BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

GRAPH NEURAL NETWORKS FOR DOCUMENT ANALYSIS

ANALÝZA OBSAHU DOKUMENTŮ POMOCÍ GRAFOVÝCH NEURONOVÝCH SÍTÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. NIKOLAS PATRIK**
AUTOR PRÁCE

SUPERVISOR **Ing. MICHAL HRADIŠ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

149893

Institut:	Department of Computer Graphics and Multimedia (UPGM)
Student:	**Patrik Nikolas, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Bioinformatics and Biocomputing
Title:	**Graph Neural Networks for Document Analysis**
Category:	Computer vision
Academic year:	2022/23

Assignment:

1. Familiarize yourself with graph neural networks.
2. Find and study current methods which use graph neural networks for document analysis.
3. Design a suitable method for semantic analysis of documents.
4. Create a suitable dataset or extend one of the existing datasets.
5. Evaluate the proposed method on the dataset.
6. Evaluate the results achieved.
7. Create a short video presenting the results of your work.

Literature:

- Brian Davis, Bryan Morse, Brian Price, Chris Tensmeyer, Curtis Wiginton: Visual FUDGE: Form Understanding via Dynamic Graph Editing. ICDAR 2021.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Hradiš Michal, Ing., Ph.D.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	16.5.2023

Abstract

In this thesis we use for graph neural networks for document analysis. In the beggining we introduce how these graph convolutional networks work and also we introduce concept which is used for their implementation. Next, we explain current solution that solves semantic labeling of text entities in scanned documents, what is also same as the goal of this thesis. In following chapter we present solution which should be used for the mentioned problem as well as another problem which is extraction of specific data using active learning. Gradually, we explain how this solution was implemented and what tools we have used. Before ending, we show our dataset, we have annotated and we meant to use for evaluation and training of our solution. In the end, we present results of this thesis, compare our model with others and also evaluate how our model was able to extract specified data using active learning.

Abstrakt

V tejto práci sa zameriaveme na analýzu dokumentov pomocou grafových neuronových sietí. Na začiatok si predstavíme ako tieto grafove konvolučné siete fungujú a predstavíme si koncept na základe ktorého sa dajú naimplementovať. Ďalej rozoberieme súčasné riešenia ktoré sa zaoberajú semantickým označovaním entít v skenovaných dokumentoch, čo je aj cieľom tejto práce. Následne si predstavíme navrh riešenie ktoré by malo riešiť túto problematiku spolu s ďalším problémom na ktorý sa chceme zamariať v tejto práci a tým je výber textových entít z dokumentov pomocou aktívneho učenia. Postupne si predstavíme ako bolo toto riešenie implementované a aké nástroje sme pritom použili. Pred koncom si predstavíme dataset ktorý sme annotovali pre vyhodnotenie a tréning našeho riešenia. Na záver si predstavíme výsledky tejto práce, porovnáme vysledky s ostatnými prístupmi ktoré sa zamerievajú na podobný problém a ešte vyhodnotíme ako náš model zvládol extrakciu informáciu pomocou aktívneho učenia.

Keywords

graph neural networks, graph convolutional networks, fudge, layoutlm, docformer, layout analysis, document understanding, machine learning, neural networks, funsd, naf, active learning, label studio

Kľúčové slová

grafové neuronové sieťe, grafové konvolučné siete, fudge, layoutlm, docformer, analýza rozloženia dokumentov, porozumenie tlačeným dokumentom, strojové učenie, neuronové siete, funsd, naf, aktívne učenie, label studio

Reference

PATRIK, Nikolas. *Graph Neural Networks for Document Analysis*. Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Michal Hradiš, Ph.D.

Rozšírený abstrakt

Automatické spracovávanie dokumentov je súčasný trend ktorý sa momentálne rieši hlavne pomocou strojového učenia. Spracovávanie dokumentov v sebe ukrýva veľa rozných úloh spracovania prirodzeného jazyka, avšak samotný text niekedy nie je dostačujúci, keďže dokumenty v sebe ukrývajú aj veľa vizuálnej informácie. Táto práca sa hlavne zaoberá dvoma úlohami, ktoré vykonáva nad skenovanými dokumentami. Tieto dve úlohy sú: priradenie textových polí do tried podľa ich významu a ďalej extrakciou informácií z dokumentov podľa danej špecifikácie.

Na začiatok v tejto práci predstavujeme ako fungujú grafové neurónové siete. Tieto metódy si taktiež porovnáme s klasickými metódami strojového učenia, kde zároveň vysvetlíme aké výhody v sebe skrýva strojové učenie nad grafmi. Následne si tiež predstavíme spôsob implementácie týchto grafových vrstiev ktoré sa využívaju v grafových konvolučných sieťach.

V ďalších kapitolách si opíšeme súčasné riešenia ktoré sa zaoberajú podobnou problematikou a taktiež si predstavíme datasety na ktorých boli tieto riešenia vyhodnotené. V tejto kapitole si predstavíme aj riešenie menom FUDGE, ktoré sa zameriava na podobnú problematiku ktorú riešime v tejto práci. Taktiež využíva grafové neuronové sieťe s veľmi zaujímavou architektúrou ktorá vykonáva zmeny grafov ako sa daný model učí. Ďalej si predstavíme tiež zaujimavý prístup ktorý využíva učenie sa vstupných príznakov tak aby najlepšie reprezentovali daný textový prvok v dokumente.

Následne si predstavíme návrh nášho riešenia, ktoré sa skladá z dvoch krokov. Prvým krokom je prevod skenovaného obrázku do grafu, ktorý použijeme ako vstup pre náš model. Ďalšia časť nášho riešenie je potom model ktorý sa skladá z grafových konvolučných vrstiev. Mimo daného návrhu riešenia si taktiež predstavíme ako je náš repozitár štrukturovaný alebo aj aké nástroje sme použili. Taktiež spomenieme aké knižnice nám pomohli dané riešenie naimplementovať.

Potom sa postupne dostaneme k procesu označovania nášej dátovej sady do tried ktoré sme si sami zaviedli pre najlepšie vystihnutie rozloženia textových prvkov v dokumente. Ďalej si predstavíme pravidlá ktoré sme museli dodržiavať aby prvky v daných dokumentoch boli označené čo najkonzistnejšie a zároveň najsprávnejšie keď sa jedná o rôzne druhy dokumentov.

Nakoniec tejto práca, vyhodnocujeme naše riešenie nad základou dátovou sadou ktorý využívali aj riešenie ktoré sme si predstavili a ktoré sa zaoberajú podobnými úlohami. Mimo iného sme taktiež vyhodnotil náš model aj nad dátovou sadou ktorú sme vytvorili a taktiež sme na nej otestovali aj extrakciu predefinovaných informácií. Žial počas testovania sa zistilo že daný dataset nie je vhodný na vyhodnocovania danej úlohý kedže neobsahoval priliš často špecifické prvky v daných dokumentoch. Na základe tohoto zistenia sme sa rozhodli využiť inú dátovú sadu ktorá spĺnovala toto kritérium. Výsledky žial ukázali že náš model nebol schopný extrahovať informácie z dokumentov tak ako sme čakali.

