sparse Transformers include all context tokens, and thus do not exploit any such structure. GNN-based models which include nodes for Detected Entities as well as generic nouns would be significantly less biased than those with Special Entities. This is since many sentences contain Named Entities, and every sentence contains a noun. Thus such a GNN-based QA model would represent information from all around the given context in an unbiased manner.

6.3 Future work

1. Sparse Transformers and Document selection models are currently SOTA for MHQA tasks [64, 31, 114], however GNN-based QA may excel in scenarios where Sparse Transformers are infeasible, such as very long context QA. To validate the continued study of these GNN-based QA models, it may be necessary to benchmark models with a challenging new MHQA dataset. Such a dataset should contain very long context sequences (>4000 tokens) made up of many documents. The important information contained in these documents should be distributed in a dispersed and unstructured manner, in order to make document selection infeasible. Lastly, the question answering should require a high degree of global context information modeling. As an example consider a long form QA task which may be given to a university level history student. Such a task requires reading and integrating many interrelated facts and opinions from many distinct documents with a completely unpredictable distribution of important information. To this end, the QuALITY dataset [10] is promising, with an average token length of 5000 tokens per context example.

2. Our results motivate the addition of edge type embeddings to Sparse Transformers. To validate this, a BigBird-style Sparse Transformer could be trained to perform MHQA while using edge type embeddings. This would reveal whether edge type information is useful for token level models, or just node level models.

3. Token level Transformer models typically benefit from end-to-end pretraining, however GNN-based MHQA currently only makes use of pretrained token embedders. Thus, a unified method of pretraining GNN-based MHQA models in an end-to-end fashion may help to bridge the performance gap between token level and node level QA models.

6.4 Final Thoughts on the Long Context Problem

In this work we have primarily explored the use of coarse grained node representations as a method of reducing compute requirements in QA. However, grouping tokens into nodes can only reduce compute requirements so much. Using yet coarser nodes would result in more dimensionality reduction, which may reduce modeling capacity even further. GNN-based MHQA models are naturally sparse, as not all nodes are connected. When combined with implementation insights from BigBird [114], this should allow GNN-based MHQA models to scale to even longer context sequences. However, this too has its limits, with extra graph sparsity reducing the expressiveness of the attention mechanism [114]. Finally, document and sentence selection mechanisms [31, 64, 15] are able to iteratively select relevant context without ever modeling global context information, however some information may not be recognisable as important without global context. It is worth noting that all of these strategies are independent, and thus could be combined into a single system. HGN for example combines sparsely connected node level processing with document selection. However, fundamental problems remain. Thus we believe that a different approach entirely may be required for extremely long context problems. Fusion-in-Decoder models [48] are able to use self-attention to process input documents in a way which scales linearly with the number of input documents. These models only allow for cross-document communication in the decoder stack of a Transformer, which may limit the expressiveness of cross-document reasoning. The Perceiver model [49] is a very appealing candidate. It is able to query fine-grained and global information without relying on sparsity or independent document encoding. To our knowledge, the Perceiver model has not been thoroughly evaluated on MHQA tasks.

Chapter 7

Conclusion

In this work, we created a configurable Graph Neural Network (GNN)-based Multihop Question Answering (MHQA) system and used it to compare GNN architectural choices. We evaluated our models using the challenging Wikihop [106] MHQA dataset. We compared two prominent attention mechanisms, and a non attention-based alternative. We found that the Scaled Dot Product (SDP) attention found in the Transformer [101] delivered our highest task performance. We also compared GNN-style update functions (see Sec 2.5.1) to the Transformers update function which we have named the Transformer Update Function (TUF). Our results demonstrated that the TUF works especially well with SDP attention, which together define the Transformer. Our results also showed that the TUF is beneficial to all other tested GNNs. We also compared GNN-style edge information against Transformer-style edge information, and again found that the Transformer style edge type embeddings (see Sec 2.4.5) produced the highest task performance. Overall, the Transformer is the best GNN that we tested. We further performed an analysis into the memory requirements of the Transformer-style implementation of the attention mechanism to the GNN-style implementation and found that the Transformer style implementation is more efficient for most graphs excepting very sparse graphs. From this we can recommend that GNN-based MHQA systems should consider using the Transformer model in both architecture and implementation.

