Re-Implementation of Xiao et al. (2019): Dynamically Fused Graph Network for Multi-hop Reasoning

Yu-Wen Chen, Siyana Pavlova, Simon Preissner
Dpt. of Language Science and Technology Saarland University
s8{ynchen|sipavl|siprei}@stud.uni-saarland.de

Abstract

In this project report, we present and partly evaluate our re-implementation of a state-of-the-art neural multi-hop question answering system and comment on our experiences throughout the development process.

1 Introduction

Multi-hop Question Answering requires higher capabilities of the machine to comprehend and reason over the information scattered around several documents to answer the given question correctly. From a human being's perspective, such reasoning process requires information flowing from one passage to another passage and this connection gradually updates and influences human's understanding of the message. As such, Graph Neural Networks (GNNs) is a favorable approach to mimic human's step-by-step reading comprehension and reasoning. Former works applying GNNs to multi-hop Question Answering build fixed global entity graph of each QA pair and thus the reasoning is implicit. Also, for open-domain multi-hop Question Answering the answer may not even exist in the entity graph. Therefore, Dynamically Fused Graph Network (DFGN, Xiao et al., 2019) is proposed to tackle these challenges.

Two unique components play crucial roles of the performance of the DFGN regarding the aforementioned challenges. Firstly, a *Dynamic Entity Graph* is constructed for explicit reasoning and interaction between query and documents. Noisy information from irrelevant paragraphs is filtered out with masks and useful information can be more evident. Secondly, the *fusion process* not only propagates the information from the paragraphs to the entity graph but helps to pass the information from the entity graph back to the documents. The combination of these two particular components helps to construct a more informative entity graph with regard to the cross-interaction of entity and documents and leads to more accurate answers.

We selected this paper as our re-implementation project of the seminar "Advances in Question Answering", taught in the winter semester 2019/20 at Uni Saarland. This report is intended to be a presentation of our re-implementation as well as a documentation of our challenges and the insights we (the authors) had during the project. The documentation is sectioned as follows: we first give an overview of the related work in §2. The HotpotQA data and the whole model setup is discussed in §3. From §4 to §8, we elaborate on the details of the five modules – Paragraph Selector, Graph Constructor, Encoder, Fusion Block and Prediction Layer, and how we bring them altogether. Training details of Paragraph Selector and DFGN are reported in §9. Finally, the results of Paragraph Selector and DFGN model are presented in §10 and conclusion and discussion are in §11 and §12.

2 Related Work

Some components of DFGN are inspired by previous work to simulate the intricate reasoning process of humans to some degree. We looked into the following papers and integrated those models into the DFGN model as indicated in the original paper.

In the Encoder module, a bi-attention layer (Seo et al., 2018) is used to improve the cross interaction between query and context. Bi-Directional Attention Flow (BiDAF) contains six layers to acquire the contextual representations at different levels of granularity. The first three layers model the embeddings at character, word, and contextual levels which apply to both context and query. However, for our implementation, BERT is used to obtain the contextual embeddings and they are fed into the subsequent Attention flow layer. It computes attention in both directions (i.e. query to context

and vice-versa) and outputs the query-aware vector representation of the context words. As BiDAF is implemented in two components (Encoder and Fusion Block) in DFGN, we decide to implement this common architecture as a utility class.

In the Fusion Block, Dynamic Graph Attention, serving as the simulation of human's gradual searching and reasoning behavior, is inspired by Graph Attention Networks (GATs, Velickovic et al., 2017). By stacking graph attentional layers, GATs allow for assigning different weights to different nodes within a neighborhood in different sizes, and outputs higher-levels of node features. Dynamic Graph Attention inherits this property and features of each node are updated according to the information propagated from neighboring nodes.

For the Prediction Layer, DFGN follows the same structure as in the original HotpotQA paper (Yang et al., 2018). Three key components – self-attention, bi-attention, and RNNs – serve as the building blocks of this model. Linear layers are used to output the spans of the answer, supporting sentences, and answer types ('yes', 'no', or 'span-based'.) In the original paper, paragraphs and question are served as separate inputs to the model. But in the DFGN paper, inputs are from the last Fusion Block. It is also worth noting that in (Yang et al., 2018), a binary classifier is incorporated into this model to predict each sentence's probability of being a supporting fact which helps to evaluate the model's explainability. However, (Xiao et al., 2019) do not specify such details so we used a linear layer to compute the supporting scores.