Graph Neural Networks for Document Analysis

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Michal Hradiš Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Nikolas Patrik
May 16, 2023

Acknowledgements

I would like to thank my supervisor for the knowledge he gave me which lead me to tackle all the challenges I have met during development with less struggle. Also, I would gladly thank my friend, Andrej Zaujec, who despite my inability to start working on this thesis always pushed me hard so I managed to finish it all thanks to him.

Contents

List of Figures

Chapter 1

Introduction

Document understanding is a very widely used term for various NLP tasks, such as document classification, entity labeling in the document, information extraction, or layout analysis. Most of these tasks can also be divided into two separate classes based on how the document data are provided to us. First, there is a visual task where we retrieve these documents as images, and we have to retrieve all the information from them first. There are also provided documents that are already as text, and there is no need for pre-processing as in the previous ones. This thesis aims to perform visual layout analysis for various types of documents which means that we retrieve the needed information from images, in our case they are scanned documents. We will retrieve this information from various types of documents, not only a single example of the document, which is called templating. This thesis aims to use for the previously mentioned task regularly a new tool in the machine learning toolbox which is called Graph Neural Networks.

In the following chapters, we will gradually introduce the whole approach to solving this task. Beginning with Graph Neural Networks 2, where machine learning on graphs is introduced. More specifically in this chapter, it is described how machine learning on graphs compares to traditional approaches. We will introduce a machine-learning approach for graphs called Graph Convolutional Networks and a framework for their implementation, the GN block.

Afterward, in chapter 3, we will describe various approaches which relate directly to this thesis, either by solving the same or similar task or using graph neural networks for document understanding. Firstly, we will briefly introduce datasets on which these approaches are evaluated. After that, we describe an interesting solution that uses Graph Convolutions Networks to determine a layout of the document which is called FUDGE. Next, we talk about solutions that use embedding learning using transformer architectures LayoutLM and LayoutLMv2. At the end, we briefly describe the Docformer architecture which uses different approaches that we have met in previous solutions. Lastly, we described some of the pretraining tasks that these solutions trained to learn the embeddings.

Furthermore, this thesis proposes its own approach in chapter 4. In the beginning, we describe what tasks this thesis aims to solve. Next, we describe the whole process of how the given image is converted into a graph which can be then used in the Graph Neural Network framework. And the end of this chapter we will introduce our machine learning model on graphs which was used to solve previously introduced tasks.

Then follows the chapter with implementation details 5. Here we introduce various tools and frameworks which were used during the development of this thesis. We present the repository structure and how the code is structured. Also, we describe how the solution

is split into services to work independently as a whole. We are ending this chapter with a description of the training process and how we used a tool called Tensorboard to carefully monitor the training process. Also, we mentioned the evaluation process and tool we implemented to more carefully analyze our model's predictions.

In the next chapter 6 about reannotating FUNSD dataset we describe how the annotator should properly and consistently choose labels in our dataset and provider them with these guidelines. We properly describe each label and the conditions it has to meet to be assigned the given label. Lastly, we present the created dataset.

The last chapter is about experiments we have done. We tested our solution on the basic FUNSD dataset, in order to compare it with other solutions which are mentioned in chapter 3. After that, we test our solution on a re-annotated dataset but here we only examine the layout of the pages. Lastly, we experiment with active learning and test how our solution can handle the second task we specified, which is the extraction of specific data. We test this by mimicking similar behavior on our re-annotated dataset and after that, we test it out also in a real-life scenario.

This thesis should provide a reader with knowledge of how graph neural networks work, and what are the current solutions which are solving some of the document understanding tasks. The reader will also discover how this document analysis can be done using Graph Neural Networks. This thesis also introduced the most general approach for retrieving data in various types of documents and how to best specify this task so our machine learning model can learn this information to retrieve data the user has specified to it.

Chapter 2

Graph Neural Networks

The modern machine learning toolbox is designed for simple sequences or grids (text or sound and images). But some data can not be represented as these simple structures. Some real-life data are usually in the form of networks of arbitrary size and complex topological structure, most of the time with no space locality. These networks can be represented as mathematical structures called graphs. An example of these might be social networks where we want to connect users with the same interests and thus predict what some users might like. Another example is a knowledge-based graph with protein structures and their side effect, treatments, etc. Based on these relationships, we can predict the properties of other proteins with similar features. In the following sections, we will define graph structure properly, briefly mention how the modern machine learning toolbox works with structured data, and apply these principles to graphs. Afterwards, it is followed by a section, where we introduce how graph neural networks might be implemented with the framework called Graph Network (GN) block.

2.1 Modern machine learning toolbox

To a large extent, the task of document understanding uses datasets that have strictly defined classes for entities and relationships between these entities. This is defined as supervised learning task, which for the given input $\mathbf{x}$ tries to predict label $\mathbf{y}$. Modern machine learning takes supervised learning tasks as an optimization problem. We usually define some objective function $\mathcal{L}(\mathbf{y}, f(\mathbf{x}, \theta))$ which represents loss on current dataset where y represents the ground true label and $f(\mathbf{x}, \theta)$ is some model with model parameters θ. The goal of supervised learning is find parameters of model θ such that function $\mathcal{L}$ takes on the smallest possible values. This is defined as expression:

$$\min_{\theta} \mathcal{L}(\mathbf{y}, f(\mathbf{x}, \theta)) \tag{2.1}$$

There are many existing loss functions defined based on the task we trying to solve. For finding best the parameters of our model we use gradient descent (in practice stochastic gradient descent) where we firstly find the gradient of our objective function defined as follows:

$$\bigtriangledown_{\theta} \mathcal{L}(\mathbf{y}, f(\mathbf{x}, \theta)) = (\frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}, \dots) \tag{2.2}$$

Gradient defines the direction in which function has the highest increase; thus, we must move in the opposite direction to minimize an objective function. This movement in the

opposite direction is called optimizing model's parameters θ. It is defined as mathematical expression,

$$\theta_{new} \leftarrow \theta_{old} - \eta \bigtriangledown_{\theta_{old}} \mathcal{L} \tag{2.3}$$

where η parameter is called learning rate, which tells us how fast we move in the opposite direction of the gradient.

This whole process is then called supervised learning. Most of the current real-world machine learning problems, such as text recognition, image classification, or object detection, are solved by this approach.

2.1.1 Shortcomings of modern machine learning approaches for graphs

Assume we have graph defined as $\mathcal{G}(V, A, X)$, where V is set of nodes, A stands for binary (connected or not connected) adjacency matrix and finally X is matrix of node features $X \in \mathbb{R}^{m \times |V|}$. Also, we define $N(v)$, which represents the set of neighbors of node v.

A naive approach[8] using the current machine learning toolbox could be concatenating a row of adjacency matrix for a given node with its features and feeding it into the neural network. However, this approach has several issues with this idea. In the first place, it has $O(|V|)$ model parameters, which might seem like not much, but for example, social network graphs with billions of users (nodes), will result in really complex model with more than billions parameters. Another huge issue is that our neural network has a fixed input layer size. For example, if a new user registers to the social network, it would be needed to retrain our neural network. Combined with the first issue, it would be impossible to maintain this model because training might take longer than the average period of new users registering. Furthermore, the biggest issue is that this approach is sensitive to node ordering, which might cause invalid predictions for isomorphic graphs with different node ordering, which is not the most desirable solution when dealing with graphs. More preferable solutions make predictions on isomorphic graphs equivalently, thus called permutation invariant or so-called equivariant.