Our results support the notion that sparsity decreases attention modeling capacity, however they also show that including edge type information can help to recover this lost performance. Our results also show that without edge information, there may be no value in utilising hand crafted graph construction rules, and that random graph connections may be as effective.

Appendix A

Appendix

Shown here in tables A.1 and A.2 are the full set of test results collected to compare different GNN architectures. GNNs are compared in terms of their GNN Core (see Sec 3.3.1) and their in-out asymmetries (see Sec 2.5.2).

GNN Core	Model No.	Gating	TUF	MLP Asym	SAGE Asym	Dev Accuracy	Name
Edge	1.	Yes	No	No	No	60.5	HDE-Base (ours)
	2.	No	No	No	No	52.7	.
	3.	Yes	Yes	No	No	61.5	.
	4.	No	Yes	No	No	64.1	.
	5.	No	No	Yes	No	58.2	.
	6.	Yes	No	Yes	No	60.3	.
	7.	Yes	No	No	Yes	61.9	.
	8.	No	No	No	Yes	62.1	.
Switch	9.	Yes	No	No	No	58.9	.
	10.	Yes	No	Yes	No	58.0	HDE's GNN [99]
	11.	No	No	Yes	No	58.6	R-GCN [86]
SAGE	12.	Yes	No	No	Yes	58.0	.
	13.	Yes	No	No	No	57.4	.
	14.	Yes	No	Yes	No	59.6	.
	15.	No	No	No	Yes	Failed to Train	GraphSAGE [38]

TABLE A.1: The results of our GNN Core and In-Out Asymmetry experiments for our HDE-Base model.

GNN Core	Model No.	Gating	TUF	MLP Asym	SAGE Asym	Dev Accuracy	Name
GAT	16.	Yes	Yes	No	No	64.0	Att-Base (ours)
	17.	No	Yes	No	No	60.4	.
	18.	Yes	No	No	No	62.9	.
	19.	No	No	No	No	29.0	GAT [102]
SDP	20.	Yes	Yes	No	No	64.7	.
	21.	Yes	No	No	No	61.4	.
	22.	No	Yes	No	No	66.0	Transformer [101]
	23.	No	No	No	No	Failed to Train	.
Edge	24.	Yes	Yes	No	No	60.2	.
	25.	No	Yes	No	No	62.5	.
	26.	Yes	No	No	No	58.5	.

TABLE A.2: The results of our GNN Core and In-Out Asymmetry experiments for our Att-Base model.

Bibliography

[1] M. D. Aaditya et al. "Layer Freezing for Regulating Fine-tuning in BERT for Extractive Text Summarization". In: *25th Pacific Asia Conference on Information Systems, PACIS 2021, Virtual Event / Dubai, UAE, July 12-14, 2021*. Ed. by Doug Vogel et al. 2021, p. 182. URL: https://aisel.aisnet.org/pacis2021/182.

[2] Abien Fred Agarap. "Deep Learning using Rectified Linear Units (ReLU)". In: *CoRR* abs/1803.08375 (2018). arXiv: 1803.08375. URL: http://arxiv.org/abs/1803.08375.

[3] Ekin Akyürek and Jacob Andreas. "Lexicon Learning for Few-Shot Neural Sequence Modeling". In: *CoRR* abs/2106.03993 (2021). arXiv: 2106.03993. URL: https://arxiv.org/abs/2106.03993.

[4] Simna Ashraf. "SOCIAL MEDIA NEWS GENERATION". In: *Medium* (2020). URL: https://medium.com/@simnaashraf7/social-media-news-generation-2d7db2d5f18a.

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer Normalization". In: *stat* 1050 (2016), p. 21.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1409.0473.

[7] Peter W Battaglia et al. "Relational inductive biases, deep learning, and graph networks". In: *CoRR* abs/1806.01261 (2018). arXiv: 1806.01261. URL: http://arxiv.org/abs/1806.01261.

[8] Iz Beltagy, Matthew E. Peters, and Arman Cohan. "Longformer: The Long-Document Transformer". In: *CoRR* abs/2004.05150 (2020). arXiv: 2004.05150. URL: https://arxiv.org/abs/2004.05150.