3 Setup & Datasets

As we had no previous experience with large neural architectures and GPU computing, the organization of data underlying this project is a learning experience that seems worth mentioning.

Data Set. As in (Xiao et al., 2019), we use the HotpotQA (Yang et al., 2018) training data set ('trainset', ∼90.5K data points) for training and on-line evaluation and the `dev_distractor` data set ('devset', ∼7.4K data points) for testing. By using part of the trainset for small evaluation rounds during training, we ensure that no test data from the devset is used for training. In each data point, the following information is relevant for our system: the question, the answer, 10 paragraphs (2 relevant and 8 distractors) each headed by a unique title, and the list of supporting facts (title-index pairs that express which sentence in a paragraph, referenced by title, contains a partial answer).

Outputs and Model Storage. We use a separate directory for outputs and models. The PyTorch models we obtain are too large to be moved via git. For this reason, we copy the models from the compute cluster to our local machines via `scp` and upload them to a Google Drive.[1] Our test results (training logs, predictions, test scores) are kept in the same separate directory as to not clutter the repository containing the code.

Repository Structure. For development and code maintenance we use a git repository[2] which contains our training and evaluation scripts as well as four directories: `modules/` for all of the five modules described in (Xiao et al., 2019), `config/` for all configs, `models/` for selected outputs and visualization of results, and `playground/` for scripts and files not directly related to working code.

Parameter Configurations. In order to handle parameters, we implemented a utility class `ConfigReader` which takes a path to a configuration file (short: 'config') and parses its contents. As locally stored files require individual file paths, and hardware differences dictate certain parameters, we use different configs for running scripts locally vs. on the compute clusters. The only parameters that are passed as command line arguments are the path to the config and the name of a directory which will be used for the main outputs. All other paths and file names are specified in the configs.

GPU Cluster. In order to use the university's GPU compute cluster (henceforth: 'jones'),[3] we set up a virtual environment with Python3.6, PyTorch, and Cuda. Jones comprises 4 GeForce GTX Titan X GPUs with 12GB memory. For reasons of complexity, we refrain from multiprocessing and only use one GPU at a time.

[1] https://drive.google.com/open?id=1FZzxpKQGhDzaDjACcPTna117Ope-RKdE

[2] https://github.com/siyanapavlova/Advances_in_QA

[3] we adopt the cluster's GPU nodes' names: 'jones-X'.

4 Paragraph Selector

The goal of the Paragraph Selector is to select only the paragraphs that are relevant to the question. This can be reduced to a classification task that takes a paragraph and a query, and classifies the paragraph as relevant or irrelevant. We designed the Paragraph Selector as a class (`ParagraphSelector`) that wraps around the actual network class (`ParagraphSelectorNet`); this makes it easier to train, evaluate, and apply.

We searched for a while to understand how to encode sentences and queries. This involved reading into BERT and understanding what it actually looks like (attention heads, hidden states and their respective shapes, how to use the classes and methods provided by Huggingface-Transformers,[4] etc). Based on this reading, we use the `CLS` token of the encoded sentence for classification.

The classification layer is taken care of by `ParagraphSelectorNet` which inherits from `torch.nn.Module`. It is a simple neural network that takes a list of token IDs and outputs a relevance score. The token IDs represent a concatenation of a query and a paragraph. We use BERT's `[CLS]` and `[SEP]` tokens to utilize BERT's ability to represent a pair of sentences (cf. Devlin et al., 2018). The score is a measure of how closely related the query and the paragraph are. It is obtained by applying a linear layer with sigmoid prediction to BERT's output, as done in (Xiao et al., 2019).

The optimizer and the loss function used for training by (Xiao et al., 2019) were not specified. However, since they use Adam Optimizer for the other components, we decided to use that for the Paragraph selector, too. For the loss function, we originally considered MSE. However, as pointed out by Rafay Khan[5], MSE is not ideal for classification, thus we chose to use Binary Cross Entropy.