Another approach[8] which we propose for machine learning on graphs inspired by the current machine learning toolbox is using graph convolutions similar to those on images. The goal is to generalize convolutions beyond simple latices, as convolutions for sequences and images take advantage of the fact that these structures can be represented as graphs with the notion of spatial locality. On the other hand, real-world graphs do not have the notion of locality, and as mentioned with the previous approach, they are permutation invariant. In the next section, we will present how the operation of convolution can be applied to graphs.

2.2 Graph Convolutional Networks

The principle on which Graph Convolutional Networks (GCNs) are based on is that each node has defined its computational graph[8]. This idea is to learn to feed-forward information of neighboring nodes to one we calculate and aggregate it with its value. This way, each node of the graph has embedding that includes information about itself and the nodes neighboring with it. As neighboring nodes also take information from their neighbors, they practically retrieve information from the whole graph. This information flow in the graph is a great way to define a machine learning task. The point is to learn how this information

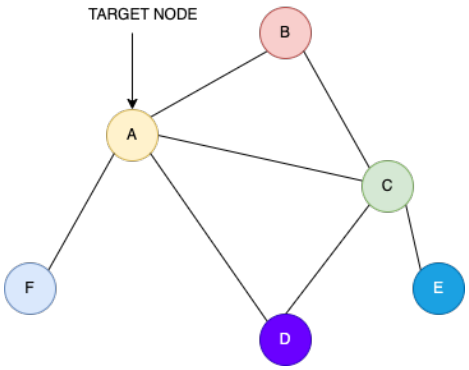

Figure 2.1: Undirected graph

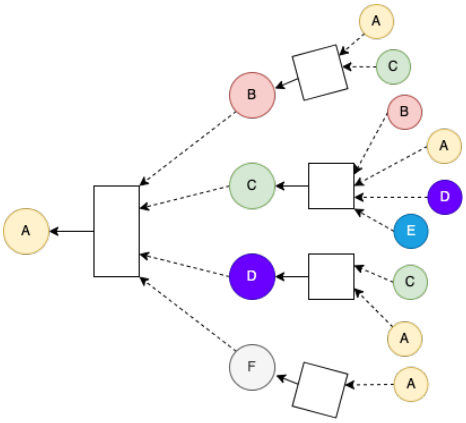

Figure 2.2: Computational graph for given undirected graph from 2.1

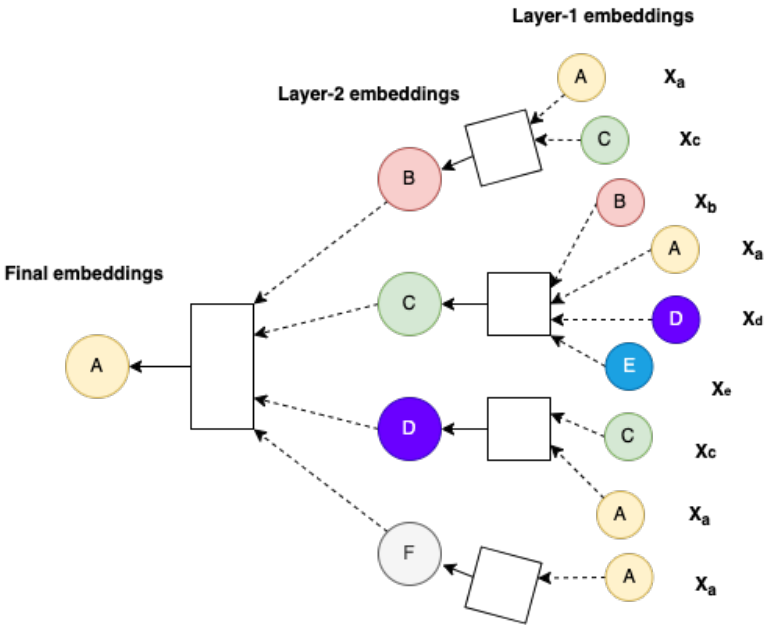

Figure 2.3: Layer-1 embeddings are the input features X_n of given node n as shown. Afterwards each embedding are calculated by GNN layer shown in picture 2.4

should flow in a graph to make a decision based on that. Example of how this computational graph is created from given undirected graph is shown in figures 2.1 and 2.2

Computational graphs consist of several layers. The basic idea of each layer is to aggregate information from neighboring nodes and process this message into the final embedding of the calculated node. The model can be of arbitrary depth and for each layer has each node its embeddings. At first layers, each node has embedding in the form of its input features as shown in figure 2.3. Each subsequent layer embedding is then calculated by aggregating embedding from previous layer nodes applying some transformation over this embedding as function (in this case, the function is neural network). This calculation is done by graph neural network layer, which is shown in figure 2.4.

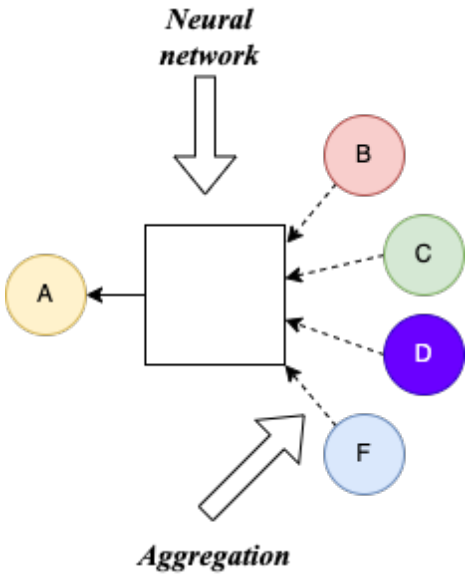

Figure 2.4: Process of transforming embeddings from one layer to the next.

In the calculation, firstly, aggregation takes place. This aggregation function must be permutation invariant; otherwise, it would produce different results for other nodes ordering even though the graph's meaning would remain the same. Such functions are, for example, sum, average, maximum. The next step is then applying some function. In this case, we use a neural network. This way, new embedding for node `A` is produced. For example, as an average of embedding of nodes `B, C, D, F` and then applying neural network will yield new feature representation for node `A`. Mathematically, it would be represented as following equation[8]:

$$h_v^{(k+1)} = \sigma\left(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} + B_k h_v^{(k)}\right), \forall k \in \{0, \ldots, K-1\} \tag{2.4}$$

By decomposing this equation into several blocks, we can see how the whole process described earlier is included within this equation. Layer one embeddings we simply define as input features of the nodes $h_v^0 = x_v$. Let assume we have then the model of K layers of graph neural layers, then the final embedding of the node will be $h^(K)_v$. As in every neural network, we define non-linearity for this equation by using function $\sigma(\ldots)$ functions. Then we represent an average of neighbor's previous layer embeddings as expression below, where $N(v)$ is set of neighboring nodes and h_u^k are embeddings from previous layer for each adjacent node for node v. Summing over these embeddings and dividing them with cardinality of neighboring nodes set gives us average, which we multiply with matrix W_k:

$$W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} \tag{2.5}$$

The matrix W_k along with the matrix B_k are trainable parameters. As shown in the equation, we also add embeddings h_v^k of the computed node k with the average of other messages. Before adding it together, we multiply the embedding of the current node by

this trainable parameter B_k, which represents the weight matrix of a hidden vector of self. The W_k represents the weight matrix for neighborhood aggregation.