[9] Parul Kalra Bhatia, Tanya Mathur, and Tanaya Gupta. "Survey paper on information retrieval algorithms and personalized information retrieval concept". In: *International Journal of Computer Applications* 66.6 (2013).

[10] Samuel R Bowman et al. "QuALITY: Question Answering with Long Input Texts, Yes!" In: *NAACL 2022* (2022).

[11] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[12] Joan Bruna et al. "Spectral Networks and Locally Connected Networks on Graphs". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2014. URL: http://arxiv.org/abs/1312.6203.

[13] Yu Cao, Meng Fang, and Dacheng Tao. "BAG: Bi-directional Attention Entity Graph Convolutional Network for Multi-hop Reasoning Question Answering". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 2019, pp. 357–362.

[14] Deli Chen et al. "Measuring and relieving the over-smoothing problem for graph neural networks from the topological view". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 3438–3445.

[15] Jifan Chen, Shih-Ting Lin, and Greg Durrett. "Multi-hop Question Answering via Reasoning Chains". In: *CoRR* abs/1910.02610 (2019). arXiv: 1910.02610. URL: http://arxiv.org/abs/1910.02610.

[16] Lili Chen et al. "Decision transformer: Reinforcement learning via sequence modeling". In: *Advances in neural information processing systems* 34 (2021).

[17] Ting Chen et al. "A Simple Framework for Contrastive Learning of Visual Representations". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 1597–1607. URL: http://proceedings.mlr.press/v119/chen20j.html.

[18] Rewon Child et al. "Generating Long Sequences with Sparse Transformers". In: *CoRR* abs/1904.10509 (2019). arXiv: 1904.10509. URL: http://arxiv.org/abs/1904.10509.

[19] Krzysztof Marcin Choromanski et al. "Rethinking Attention with Performers". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: https://openreview.net/forum?id=Ua6zuk0WRH.

[20] Fan RK Chung and Fan Chung Graham. *Spectral graph theory*. 92. American Mathematical Soc., 1997.

[21] Junyoung Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *CoRR* abs/1412.3555 (2014). arXiv: 1412.3555. URL: http://arxiv.org/abs/1412.3555.

[22] Ronan Collobert et al. "Natural language processing (almost) from scratch". In: *Journal of machine learning research* 12.ARTICLE (2011), pp. 2493–2537.

[23] Stéphane d'Ascoli et al. "ConViT: Improving Vision Transformers with Soft Convolutional Inductive Biases". In: *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 2286–2296. URL: http://proceedings.mlr.press/v139/d-ascoli21a.html.

[24] Raj Dabre, Chenhui Chu, and Anoop Kunchukuttan. "A survey of multilingual neural machine translation". In: *ACM Computing Surveys (CSUR)* 53.5 (2020), pp. 1–38.

[25] Zihang Dai et al. "Transformer-XL: Attentive Language Models beyond a Fixed-Length Context". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019, pp. 2978–2988.

[26] Nicola De Cao, Wilker Aziz, and Ivan Titov. "Question Answering by Reasoning Across Documents with Graph Convolutional Networks". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 2019, pp. 2306–2317.

[27] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *NAACL-HLT (1)*. 2019.

[28] Prafulla Dhariwal et al. "Jukebox: A Generative Model for Music". In: *CoRR* abs/2005.00341 (2020). arXiv: 2005.00341. URL: https://arxiv.org/abs/2005.00341.

[29] Bhuwan Dhingra et al. "Neural Models for Reasoning over Multiple Mentions Using Coreference". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. 2018, pp. 42–48.

[30] Chris Dyer. "Should Neural Network Architecture Reflect Linguistic Structure?" In: *Proceedings of the 21st Conference on Computational Natural Language Learning (CoNLL 2017), Vancouver, Canada, August 3-4, 2017*. Ed. by Roger Levy and Lucia Specia. Association for Computational Linguistics, 2017, p. 1. DOI: 10.18653/v1/K17-1001. URL: https://doi.org/10.18653/v1/K17-1001.

[31] Yuwei Fang et al. "Hierarchical Graph Network for Multi-hop Question Answering". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2020, pp. 8823–8838.

[32] William Fedus, Barret Zoph, and Noam Shazeer. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity". In: *CoRR* abs/2101.03961 (2021). arXiv: 2101.03961. URL: https://arxiv.org/abs/2101.03961.