To predict relevance scores for each paragraph with respect to the query, we needed labeled data. We used the supporting facts and the paragraph topics for each paragraph of an item to construct such data. All paragraphs whose topics coincide with at least one supporting fact are assigned a 1 and all the rest a 0.

To construct the context of a data point, we wrote a function (`make_context`), which takes a data point and a threshold, and returns the paragraphs that have a relevance score to the query higher than the threshold. Each paragraph is trimmed so that the context does not exceed a specified text length. This is done for two reasons: (1) we use BERT in a following module (Encoder) to encode the whole context and BERT has a sequence limit of 512 and (2) processing longer sequences is more computationally heavy while not improving the performance much, as demonstrated in §10.

First Rounds of Training/Testing. The Paragraph Selector is one of the first neural models that we have trained and the first one using a GPU, hence the first rounds of training and evaluation were full of both errors and insights.

First, we faced memory issues involving the `with torch.no_grad()` setting for the BERT encoder model. We solved these issues with support from our project supervisor so that training of the Paragraph Selector also fine-tunes BERT. Consequently, we had to drastically adjust the batch size for small training runs on our own devices: we started out training on an Nvidia MX250 GPU (2GB memory) and had to set the batch size to 2 or 3. Apart from the hardware limitations, the implementation probably has the potential to become more efficient.

Another lesson is concerning the evaluation of training. When plotting the losses for models, we observed that even though there is a downward trend, the model apparently does not converge at all, as can be seen in Figure 1. We repeated this experiment with varying learning rates and batch sizes, but could not find a good configuration.

With support from our project supervisor, we figured out the source of our issues. We analyze loss values by averaging multiple subsequent steps from training to obtain a smoother curve. Figure 2 shows the learning curve of one of the small models trained on the GPU mentioned above. Additionally, we implemented a development mechanism that periodically evaluates the model and saves it if there is an improvement.

5 Graph Constructor

Given a context, i.e. a list of paragraphs which in turn are lists of sentences, the Graph Constructor works in five steps: (1) perform named entity

[4] https://github.com/huggingface/transformers

[5] Why Using Mean Squared Error(MSE) Cost Function for Binary Classification is a Bad Idea?

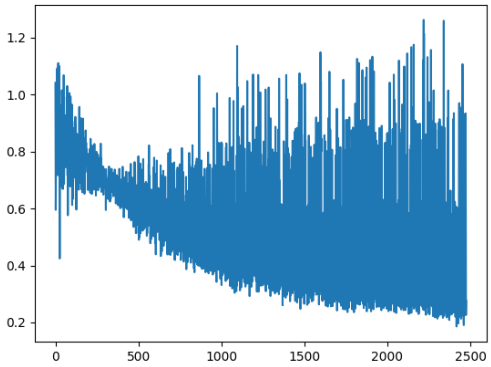

Figure 1: loss values per gradient descent step. Training on 10K paragraphs (1 epoch, batch size: 4) with text length limit of 250 tokens; learning rate: 1e-5

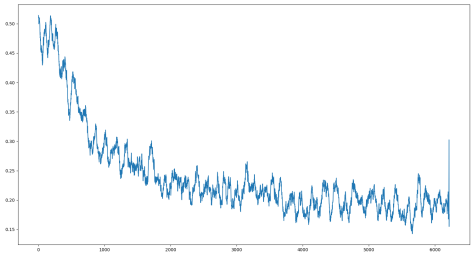

Figure 2: training loss per step (averaged across 50 subsequent steps). Training on 10K questions (=100K paragraphs, no downsampling; 1 epoch; batch size: 16) with text length limit of 250 tokens; learning rate: 1e-5. (values for the last 50 steps are not computed correctly)

recognition (NER) with Flair (Akbik et al., 2019), (2) establish a set of nodes, one per entity mention, (3) connect nodes as described in section 3.2 of (Xiao et al., 2019), (4) prune the graph to the 40 most connected nodes, and (5) construct a binary mapping M between nodes and tokens. Note that step 5 is an important part of the Fusion Block (§7) as it is used to distribute information from entity mentions to their context tokens.

The Graph Constructor (implemented as the class `EntityGraph`) is presented as a relatively straightforward module. However, it showed throughout the project that the work on the interface between strings and tensors is much more intricate than assumed. Therefore, the following is an account of the challenges we encountered.