2.2.1 Training of graph neural networks

Training of graph neural networks is relatively simple and nearly nothing different from classical convolutions networks. All that is needed is to define the correct loss function for our problem. As mentioned in this thesis, we will be dealing with a visual document understanding problem, an example of supervised learning where we classify entities (nodes) and relationships (edges) into various categories. However, currently, our model of graph neural networks holds information only about nodes' embeddings. This shortcoming of this approach will be solved by using a framework called Graph Network block, which is introduced in the next section 2.3.

If we define generally loss function as $\mathcal{L}(y, f(x, \theta))$, where y is label of our entity and x is its input feature vector, we can then process learning parameters θ as it was described in section 2.1. The problem is how to define function $f(x, \theta)$ for graph layers we introduced in this section. Our graph layers for each node assign new embeddings, which are not comparable with labels for our answer. The solution to this inconvenience is inspired by the convolutional networks we know, and by adding linear layers to our model, we will transform node embeddings into correct labels. This way, we can easily calculate the gradient of our loss function and update parameters, respectively.

2.3 Graph Network block

Graph Network block [2] represents a framework in which we define a class of functions for relational reasoning over graph-structured representations, also called the „graph-to-graph“ module. This framework provides us with representing graph neural networks layers generally and also enables us to change behavior by simply editing the set of functions it uses. This module takes a graph as input, performs calculations over the structure, and returns a graph as output.

The issue with the previous definition of graph layers was having a graph with information stored only on nodes but not edges. This framework considers edge information and global information about the whole graph. Also, this graph can have directed edges that can be useful in various tasks but can be simply converted to an undirected graph by averaging both directions of edges into one.

Firstly we have to redefine our graph definition as 3-tuple $\mathcal{G} = (\mathbf{u}, V, E)$, where $\mathbf{u}$ is an global attribute of graph, V is set of node as in the previous definition and E is set of directed edges defined as $e \in E; e = (\mathbf{e}, r, s)$, where $\mathbf{e}$ is edge attribute and r and s are indexes of receiver and sender nodes. Illustrated view on this kind of graph is shown in figure 2.5, where V and E set would be represented as:

$$\begin{aligned} \mathcal{G} = (\mathbf{u}, V, E); V = \{A, B, C, D\}; \\ E = \{(\mathbf{d}, A, C), (\mathbf{a}, B, A), (\mathbf{c}, B, C), (\mathbf{f}, C, A), (\mathbf{b}, C, B), (\mathbf{g}, D, C)\}. \end{aligned} \tag{2.6}$$

With a defined structure of the graph, we can afterward introduce computation inside graph network block in algorithm 1.

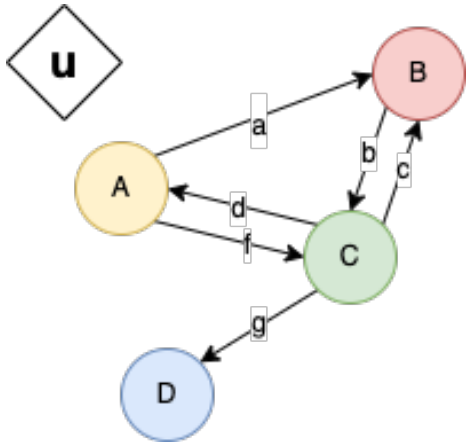

Figure 2.5: Directed graph for GN block

Algorithm 1 Steps of computation in a full GN block taken from [2]

function GRAPHNETWORK(E, V, $\mathbf{u}$)
  for $k \in \{1 \dots |E|\}$ **do**
    $\mathbf{e}'_k \leftarrow \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{r_k}, \mathbf{u})$ ▷ 1. Compute updated edge attributes
  end for
  for $i \in \{1 \dots |V|\}$ **do**
    let $E'_i = \{(\mathbf{e}'_k, r_k, s_k,)\}_{r_k=i, k=1:|E|}$
    $\overline{\mathbf{e}}'_i \leftarrow \rho^{e\rightarrow v}(E'_i)$ ▷ 2. Aggregate edge attributes per node
    $\mathbf{v}'_i \leftarrow \phi^v(\overline{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$ ▷ 3. Compute updated node attributes
  end for
  let $V' = \{(\mathbf{v}'\}_{i=1:|V|}$
  let $E' = \{(\mathbf{e}'_k, r_k, s_k,)\}_{k=1:|E|}$
  $\overline{\mathbf{e}}' \leftarrow \rho^{e\rightarrow u}(E')$ ▷ 4. Aggregate edge attributes globally
  $\overline{\mathbf{v}}' \leftarrow \rho^{v\rightarrow u}(V')$ ▷ 5. Aggregate node attributes globally
  $\mathbf{u}' \leftarrow \phi^u(\overline{\mathbf{e}}', \overline{\mathbf{v}}', \mathbf{u})$ ▷ 6. Compute updated global attribute
  return (E', V', **u**')
end function

GN block[2] consist of set of 6 function, where 3 function ϕ are called „update“ functions and other three ρ functions are for aggregation. In following list we specify what each of these functions takes care of:

- ρ^e across all edges computes per-edge update
- ρ^v across all nodes to compute per-node update
- ρ^u global update of global attribute $\mathbf{u}$
- ϕ^x functions takes set of elements and do specified aggregation over it

The properties of the aggregation function must be invariant to the permutation of input and be able to consume a variable number of arguments.

Computation in GN block[2] proceeds from edge level, to nodes and finally global attribute. Algorithm 1 does following steps with graph provided as input:

1. for each edge we apply function ϕ^e with edge attributes vector, sender and receiver nodes attributes vectors and our current global attribute as parameters.
2. for each node in graph there are aggregated newly computed edge attributes that are connected to given node
3. then we use this edge aggregation result, current node attribute vector and global attribute to compute new node attribute for given node
4. finally we do the aggregations for each newly computed edges and nodes
5. these aggregation are then feed forwarded into global attribute update along with old global attribute
6. GN block then returns whole new graph with new node, edge and global attributes.

GN block has excellent expressive power for graph neural networks due to customization of update and aggregation functions, thus making it an excellent tool for various machine learning tasks on graphs.

Chapter 3

Current approaches to document understanding

Visual document understanding or simply document understanding is a machine learning task that has really great support of its development due to the high amount of digitization of official documents nowadays. The point of this task is to process large volumes of data, typically scans of these documents, and store the information contained in those, thus creating a database with this data. While this task is effortless to implement for precisely defined documents (might also be called templating), the success rate of this approaches significantly decreased when other types of documents are feed forwarded to the procession. Due to the high number of types of documents and the trend of constantly changing official documents along with laws and other factors, it would be impossible or at least inconvenient to have a model for each document. Also, it would require a vast amount of annotated data for each type of document. This process of training a new model and annotating data for each of the official documents would be so tiring for everyone involved that it would be undesirable even to try so.