[33] Weijiang Feng et al. "Audio visual speech recognition with multimodal recurrent neural networks". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, pp. 681–688.

[34] Matthias Fey and Jan Eric Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *CoRR* abs/1903.02428 (2019). arXiv: 1903.02428. URL: http://arxiv.org/abs/1903.02428.

[35] Justin Gilmer et al. "Neural message passing for Quantum chemistry". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. 2017, pp. 1263–1272.

[36] Yuan Gong, Yu-An Chung, and James R. Glass. "AST: Audio Spectrogram Transformer". In: *CoRR* abs/2104.01778 (2021). arXiv: 2104.01778. URL: https://arxiv.org/abs/2104.01778.

[37] Jean-Bastien Grill et al. "Bootstrap Your Own Latent - A New Approach to Self-Supervised Learning". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle et al. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/f3ada80d5c4ee70142b17b8192b2958e-Abstract.html.

[38] William L Hamilton, Rex Ying, and Jure Leskovec. "Inductive representation learning on large graphs". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017, pp. 1025–1035.

[39] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[40] Kaiming He et al. "Masked Autoencoders Are Scalable Vision Learners". In: *CoRR* abs/2111.06377 (2021). arXiv: 2111.06377. URL: https://arxiv.org/abs/2111.06377.

[41] Ruining He et al. “RealFormer: Transformer Likes Residual Attention”. In: *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*. Ed. by Chengqing Zong et al. Vol. ACL/IJCNLP 2021. Findings of ACL. Association for Computational Linguistics, 2021, pp. 929–943. DOI: 10.18653/v1/2021.findings-acl.81. URL: https://doi.org/10.18653/v1/2021.findings-acl.81.

[42] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. “Distilling the Knowledge in a Neural Network”. In: *CoRR* abs/1503.02531 (2015). arXiv: 1503.02531. URL: http://arxiv.org/abs/1503.02531.

[43] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-term Memory”. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[44] Aidan Hogan et al. “Knowledge Graphs”. In: *ACM Comput. Surv.* 54.4 (2021), 71:1–71:37. DOI: 10.1145/3447772. URL: https://doi.org/10.1145/3447772.

[45] Cheng Hsu and Cheng-Te Li. “RetaGNN: Relational Temporal Attentive Graph Neural Networks for Holistic Sequential Recommendation”. In: *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*. Ed. by Jure Leskovec et al. ACM, 2021, pp. 2968–2979. DOI: 10.1145/3442381.3449957. URL: https://doi.org/10.1145/3442381.3449957.

[46] Cheng-Zhi Anna Huang et al. “Music Transformer: Generating Music with Long-Term Structure”. In: *International Conference on Learning Representations*. 2018.

[47] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[48] Gautier Izacard and Edouard Grave. “Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering”. In: *EACL 2021-16th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics. 2021, pp. 874–880.

[49] Andrew Jaegle et al. “Perceiver: General Perception with Iterative Attention”. In: *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 2021, pp. 4651–4664. URL: http://proceedings.mlr.press/v139/jaegle21a.html.

[50] Michael Janner, Qiyang Li, and Sergey Levine. “Offline Reinforcement Learning as One Big Sequence Modeling Problem”. In: *Advances in neural information processing systems* 34 (2021).

[51] Mandar Joshi et al. “Spanbert: Improving pre-training by representing and predicting spans”. In: *Transactions of the Association for Computational Linguistics* 8 (2020), pp. 64–77.

[52] Mandar Joshi et al. “TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. Ed. by Regina Barzilay and Min-Yen Kan. Association for Computational Linguistics, 2017, pp. 1601–1611. DOI: 10.18653/v1/P17-1147. URL: https://doi.org/10.18653/v1/P17-1147.

[53] Ibrahem Kandel and Mauro Castelli. “The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset”. In: *ICT Express* 6.4 (2020), pp. 312–315. DOI: 10.1016/j.icte.2020.04.010. URL: https://doi.org/10.1016/j.icte.2020.04.010.

[54] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: *International Conference on Learning Representations*. 2017.

[55] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. "Reformer: The Efficient Transformer". In: *International Conference on Learning Representations*. 2019.

[56] Rik Koncel-Kedziorski et al. "Text Generation from Knowledge Graphs with Graph Transformers". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 2019, pp. 2284–2293.