Early implementations used Stanford's CoreNLP API[6] to a server for NER. As this is only feasible for small amounts of data, we switched to Flair.

Initially, graphs were implemented as lists of tuples, but we soon changed the data structure to dictionaries of dictionaries for reasons of mutability and accessibility. Each node, accessed by its ID, is a dictionary with information about the entity's position, its string form, and its relations to other nodes. Relations are symmetrical, but as the graph only implements unidirectional relations, two related entity nodes a and b each hold one of the relations $R(a, b)$ and $R(b, a)$.

Mapping Tokens to Entities. The largest share of work for this module was spent on the `entity_matrix()` method that implements step 5 from above (relating entities to tokens). The main problem is that Flair, which provides entity strings, tokenizes differently from BERT, which we use for tokenization. We therefore devised an algorithm that uses string comparisons on lowercased strings that have been stripped from spaces as well as BERT's 'wordpiece' tag "##".

It works as follows: we work through all entities (from Flair) in the order of their appearance in text. Iterating over and accumulating the tokens (from BERT), we use `endswith()` to detect when a sequence of tokens has added up to end with the same character sequence as the named entity that is currently processed. In such a case, we construct a string by going backwards from this point, consuming one token at a time, until the string exactly matches the current named entity's character sequence. The number of steps that we had to take back until the constructed string matched the entity is the number of tokens that this entity spans.

This algorithm is slow, but it works almost flawlessly. In multiple iterations of debugging, we ensured that any remaining errors introduced by `entity_matrix()` induce skipping of a training example rather than triggering a potentially fatal runtime error.

6 Encoder

The encoder module obtains contextual embeddings of a concatenation of query and context (from the uncased BERT$_{base}$) and passes these embeddings through a Bidirectional Attention Flow (BiDAF, Seo et al., 2018) achritecture, which is used to enhance the cross interaction between query and context. The code is publicly available;[7] how-

[6] https://github.com/stanfordnlp/python-stanford-corenlp

[7] https://github.com/allenai/bi-att-flow

ever, the implementation is in TensorFlow.

Judging from the BiAttention class in their code, (Xiao et al., 2019) re-implemented the BiDAF algorithm in PyTorch, so we decided to do that, too. We based our version on an open-source PyTorch implementation[8] which serves our needs although it has a slightly lower performance than the original.

We adapted the code by deleting all unnecessary parts,[9] including character embeddings, making minor changes to the `BiDAF` class, and adding another linear layer (output size 300) as the output layer to conform to d_2 from section 3.3 of (Xiao et al., 2019). We call the encoder twice, once to convert the context and once to convert the query.

Length Issues. The context from the Paragraph Selector is limited to a certain length (maximum: 512), but the encoder takes a concatenation of query and context. This means that if context plus query exceed the maximum input length, we need to clip some tokens from the context in order to fit in the query. We then pass everything to the BERT model.

Our solution is not flawless if the custom context length is close to the maximum of 512. For example, if the specified context length is 500 and there is a query of length 20, the Encoder will return a context of length $500 - ((500 + 20) - 512) = 492$. We thus might end up with a context which is shorter than necessary. We solve this by padding. Similarly, in the rare cases when the context we get from Paragraph Selector is shorter than the desired length, we pad it.

7 Fusion Block

The Fusion Block is the core of DFGN. Its description in (Xiao et al., 2019) is more mathematical than other modules and it operates almost exclusively on tensors, so re-implementation required a good understanding of the notation and tensor operations in PyTorch, which we learned and exercised in the process.

Doc2Graph. The function `tok2ent()` creates entity embeddings from token embeddings. For this, it first needs to map tokens onto entities (i.e., construct a binary matrix $M^{L \times N}$[10] from L tokens and N entities). Differently to (Xiao et al., 2019), this step is already performed in the Graph Constructor. As the tokenization of Flair, used for NER, differs from BERT's tokenization, M is an $m : 1$ mapping; one entity can be mapped to multiple tokens, even when it is a single-word entity.