In the present-day, multiple solutions are trying to implement the task of document understanding on generally every type of official document. However, the reliability of these solutions is much lower than the ones using the templating approach. Moreover, these approaches differ in the used architecture of machine learning models and in using the input features.

In the next section, we describe datasets used to evaluate these approaches. In the following sections, we will introduce an approach called FUDGE[4] which is the only approach using graph neural networks for document understanding. Also, we mention current state-of-art approaches as LayoutLM[22] and LayoutLMv2[21] and also one other approach called DocFormer[1] which uses the compelling architecture of transformers.

3.1 Benchmarking datasets

The two most known datasets, NAF and FUNSD are used in the most of the articles dealing with document understanding.

NAF National Archives Forms dataset, also called NAF, was released the first version in conjunction with the paper „Deep Visual Template-Free Form parsing“[3] and afterwards new updated version was used in paper „Visual FUDGE: Form Understanding via Graph

Editing“ [3]. However, these are only two papers that evaluated its models on this dataset. The dataset contains varied layout forms, with train validation and test sets having disjoint form layouts. Also, these images contain a lot of noise because of the degradation of the machinery used to print them. Most of the images contain entities written by hand, and some are written by hand entirely. NAF dataset is split into the training set with 708 images, 75 validation examples, and 77 test set images. In comparison with FUNSD dataset this pretty large dataset, which is the reason why we decided not to continue using it because reannotation for our specific task would be very time-consuming.

NAF dataset contains two main types of entities: fields and inputs. Also, text entities are not created. However, text lines that are part of the same text entity are connected by relationship. There is also a relationship that connects fields with their corresponding inputs.

FUNSD Form understanding in noisy scanned documents (FUNSD) [7] dataset aims to require structured content of form documents. The dataset comprises of 199 real fully annotated form documents. The dataset consists of two named entities called: „question“ and „answer“, which are then connected by a relationship. Also, there are two more entities that could not be linked to each other. These were: „header“ and „other“ entities. FUNSD is split only into training and test set. FUNSD compared to the NAF dataset, is relatively small. Due to this fact, we decided to first evaluate our model on this simple dataset because every of the further mentioned approaches as LayoutLM[22], LayoutLMv2 [21], and DocFormer [1] uses this dataset for evaluation of visual document understanding tasks. Also, we decided to re-annotate this dataset for our specific task which will be mentioned in the following chapter.

3.2 FUDGE

FUDGE, which stands for Form Understanding via Dynamic Graph Editing [4], is an approach that uses Graph Convolutional Networks for dynamic graph editing. Unlike other approaches, it makes heavy use of spatial features, which also use embeddings generated from language modeling approaches such as BERT[5].

3.2.1 Architecture

FUDGE architecture consists of several components. Firstly text line detection takes place after the edge proposal network predicts edges for each text line. This way, we retrieve our initial graph. The series of graph convolutional networks (GCN) composed from several GN blocks from 2.3 makes edits on these initial graph structures. Finally, after third GCN process modifications of our graph, we retrieve our final graph representing entities contained in the form with relationships between them. This whole process is illustrated in picture 3.1, and in the following subsections, we present how each of these is implemented.

Text line detection FUDGE uses a fully convolutional detector with YOLO[11] predictor head. This detector is pre-trained on a large dataset and then fine-tuned with the rest of the model.

Edge proposal The edge proposal network is a simple two-linear layer network with ReLu activation in between, predicting edges in the initial graph. It receives features of

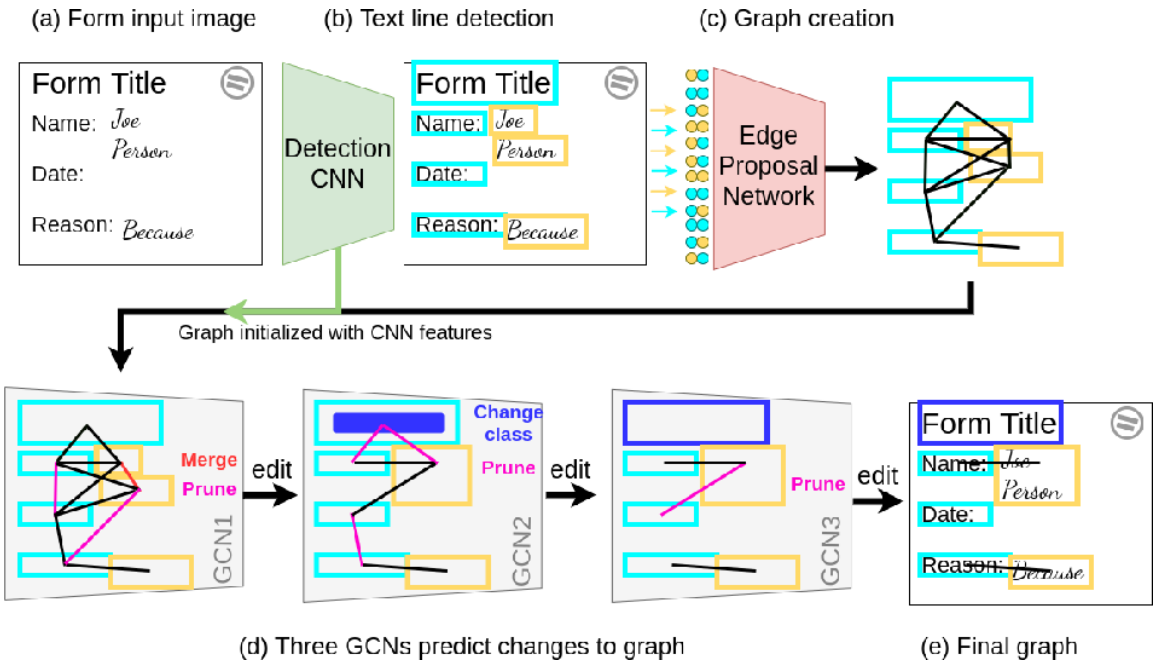

Figure 3.1: Overview of FUDGE model. From a form image (a) text line detection (b) is performed. Then (c) an edge proposal score is computed for each possible edge. After thresholding the scores, the remaining edges form the graph. The graph is initialized with spatial features and features from the CNN detector. A series of GCNs are run (d), each predicting edits to the graph (pruning irrelevant edges, grouping text lines into single entities, correcting oversegmented lines). The final graph (e) is the text entities and their relationships. This figure was taken from [4]

each pair of detected text lines and predicts the likelihood of these two nodes having an edge between them. Half of the relationships sorted based on the score are then used for the initial graph. The features are differences of x and y positions for all corresponding corners and also its centers, including L2 distance of these, height and width of text lines, normalized x and y position for both bounding boxes, whether there is a line of sight between the boxes, and the detection confidences and class predictions for both boxes. We predict both permutations of the pair orders and average them.