[57] Harry H Ku et al. "Notes on the use of propagation of error formulas". In: *Journal of Research of the National Bureau of Standards* 70.4 (1966), pp. 263–273.

[58] Miroslav Kubat. "Neural networks: a comprehensive foundation by Simon Haykin, Macmillan, 1994, ISBN 0-02-352781-7." In: *The Knowledge Engineering Review* 13.4 (1999), pp. 409–412.

[59] Zhenzhong Lan et al. "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations". In: *International Conference on Learning Representations*. 2019.

[60] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[61] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural Networks* 6.6 (1993), pp. 861–867. DOI: 10.1016/S0893-6080(05)80131-5. URL: https://doi.org/10.1016/S0893-6080(05)80131-5.

[62] Jing Li et al. "A Survey on Deep Learning for Named Entity Recognition". In: *IEEE Trans. Knowl. Data Eng.* 34.1 (2022), pp. 50–70. DOI: 10.1109/TKDE.2020.2981314. URL: https://doi.org/10.1109/TKDE.2020.2981314.

[63] Xiaohan Li et al. "Pre-training Recommender Systems via Reinforced Attentive Multi-relational Graph Neural Network". In: *2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, December 15-18, 2021*. Ed. by Yixin Chen et al. IEEE, 2021, pp. 457–468. DOI: 10.1109/BigData52589.2021.9671830. URL: https://doi.org/10.1109/BigData52589.2021.9671830.

[64] Xin-Yi Li, Wei-Jun Lei, and Yu-Bin Yang. "From Easy to Hard: Two-stage Selector and Reader for Multi-hop Question Answering". In: *CoRR* abs/2205.11729 (2022). DOI: 10.48550/arXiv.2205.11729. arXiv: 2205.11729. URL: https://doi.org/10.48550/arXiv.2205.11729.

[65] Yujia Li et al. "Gated Graph Sequence Neural Networks". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1511.05493.

[66] Yinhan Liu et al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *CoRR* abs/1907.11692 (2019). arXiv: 1907.11692. URL: http://arxiv.org/abs/1907.11692.

[67] Saeed Masoudnia and Reza Ebrahimpour. "Mixture of experts: a literature survey". In: *Artif. Intell. Rev.* 42.2 (2014), pp. 275–293. DOI: 10.1007/s10462-012-9338-y. URL: https://doi.org/10.1007/s10462-012-9338-y.

[68] Roland Memisevic. "Learning to Relate Images". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 35.8 (2013), pp. 1829–1846. DOI: 10.1109/TPAMI.2013.53. URL: https://doi.org/10.1109/TPAMI.2013.53.

[69] Alessio Micheli. "Neural Network for Graphs: A Contextual Constructive Approach". In: *IEEE Trans. Neural Networks* 20.3 (2009), pp. 498–511. DOI: 10.1109/TNN.2008.2010350. URL: https://doi.org/10.1109/TNN.2008.2010350.

[70] Yimeng Min, Frederik Wenkel, and Guy Wolf. "Scattering gcn: Overcoming oversmoothness in graph convolutional networks". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 14498–14508.

[71] Shervin Minaee et al. "Deep Learning-based Text Classification: A Comprehensive Review". In: *ACM Comput. Surv.* 54.3 (2021), 62:1–62:40. DOI: 10.1145/3439726. URL: https://doi.org/10.1145/3439726.

[72] Nurshazlyn Mohd Aszemi and Dhanapal Durai Dominic Panneer Selvam. "Hyperparameter Optimization in Convolutional Neural Network using Genetic Algorithms". In: *International Journal of Advanced Computer Science and Applications* 10 (June 2019), pp. 269–278. DOI: 10.14569/IJACSA.2019.0100638.

[73] Isabel Papadimitriou, Richard Futrell, and Kyle Mahowald. "When classifying grammatical role, BERT doesn't care about word order... except when it matters". In: *CoRR* abs/2203.06204 (2022). DOI: 10.48550/arXiv.2203.06204. arXiv: 2203.06204. URL: https://doi.org/10.48550/arXiv.2203.06204.

[74] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "Glove: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. ACL, 2014, pp. 1532–1543. DOI: 10.3115/v1/d14-1162. URL: https://doi.org/10.3115/v1/d14-1162.