As the original implementation deviates from its description in (Xiao et al., 2019), we met with the project supervisor to be clear about the following steps. To make the N entity embeddings, we start with $\mathbf{M}^{L \times N}$ from the Graph Constructor and a matrix $\mathbf{T}^{L \times d_2}$ of the L context token embeddings from the encoder module. By expanding, transposing, and adding dimensions, we obtain entity embeddings of shape $N \times d_2 \times L$. Then we apply mean and max pooling along the last dimension and concatenate the two pooled representations along d_2 to obtain entity embeddings of shape $N \times 2d_2$, which we finally transpose to $\mathbf{E}_{t-1}^{2d_2 \times N}$ in order to conform to the description in (Xiao et al., 2019).

Sitting down together with our project supervisor encouraged us to try the numerous extremely useful functions for conversion of matrices provided by PyTorch, which alleviate many problems of inconsistency between (Xiao et al., 2019) and their implementation; we just did not know about these solutions before. Apart from that, it was helpful to discuss mean/max pooling and certain other notions in order to get an intuition for these techniques and to be clear about the subsequent implementation.

Dynamic Graph Attention. This part of the Fusion Block was the most mathematical one, thus requiring a multitude of matrix operations. However, thanks to the insights we got while implementing Doc2Graph, we found this rather straightforward to implement. The main confusion we had about this part was when to do matrix multiplication and when to do element-wise multiplication. We managed to solve this by carefully tracking the shapes of each matrix or vector in formulas 1 – 9 in (Xiao et al., 2019).

Graph2Doc. This method is applied after the Dynamic Graph Attention step. It translates the information in the updated entity embeddings to all context embeddings to return a complete context that can be passed to the Prediction Layer or to another iteration of the Fusion Block. Graph2Doc uses the same binary matrix M as Doc2Graph.

[8] https://github.com/galsang/BiDAF-pytorch

[9] We make sure to clearly mark the passages that we copied in the source code.

[10] we use L to denote number of tokens instead of M (as in the original paper) in order to avoid confusion.

Query Updating. In the original paper, to update the query, the authors use BiDAF again. We initially had the BiDAF algorithm as part of the Encoder. In order to be able to use the same code for Query Updating, we decided to refactor the BiDAF algorithm as a utility class. This way we were able to use BiDAF both for the Encoder and for Query Updating without any redundancies.

8 Prediction Layer

(Xiao et al., 2019) follow the same structure of prediction layers as (Yang et al., 2018), namely four LSTMs (one each for: supporting facts, start position of the answer, end position of the answer, and answer type), followed by linear layers. In our case, after the linear layers for start and end position, we also apply softmax. When training, we use four cross entropy losses as specified in formula 15 of (Xiao et al., 2019). They also mention a "Weak Supervision" procedure after the initial prediction and obtain a fifth loss that they add to the objective. However, due to time restrictions and lack of clarity about the exact procedure, we did not implement this part of the paper.

The issues we faced with this module were related mostly to the output sizes. Our confusion was caused by PyTorch's `CrossEntropyLoss()` function requiring different shapes for predicted scores and true labels. Once we managed to sort this out, the rest of the module ran smoothly.

9 Experiments

The following section describes the training setups for various models as well as an account of general challenges which we faced during debugging and testing our implementation. As described in (Xiao et al., 2019), the 5-modules architecture which we call "full network" consists of two separately trained models: the Paragraph Selector (§4) on one side and Encoder, Fusion Block, and Prediction Layer (§6, §7, and §8), which we call "DFGN", on the other. For this reason, we will describe experiments on these two parts in separate sections.

Debugging and Small Experiments. Due to GPU memory issues on local machines, we implemented an option to train on the CPU in order to be able to debug more conveniently. When training on a GPU, the data that is passed to `CUDA` includes all variables in tensor form (token IDs, labels, and the binary matrix M (§5) as well).

As a reminder to the importance of basic Python programming skills, we also found a bug in the ConfigReader utility class (§3): typecasting strings to booleans (e.g., `bool('False')`) checks whether that string is non-empty; it is *not* a typecast that parses the strings "True"/"False" to their boolean values.

During debugging, we restricted the length of each question's context to 50 tokens for faster processing. This resulted in issues for contexts without any detected named entities. In these cases, there is no entity graph and the data point cannot be used for training. We further discovered that some of the questions simply do not contain named entities at all. This underlines one obvious shortcoming of the DFGN architecture in general: it is unable to answer questions which are neither "yes"/"no" questions nor related to named entities. These comprise about 17% of HotpotQA.