Feature extraction FUDGE uses graphs with features both on their nodes and edges because of GN block uses. The edge proposal network gives us the initial structure of the graph, but the feature extraction step gives each node and edge its feature vector. FUDGE uses both spatial and visual features; however, there are no language features.

Iterated graph editing with series of GCNs Graph editing is performed with a series of three GCNs, where each of them predicts various graph modifications. GCNs are composed of several GN blocks; however, GN block requires the graph to have directional edges, so each edge from the initial graph is duplicated. Afterward, when the final graph is created, we average these directional edges into one undirected edge. GN blocks that FUDGE uses are defined without the global attributes and also it uses attention for aggregation of edge features for the node update. Functions that GN block uses are specified

as follows. **GN block edge update** concatenates the edge feature with its connected nodes' features, and this feature vector is then passed through a 2-layer fully connected ReLu network. The output is then summed with the previous features(residual) to produce the new edge features. **GN block function** for aggregation of edges then is defined as multi-head attention (using four heads) [18] where the node's features are used as the query and its edges' features are used as the keys and values. And finally, **GN block update node function** first appends the aggregation of the edge features with the node's current features, and this is feed-forwarded into a two-layer linear, fully connected ReLu network. The output is then summed with edges and its previous features(residual) to produce the new node features. The first two GCNs have seven layers, and the last has 4, which is due to most decisions about graph editing being done in the first two iterations.

Each node then predicts the class for all detected entities. Each edge can predict whether the edge should be pruned or if the entities should be grouped together into a single entity. Similarly, if the detected text lines are over-segmented and should be merged, or if it is an actual relationship.

3.3 LayoutLM, LayoutLMv2 and Docformer

.

In the following section, we will introduce three somewhat similar approaches. All of these mentioned approaches use pre-training for their model and then fine-tuning it on the downstream task.

LayoutLm LayoutLM[22] takes BERT [5] as its backbone. The BERT model is an attention-based bidirectional language modeling approach. The architecture of the BERT model consists of a multi-layer bidirectional Transformer encoder. As input, it takes a list of tokens processed using WordPiece[19]. The BERT model can then generate a contextualized representation of the text. However, for visually rich documents, spatial information is missing. LayoutLM, due to this reason, extends the BERT model with 2-D positions embedding layers for each corner of the bounding box of the given detection. This approach is then combined with image embedding layers, where these embeddings are modeled as tokens of image features from the Faster R-CNN[13]. LayoutLM gives us the final representation for several NLP tasks in visually rich documents. This thesis aims at the form understanding task, which is done on the FUNSD dataset [7]. FUNSD dataset consists of two tasks, semantic labeling task, and semantic linking. LayoutLM aims only to solve semantic labeling. This thesis also aims to solve this task. LayoutLM's final representation is then fed forwarded into a linear layer followed by a softmax layer to predict the label of each token.

LayoutLMv2 The LayoutLMv2[21] is a fairly similar approach as its predecessor. It uses its own multi-modal Transformer architecture as the backbone. Also, this paper introduces its own model of spatial-aware self-attention for better modeling of document layout. Input embedding for this backbone consists of text embeddings, which are created by following common practice using WordPiece[19], visual embeddings, and layout embeddings. LayoutLMv2 also uses two embedding layers to embed the x-axis and the y-axis separately. Layout embedding is then given as

$$l_i = Concat(PosEmb2D_x(x_{min}, x_{max}, width), PosEmb2D_y(y_{min}, y_{max}, height)),$$

unlike in LayoutLM where each of these coordinates was embedded separately, without taking into account the width and height of the bounding box. Finally, the visual embeddings are created as output features of a CNN-based visual encoder. For this purpose ResNext-FPN[20] is used as backbone. For visual embeddings, the image is resized to 224x224 and fed forwarded into the given visual backbone. Then it is averaged-pooled into fixed-size $W \times H$ and flattened into a sequence using a flattened layer. Finally, the linear layer projects these embeddings into embedding space, the same as the text embedding has. These embeddings are then added together along with 1D position embedding in the sequence and given segment embedding to form final visual embeddings.

DocFormer The Doc-Former[1] architecture is again really similar to previously mentioned approaches. It uses Transformer Encoder Multi-Modal training, more precisely using a Discrete Multi-Modal approach. The DocFormer unties visual, text, and spatial features using this architecture. This is done as residual connections for each transformer layer where spatial and visual features are passed. DocFormer does it because spatial and visual dependencies might differ across layers. In each layer, visual and text feature undergoes self-attention with shared spatial features. The spatial features or layout features as they are called in LayoutLMv2[21] are encoded coordinates of the top left and right bottom corner using two embedding layers, similarly as in LayoutLM[22]. Visual features are also done nearly the same way as in LayoutLMv2[21] however, to the flattened linear projection embedding, there are no added embeddings of 1D position and segment. Text features are generated precisely, just like in the two previous approaches tokenizing sequences using WordPiece[19] and creating embedding of each word using the embedding layer.

3.3.1 Pre-training

In all of these approaches pre-training phase is done on IIT-CDIP document collection [10] where OCR extracts texts and corresponding word-level bounding boxes. The extracted information is then used for training on several pre-training tasks.

LayoutLM[22] and LayoutLMv2[21] are both pre-trained on the same task called **Masked Visual-Language Modeling**. This enables the model to learn better on the language side with the cross-modality clues. In this task, some tokens are masked, and we ask our model to recover these tokens; however, the spatial information remains unchanged, so our model knows exactly where the masked token is present on the page. LayoutLM uses also uses **Multi-label Document Classification** task; however, this approach did not prove any significant improvement in performance for our chosen downstream task.

In addition, LayoutLMv2[21] also uses two more tasks for pre-training. These are **Text-Image Alignment** and **Text-Image Matching**. **Text-Image Alignment**(TIA) helps the model to learn spatial location correspondence between images and coordinates of bounding boxes. In TIA, some text lines are randomly selected, and image regions are covered. On top of the Transformer-encoder is then added a linear layer, which predicts whether each token was covered. **Text-Image Matching** is used for the model to help learn the correspondence between document image and textual content. Text and final representations of the Transformer backbone are then fed forward into the classifier, which says if the text is contained in the image or not.

Doc-former[1] performs **Multi-Modal Masked Language Modeling** which is the same as **Masked Visual-Language Modeling** task in LayoutLM paper[22]. The next task is **Learn to reconstruct**(LTR) task which is similar to the previous one but now is

the goal to retrieve images from given final multi-modal features produced by DocFormer if they are put through a shallow decoder. The final task **Text Describes Image** is similar compared to **Multi-label Document Classification**, but this time we do not predict the label but the only binary answer if the given text represents the document image or not.

Chapter 4

Proposed solution using Graph Neural Networks for document understanding

The goal of this thesis is to create a solution for visually understanding document layouts and extracting specified information from them. While in some of the documents, it might be a simple task, there might be others that are challenging even for humans. Understanding document layout can help out to structure information extracted from text. Extracted text can be used then for knowledge graph creation which might help in searching for information. When the search is done, most of the information can be filtered out based on the layout of documents; for example, when searching for answers to a certain question on the web we can filter information from top to bottom based on the layout of the document, where we start with headings first. We remove from search all documents where the heading does not correlate with a given question. Then we have subheadings, paragraphs, text keys, and corresponding values. All this information can be structured the way we would like and then do various processes on this structured data. Layout from documents can be retrieved very simply from structured documents like HTML, but for scanned documents, the task is becoming very complex when several types of documents have to be dealt with.