[75] Matthew E. Peters et al. "Deep Contextualized Word Representations". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*. Ed. by Marilyn A. Walker, Heng Ji, and Amanda Stent. Association for Computational Linguistics, 2018, pp. 2227–2237. DOI: 10.18653/v1/n18-1202. URL: https://doi.org/10.18653/v1/n18-1202.

[76] Thang M. Pham et al. "Out of Order: How important is the sequential order of words in a sentence in Natural Language Understanding tasks?" In: *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*. Ed. by Chengqing Zong et al. Vol. ACL/IJCNLP 2021. Findings of ACL. Association for Computational Linguistics, 2021, pp. 1145–1160. DOI: 10.18653/v1/2021.findings-acl.98. URL: https://doi.org/10.18653/v1/2021.findings-acl.98.

[77] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI blog* 1.8 (2019), p. 9.

[78] Pranav Rajpurkar, Robin Jia, and Percy Liang. "Know What You Don't Know: Unanswerable Questions for SQuAD". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 2018, pp. 784–789.

[79] Pranav Rajpurkar et al. "SQuAD: 100,000+ Questions for Machine Comprehension of Text". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 2016, pp. 2383–2392.

[80] Roshan Rao et al. "Transformer protein language models are unsupervised structure learners". In: *International Conference on Learning Representations*. 2020.

[81] Anna Rogers, Matt Gardner, and Isabelle Augenstein. "Qa dataset explosion: A taxonomy of nlp resources for question answering and reading comprehension". In: *ACM Computing Surveys (CSUR)* (2021).

[82] Gurudutt Rohit, Ekta Dharamshi, and Natarajan Subramanyam. "Approaches to Question Answering Using LSTM and Memory Networks: SocProS 2017, Volume 1". In: Jan. 2019, pp. 199–209. ISBN: 978-981-13-1591-6. DOI: 10.1007/978-981-13-1592-3_15.

[83] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[84] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[85] Franco Scarselli et al. "The Graph Neural Network Model". In: *IEEE Trans. Neural Networks* 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605. URL: https://doi.org/10.1109/TNN.2008.2005605.

[86] Michael Schlichtkrull et al. "Modeling relational data with graph convolutional networks". In: *European Semantic Web Conference*. Springer. 2018, pp. 593–607.

[87] Stefan Schweter and Sajawel Ahmed. "Deep-EOS: General-Purpose Neural Networks for Sentence Boundary Detection". In: *Proceedings of the 15th Conference on Natural Language Processing, KONVENS 2019, Erlangen, Germany, October 9-11, 2019*. 2019. URL: https://corpora.linguistik.uni-erlangen.de/data/konvens/proceedings/papers/KONVENS2019%5C_paper%5C_41.pdf.

[88] Andrew W Senior et al. "Improved protein structure prediction using potentials from deep learning". In: *Nature* 577.7792 (2020), pp. 706–710.

[89] Andrew W. Senior et al. "An empirical study of learning rates in deep neural networks for speech recognition". In: *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*. IEEE, 2013, pp. 6724–6728. DOI: 10.1109/ICASSP.2013.6638963. URL: https://doi.org/10.1109/ICASSP.2013.6638963.

[90] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural Machine Translation of Rare Words with Subword Units". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. DOI: 10.18653/v1/p16-1162. URL: https://doi.org/10.18653/v1/p16-1162.

[91] Min Joon Seo et al. "Bidirectional Attention Flow for Machine Comprehension". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=HJ0UKP9ge.

[92] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. "Self-Attention with Relative Position Representations". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. 2018, pp. 464–468.

[93] Olivier Sigaud et al. "Gated networks: an inventory". In: *CoRR* abs/1512.03201 (2015). arXiv: 1512.03201. URL: http://arxiv.org/abs/1512.03201.

[94] Marco Antonio Calijorne Soares and Fernando Silva Parreiras. "A literature review on question answering techniques, paradigms and systems". In: *Journal of King Saud University-Computer and Information Sciences* 32.6 (2020), pp. 635–646.

[95] Linfeng Song et al. "Exploring Graph-structured Passage Representation for Multi-hop Reading Comprehension with Graph Neural Networks". In: *arXiv* (2018), arXiv–1809.