Paragraph Selector. As it relies on classifying paragraphs rather than questions, the paragraph selector is trained separately from the other modules. We train with various similar setups; the two most promising models are trained for 3 epochs on 99% of the trainset with a learning rate of $1e^{-5}$. In order to counteract the bias towards irrelevant paragraphs (8:2 per data point), we originally implemented weighting, but never used it for training. Instead, we follow the implementation of (Xiao et al., 2019) by downsampling each point to provide as many irrelevant as relevent paragraphs. Every 40K paragraphs, we evaluate on the remaining 1% and save the model if it improves on F1-score.

For model PS_{250}, we limit each paragraph to the first 250 tokens and train with batch size 16. Model PS_{400} has a per-paragraph token limit of 400 (batch size: 8).

DFGN. The second part of the implementation is a model that combines the Encoder, Fusion Block, and Prediction Layer.

To this end, we process each question (i.e. training example) with a previously trained Paragraph Selector model and pass on the resulting context for training. This obviously introduces noise to the training data. For this reason, we implemented the utility function `make_labeled_data_for_predictor()` which applies the Paragraph Selector to the evaluation data and assimilates the points' supporting

facts to their new contexts before starting the training. This ensures that during training, the DFGN model is evaluated against the same kind of data as it is trained on.

After calculating losses for each label (supporting fact, answer start, answer end, and question type), it jointly optimizes those four cross entropy losses and propagates the error back to the network's learnable parameters. We also clear the current existing gradients before the backpropagation to avoid accumulated gradients. The network's weights are thus updated through each iteration.

Our parameter setting is largely influenced by hardware and time constraints. Multiple test runs of our training script delayed the start of training, so we train on 40K questions from the trainset for 1 epoch with a batch size of 10.[11] Our learning rate is $1e^{-4}$, as specified in (Xiao et al., 2019). Every 300 batches, we evaluate the model on 1% of the training data. As dropout values, we use 0.5 for the Fusion Block and 0.3 for the Prediction Layer. The coefficients λ_{sup} and λ_{type}, not specified in (Xiao et al., 2019), are both set to 0.5. We use the PS_{250} model with a threshold of 0.1 as Paragraph Selector, with a maximum text length of 250.

Note that if NER fails to recognize named entities, the graph is empty, making the data point useless. We track the number of training examples that are skipped for this reason.

10 Results

The Paragraph Selector and the DFGN models are trained and tested separately. The final experiment, however, combines the two parts to the "full network"; we report on the results of the full network in a separate section.

Paragraph Selector. Table 1 shows a comparison of test results with varying thresholds. We compare PS_{250} to PS_{400} and to the scores reported by (Xiao et al., 2019) (precision: 69%; recall: 97%). From table 1 it becomes apparent that the text length restrictions for PS_{250} do not have a negative effect; it performs as well as PS_{400}. However; this could also be an effect of the differences in batch sizes, which lead to twice as many updates for PS_{250}. Downsampling, while effectively reducing the number of training examples to 40% of the available data, has no apparent negative effect.

[11] this is the limit for the 12GB GPUs on jones; at least with our implementation.

Threshold	Precision	Recall	F1-score
0.2	**0.803**	0.927	**0.861**
0.1	0.779	0.945	0.854
0.1*	0.760*	0.951*	0.845*
0.05	0.738	0.952	0.831
0.03	0.714	0.961	0.819
0.01	0.657	**0.976**	0.785

Table 1: Precision, recall, and F1 scores of PS_{250} with different thresholds (we select 0.1 for further training). *scores for PS_{400}

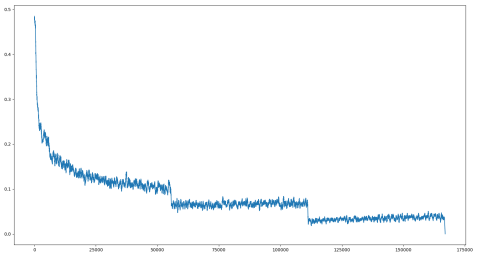

Figure 3: Training loss of PS_{250} per step, averaged across 300 consecutive steps. (values for the last 300 steps are not computed correctly)

In comparison to the model from (Xiao et al., 2019), PS_{250} differs for a threshold of 0.1 both in precision (+9%) and recall (-3%). Note that we train our models with a learning rate of $1e^{-5}$ while theirs is unclear. With a smaller threshold of 0.01, we get *almost* the same scores as they do.