There is a huge amount of data on the internet, but there could be hidden loads of potential in scanned documents if we could retrieve some structured data from them. This thesis aims to provide functionality just like that, where on a given set of documents we extract the information we are interested in. Having a set of scanned documents user can annotate a couple of examples and our model detects these annotated examples in the following documents potentially, speeding up the process of labeling the next dataset or being able to extract given information on demand based on previous specifications (annotation of couple examples) from thousands/millions next examples.

4.1 Task definition

To properly tackle the problem we have to define the tasks we are aiming to solve here. These tasks are semantic labeling of text entities in scanned documents and extraction of pre-defined information/categories from documents.

4.1.1 Semantic labeling of text entities in scanned documents

Semantic labeling of text entities in scanned documents is a challenging task that consists of several steps. However, we need to explain what text entity means and how it should be given the specified semantic label we assign to them.

Text Enitity represents a single block of text, which can be separated from the rest of the document. This can be done either semantically, based on its meaning from text, or visually based on its appearance in documents (bold text, italics, size of the text, etc.), or lastly, its position related to other text entities. Also surrounding entities might be influencing which category the given text entity belongs. These factors are hard to model, especially because modeling those relationships can be really challenging.

Semantic labeling as its name suggests, is assigning a given text entity into a certain category based on its features as we mentioned previously. The goal of this is that we want to assign a text entity the most specific label but still be within a generalized view where we can apply this label to any documents which we might encounter. This task is hard because there might always come along an entity that we won't be able to categorize in any of the specified labels and then end up assigning it the most general one which in the end might not help us to understand the document at all. We decided to approach this problem from bottom to up to be able to categorize each text entity we might have a touch with.

Starting from the bottom we have a basic *text* category. Here belongs everything, because it is the most generalized category. However, as we mentioned earlier, we want to be specific as possible. If we take a look closely at single-page form documents, as this thesis mainly works with documents like this, we might encounter several more specific categories of text entities from which these documents consist. For example, straight ahead we have *headers*. But not all headers are equal. Some of them are more important than others. Each header can summarize only certain parts of documents, others can summarize whole documents. In form-like documents, there is a certain text entity type that might not occur usually in other types of documents, but it's important to keep it separate from other entity types. These are keys and values e.g. Name and Surname might be the key for Nikolas Patrik. There is an even more specific entity that further specializes in these entities found in form documents and that is the key description. In forms, there usually might be questions about how to precisely fill in the blank space for the Key entity. For example, we have key *age* which is really ambiguous about what it should contain. We can use key descriptions which might describe what age you should put in e.g. age when you were vaccinated. Using the principle `<label>Description` we can come along couple more specific entities, which are usually present in documents. If we would describe for example what the header really is, it would be something like; it is a short summary of what the page contains. However, there might be also a long summary of what the page is about. The same goes for the sub-headers. This way we could describe a lot more categories from which documents consist. We will describe what categories of text entities we chose in our task in the next chapter 6 where we deal with the re-annotation of FUNSD[7] dataset.

4.1.2 Extraction of pre-defined information

The extraction of predefined information task is vaguely said that we want to retrieve as much specific data from documents as possible. By specific data, we can think of anything. In form documents, we might mean value for certain keys, in other documents, it might be headers, and so on. The goal of this section is to describe how we are going to specify which data is exactly what we want.

Let's assume we can assign to each entity certain embedding that defines its features. Having these embeddings can help us to compare the embeddings if two text entities are the same. This is a pretty straightforward approach. The question that arises is how to retrieve such embeddings for text entities. We could train simple feature extractors using e.g. Triplet Loss [17]. Nevertheless, this approach requires a huge dataset of annotated data where labels would be dependent from example to example which would be an extremely hard task to do to capture all the nuances. Mainly, we would like to lower the number of needed annotations and this is a step in the opposite direction. Also, these annotated examples might be not transferable to another set of documents. Assuming we have annotated documents where each name, address, etc. is labeled in documents with the same label for each. Our feature extractor for node embeddings will learn that all the names are same, all the addresses are the same and thus these embeddings will be closer to each other making them nearly impossible to differentiate. If we then received the documents where would two subsections be present where each of them has a name, and address inside, our embeddings could not differentiate these sections and we would end up selecting all the names even though we would be interested only in those from that one section.

Another approach could be comparing entities that conform to the same layout position and are semantically similar. Although the creation of the dataset might be simpler than in the previous case, still we encounter problems with the transferability of this solution which would not be interested so much in layout position but only in semantic similarity. Furthermore, there would be needed to define how to determine layout position and this could be also different task by task.

The approach we decided to go with is fairly simple and can be easily transferable on task by task basis with a fairly low amount of annotations needed. We will train the classifier for our specified classes on a task basis. This solution might seem really simple but there are some problems which are needed to be solved in the beginning.

Firstly, for training a classifier on a specific task it might be needed to have a bigger amount of annotations. We handle this by pre-training the model on semantic labeling task per text entity and finetuning the model for our specific data. By doing this we let our model have information of layout position as mentioned earlier encoded inside from the pre-training task and let it decide whether to use this information for our specific task. Because of pre-training, we can lower the amount of needed data to be annotated for a specific task.

Secondly, we use active learning techniques to aim mostly at examples on which our model struggles the most. Active learning helps us lower the number of needed examples to annotate due to the fact that we don't retrain our model on examples on which our model's predictions are good enough. How this is implemented is explained in the next sections.

4.2 Graph creation from images

The goal of this thesis is to provide a solution for scanned documents based on a graph neural network. To do this we need a way to convert images into graphs. This thesis uses undirected homogeneous graphs for image representation where each text entity is represented by its node.

The first step of converting images to graphs is detecting text entities. We detect text entities using Tesseract OCR[15]. We retrieve bounding boxes and transcription in `ALTO` format. ALTO format is XML format which represents page layout with several components which are hierarchically sorted. These elements are:

1. **Page** which represents whole page element,
2. **Composed Block** represents part of the page where a block of text is present and is not separated by any graphical elements or blank space,
3. **Text Block** represents text block, mostly they are paragraphs,
4. **TextLine** represents line of text,
5. **String** which are the words in the image with transcriptions,
6. and others like **Graphical Element** or **Print Space**

The above element tags each have attributes about the position which are used for bounding box generations. As each text entity should be sorted into its own category we use text lines for text entity detections and split them if there is a gap bigger than a certain constant between the words.

Although Tesseract OCR results are sometimes poor, mainly for handwritten text or small objects we decided to use it because on the datasets we were evaluating our solution we have been provided with transcriptions and bounding boxes. However, for production use, it would be required to use some OCR with better results like a Google Vision[1] (although this is paid solution for higher traffic, it comes also with a free tier if the traffic will be less than N request per month) or for higher traffic, it would be suggested to use pre-trained YOLO[11] detector as it was used in FUDGE[4] along with Tesseract OCR[15] for lines which certainly yields better results.