[96] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. "Highway Networks". In: *CoRR* abs/1505.00387 (2015). arXiv: 1505.00387. URL: http://arxiv.org/abs/1505.00387.

[97] Peng Su et al. "Using distant supervision to augment manually annotated data for relation extraction". In: *PloS one* 14.7 (2019), e0216913.

[98] Sho Takase, Jun Suzuki, and Masaaki Nagata. "Character n-Gram Embeddings to Improve RNN Language Models". In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 5074–5082. DOI: 10.1609/aaai.v33i01.33015074. URL: https://doi.org/10.1609/aaai.v33i01.33015074.

[99] Ming Tu et al. "Multi-hop Reading Comprehension across Multiple Documents by Reasoning over Heterogeneous Graphs". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019, pp. 2704–2713.

[100] Iulia Turc et al. "Well-read students learn better: On the importance of pre-training compact models". In: *arXiv preprint arXiv:1908.08962* (2019).

[101] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[102] Petar Veličković et al. "Graph Attention Networks". In: *International Conference on Learning Representations*. 2018.

[103] Jesse Vig et al. "BERTology Meets Biology: Interpreting Attention in Protein Language Models". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: https://openreview.net/forum?id=YWtLZvLmud7.

[104] Tianming Wang, Xiaojun Wan, and Hanqi Jin. "Amr-to-text generation with graph transformer". In: *Transactions of the Association for Computational Linguistics* 8 (2020), pp. 19–33.

[105] Yu Wang et al. "Application of Pre-training Models in Named Entity Recognition". In: *CoRR* abs/2002.08902 (2020). arXiv: 2002.08902. URL: https://arxiv.org/abs/2002.08902.

[106] Johannes Welbl, Pontus Stenetorp, and Sebastian Riedel. "Constructing datasets for multi-hop reading comprehension across documents". In: *Transactions of the Association for Computational Linguistics* 6 (2018), pp. 287–302.

[107] Stiaan Wiehman, Steve Kroon, and Hendrik De Villiers. "Unsupervised pre-training for fully convolutional neural networks". In: *2016 Pattern Recognition Association of South Africa and Robotics and Mechatronics International Conference (PRASA-RobMech)*. IEEE, pp. 1–6.

[108] Zonghan Wu et al. "A comprehensive survey on graph neural networks". In: *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[109] Bing Xu et al. "Empirical Evaluation of Rectified Activations in Convolutional Network". In: *CoRR* abs/1505.00853 (2015). arXiv: 1505.00853. URL: http://arxiv.org/abs/1505.00853.

[110] Peng Xu, Chaitanya K. Joshi, and Xavier Bresson. "Multigraph Transformer for Free-Hand Sketch Recognition". In: *IEEE Transactions on Neural Networks and Learning Systems* (2021), pp. 1–12. DOI: 10.1109/TNNLS.2021.3069230.

[111] Weiwen Xu et al. "Exploiting Reasoning Chains for Multi-hop Science Question Answering". In: *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*. Ed. by Marie-Francine Moens et al. Association for Computational Linguistics, 2021, pp. 1143–1156. DOI: 10.18653/v1/2021.findings-emnlp.99. URL: https://doi.org/10.18653/v1/2021.findings-emnlp.99.

[112] Zhilin Yang et al. "HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. Ed. by Ellen Riloff et al. Association for Computational Linguistics, 2018, pp. 2369–2380. DOI: 10.18653/v1/d18-1259. URL: https://doi.org/10.18653/v1/d18-1259.

[113] Donghan Yu et al. "KG-FiD: Infusing Knowledge Graph in Fusion-in-Decoder for Open-Domain Question Answering". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2022, pp. 4961–4974.

[114] Manzil Zaheer et al. "Big Bird: Transformers for Longer Sequences". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle et al. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/c8512d142a2d849725f31a9a7a361ab9-Abstract.html.

[115] Xiaohua Zhai et al. "Scaling Vision Transformers". In: *CoRR* abs/2106.04560 (2021). arXiv: 2106.04560. URL: https://arxiv.org/abs/2106.04560.

[116] Victor Zhong et al. "Coarse-grain Fine-grain Coattention Network for Multi-evidence Question Answering". In: *International Conference on Learning Representations*. 2018.