Intuitively, higher thresholds lead to higher precision and lower recall. While it is desirable to have maximum recall, it is not advisable to sacrifice precision, as larger amounts of false positives (i.e., more paragraphs in total) reduces the number of tokens per paragraph, potentially discarding information from valuable paragraphs. For this reason, we choose PS_{250} with a threshold of 0.1 for further experiments.

Further evaluation reveals that after the first epoch, the training loss does not reduce much within, but from one epoch to the next, as becomes apparent from figure 3. The development scores reveal that the model starts training with a very high recall and learns to filter out uninformative paragraphs in a fluctuating manner (figure 4). The jumps in loss values are reflected in spikes at the first and second third of figure 4.

DFGN. At the time of writing this report, the training of our DFGN network is at about 65% complete,

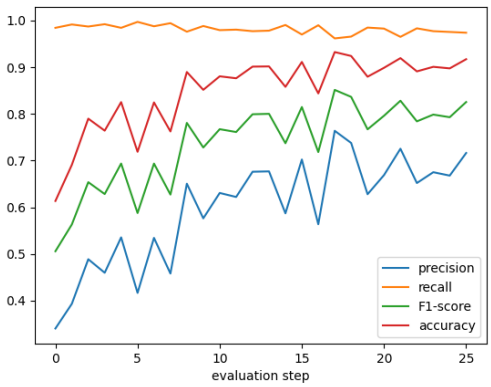

Figure 4: Scores obtained throughout training of PS_{250} (tests on 1% of the trainset).

so we cannot report final results yet. However, we have a model saved from the last in-training evaluation and we can report some results on that.

At this stage of development our network answers "yes" to all questions. We thought that the reason for this was that either there were still some bugs in our code or that the performance will improve with further training (more training data and/or more epochs).

To investigate this which one it might be, we took a look the losses logged during training; they are all `NaN`. We are inclined to think the reason for this lies in our implementation of formulas 6 and 7 from (Xiao et al., 2019). Namely, for some graph nodes we get high β-scores (formula 6) which in turn lead to high exponents (formula 7). This leads to numeric overflow which Python handles by assigning `float('inf')` to the exponents. Therefore, in formula 7 we have a division of two infinities which produces a `NaN` value. This is probably propagated further in the network thus leading to these losses.

Full Network. Without a successfully trained DFGN, we cannot test the full network. The evaluation would have covered aspects of performance, data usage, similarity, and efficiency.

First and foremost, it is interesting how our system will perform in comparison to the original. The numerous differences between the two implementations as well as the fact that our model will have been trained only on a part of the data will probably influence the score.

Secondly, we would like to report on data usage: how many training examples were used in total? A small part of HotpotQA questions does not contain any named entities, rendering them unusable in the first place. Additionally, our system excludes questions from training or evaluation if it fails to detect at least one named entity in the (shortened) context. We would like to report on this behavior.

Thirdly, (Xiao et al., 2019) state that their graphs have an average degree of 3.52. With the length restriction of PS_{250}, we expect our average number of connections per node to be clearly smaller.

Finally, as a weak indicator of the quality of our implementation, we would like to report the times taken for training and evaluation. These accounts could give insights into the potential improvements of the code which could speed up training, for example by enabling larger batch sizes.

11 Conclusion and Future Work

We have managed to almost fully re-implement the system proposed by (Xiao et al., 2019). We have a Paragraph Selector model which matches theirs in performance. While we were not able to report meaningful results for our DFGN implementation due to the issues discussed in §10, we have a complete pipeline and we are confident that these issues can be resolved with a little more work. In the rest of this section we discuss some ideas on how to build upon DFGN.

One possible improvement lies in the answer type prediction (cf. §8). As the classes "yes" and "no" only make up 6% of answers (cf. (Yang et al., 2018)), training might benefit from a weighting of answer type losses according to their distribution.

As we saw in §10, a PS_{250} model works well on its own. However, a shorter context also means fewer named entities recognized and therefore a smaller graph. It would be interesting to compare the results of a DFGN model that works with a PS_{250} model to those of one that works with a Paragraph Selector trained on longer contexts.

There are several fundamental design decisions that were not specified in (Xiao et al., 2019), such as learning rate for the Paragraph Selector, coefficients for optimization, batch sizes, epochs etc. We initially planned to train multiple models to find the best parameter configuration by ourselves. However, the search for answers to these underspecifications slowed down the development process so we could not perform hyperparameter tuning or improve the current implementation in the end.

DFGN is limited to answering questions about

named entities. This may work well for HotpotQA,[12] but in real life scenarios, there may be a lot more questions whose answers fall into neither category. We believe that a DFGN-inspired graph based approach that uses knowledge extraction instead of NER could alleviate this problem and would be an interesting direction of future work.

12 Closing Remarks

In this section, we would like to discuss our personal thoughts and reflect on our experiences throughout the whole working process. As we all were new to PyTorch and GPU computing, we spent a lot of time trying to get the correct shapes of tensors and include/exclude batch sizes in the functions. Especially with the Fusion Block,[13], implemented step by step, carefully mutating the data to follow the description as closely as possible, we gained a lot of practical experience. We now feel stronger with PyTorch programming and matrix manipulation in general. Together with connecting all the modules to a larger system, this resulted in a great learning experience.

The importance of setting up a neat repository structure is also significant for such an intricate model. For example, since there are a lot of parameters to be specified for training, instead of indicating them at command line, collecting all of them into a configuration file makes changing parameters easier at different stages of our work flow. In this way, we can efficiently experiment with different learning rates, batch sizes, fusion block passes, etc, by simply calling the configuration file in the directory `config/`. Also, each module in `modules/` is a class and can be initialized when needed and also integrated together into a bigger model. This arrangement of directory provides a structural outline and helps team communication as it is straightforward to locate which files/directory the teammates are referring to.

From the beginning of the project, we agreed to work together on most aspects of the project rather than split up the work. The reason for this was for each of us to be able to fully meet the learning objectives of this seminar, namely expand our skills and knowledge when working with QA systems and Neural Networks in general. Due to the Covid-19 situation, meeting in person was impossible for the second half of this project. Therefore, we decided to switch to Skype calls 3-4 times per week with screen sharing and peer programming.

[12] About 83% of HotpotQA answers are named entities (NE) or "yes"/"no", both of which DFGN has the potential to answer. If it worked perfectly, DFGN would have a better performance than the current HotpotQA leader. This means HotpotQA's imbalance between NE vs no NE answers allows a system that does not even tackle the second type to be SoTA.

[13] Originally dubbed 'confusion block'.

Acknowledgements

We would like to show our gratitude to the following people for their help:

Our project supervisor, Stalin Varanasi, for numerous tips and assistance throughout the project and the `scp` script to retrieve models from the GPU cluster. Saaddullah Amin for the tip to use Flair for NER in the Graph Constructor. Our fellow students Noon, Morgan, and Kathryn for their manual on setting up PyTorch and Cuda on the GPU cluster, considerably reduced the work on the setup.

References

A. Akbik, T. Bergmann, D. Blythe, K. Rasul, S. Schweter, and R. Vollgraf. FLAIR: An Easy-to-Use Framework for State-of-the-Art NLP. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 54–59, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-4010.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

M. Seo, A. Kembhavi, A. Farhadi, and H. Hajishirzi. Bidirectional Attention Flow for Machine Comprehension. *arXiv:1611.01603 [cs]*, June 2018. arXiv: 1611.01603.

P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. *ArXiv*, abs/1710.10903, 2017.

Y. Xiao, Y. Qu, L. Qiu, H. Zhou, L. Li, W. Zhang, and Y. Yu. Dynamically fused graph network for multi-hop reasoning. *arXiv preprint arXiv:1905.06933*, 2019.

Z. Yang, P. Qi, S. Zhang, Y. Bengio, W. Cohen, R. Salakhutdinov, and C. D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium, Oct.-Nov. 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259.