Following the retrieval of text entities, the next step is to create embeddings for each node in the graph. We use multi-modal embeddings from the image, position in the image, and text transcription of the entity.

Position embeddings are retrieved from the bounding box from Tesseract OCR[15]. These come in the format of $(y, x, width, height)$. These embeddings are then converted to format (x_1, y_1, x_2, y_2) as it is done in the FUNSD dataset [7]. These embeddings also work better with PIL[2] package manipulation with image afterward is simpler. Also, embeddings were converted to relative positions in percentage to retain better space locality for images with different sizes as it yielded better results during testing.

Next up, visual features were retrieved using pre-trained VGG19 model [14] using output from Max Pooling layer with dimensions `7x7x512`. The flattened output of this pooling layer returned a vector with shape $(25088,)$. Input for this pre-trained VGG19[14] model we used crops of images based on the bounding box of the entity, resize them to a certain size keeping the aspect ratio, and then cropping the middle square image so it matches the input size of the model.

Lastly, we retrieved text embeddings using the Sentence Transformer package which is based on *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*[12] article. Specifically, we used a pre-trained `all-mpnet-base-v2` model from [6]. This sentence-transformer model maps sentences & paragraphs to a 768-dimensional dense vector space and can be used for tasks like clustering or semantic search.

In the end, we concatenate all of these embeddings into a vector with size `25860`.

There is also one transformation, that is done with our graph. At the moment returned graph is just a set of nodes that are not connected. To connect these we use transformation

[1] https://cloud.google.com/vision

[2] https://pillow.readthedocs.io/en/stable/reference/Image.html

k-nearest neighbors with parameter $k = 6$. The framework we use Pytorch Geometric[3] provides us with this transformation as long as we provide the position for the given nodes. The provided positions are calculated as the middle of bounding boxes. Now we have the full graph representing the given image which can be used as input for our model described in the next section.

4.3 Graph Convolution Model

Finally, the model used in this thesis is a pretty simple model with just 3 layers of Graph Convolution Operator defined in „Semi-supervised Classification with Graph Convolutional Networks“ paper[9]. Fine-tuning of the model is done with the same model where the last Graph Convolution Operator is replaced with a new one with output channels changed to a specific number of classes in the active learning task.

[3] https://pytorch-geometric.readthedocs.io/en/latest/index.html

Chapter 5

Implementation

The solution from this thesis is implemented in Python 3[1] using the framework Pytorch Geoemtric[2] which is built on Pytorch framework[3]. The solution is also fully containerized using Docker[4] and released on Virtual Private Server where it is managed using Docker Compose Services. As mentioned solution also uses Tesseract OCR [15] which also as other frameworks will be described in the following sections.

5.1 Tools and Frameworks used

In this thesis, we used various tooling and frameworks to be able to annotate, train and evaluate our model. The most important frameworks we used were mainly Pytorch Geometric, LabelStudio, and Tesseract OCR [15] which will be described in this section.

5.1.1 Pytorch Geometric

PyG(PyTorch Geometric) is a library built upon PyTorch to easily write and train Graph Neural Networks (GNNs) for a wide range of applications related to structured data.

It consists of various methods for deep learning on graphs and other irregular structures, also known as geometric deep learning, from a variety of published papers. In addition, it consists of easy-to-use mini-batch loaders for operating on many small and single giant graphs, multi GPU-support, torch.compile support, DataPipe support, a large number of common benchmark datasets (based on simple interfaces to create your own), the GraphGym experiment manager, and helpful transforms, both for learning on arbitrary graphs as well as on 3D meshes or point clouds.

Pytorch Geometric is very similar to a Pytorch framework in many ways. Pytorch Geometric encapsulates graphs in `Data` object which is a really simple representation of the real graph. `Data` object has several attributes which are really important but you can store basically anything inside this object. This possibility of storing metadata in Data objects has allowed us to implement a tool for analyzing predictions from our model which will be mentioned in the following section. Besides the metadata `Data` object also consists of these several attributes:

[1] https://www.python.org/
[2] https://pytorch-geometric.readthedocs.io/en/latest/
[3] https://pytorch.org/
[4] https://www.docker.com/

- `Data.x` - this attribute stores input features of nodes in the graph.
- `Data.y` - this is very similar but only contains labels for each node
- `Data.edge_index` - this is a matrix that represents the connectivity of nodes. Basically, this is a representation of edges in the graph. The edges are directed. Due to this, for the undirected graph to be created, each edge must have its opposite direction.
- `Data.edge_attr` - this attribute is a matrix with features for each edge. But as we do not use it in this thesis we mentioned it just briefly.
- `Data.pos` - last attribute which is not really a graph attribute. This attribute can be used for various Transformations. In this thesis, it's used to create a graph from isolated nodes using k-nearest neighbor but there is also on similar transformation which uses this attribute. This transformation is called „Radius“ and can create a graph from isolated nodes based on a given radius and this attribute.

After creating this `Data` object from each image we can create `Dataset`. This `Dataset` will be used in `DataLoader` which creates batches of data as it would during training in Pytorch. From now, the training on our model is really similar as it would be in Pytorch framework.

5.1.2 Label Studio

Label Studio is a tool for labeling any kind of dataset which also comes with a free version. It's easy to install and the learning curve to get familiar with this tool is not that steep. When first logged in to the LabelStudio Project view shows up. When the project is selected the view changes for the Data Manager view, which contains all information about annotated tasks. When data is uploaded or imported to LabelStudio, it is converted to a Label Studio task.

Label Studio task is JSON-formatted text which describes the given data and annotations or predictions which were made on this data. Label Studio task contains `data` attribute which contains URL where annotated data is stored. Label Studio support storing the data on various storage from cloud providers, local storage where is running, uploading the data through the GUI interface and storing it locally, or storing it into PostgreSQL or Redis. As there wasn't much data to store we use the simplest option, the GUI upload. Simple enough was the Local Storage option too, but it was unsafe to use when Label-Studio is running on a public network which it was.

With uploaded data, labeling could start. The Label Studio is the perfect tool for labeling any kind of data but to be this utilizable for any project, the **Label Config** has to be set.

Label Config is a way of telling Label Studio how the data will be annotated. Basically, it is the XML-file that consist of several tags with different meaning which in the final result creates an Annotation View, which will people use in the given project. Tags can be divided into two categories which are:

- Object tags - these tags represent annotated data whether it is image, sound, text, etc.

- Control tags - usually data has to be labeled if we want to use it for training. Labels can have various forms and thus Label Studio supports several ways how to control labeling these objects.

Another very interesting feature Label Studio provides is the connection of machine learning backends. Machine learning backends are basically HTTP servers running REST API which handles requests for `/predict` endpoint where Label Studio tasks are sent. These machine learning backends can be used set up to one of the following:

- **Pre-labeling** by letting models predict labels and then have annotators perform further manual refinements.
- **Auto-labeling** by letting models create automatic annotations.
- **Online Learning** by simultaneously updating your model while new annotations are created, letting you retrain your model on-the-fly.
- **Active Learning** by selecting example tasks that the model is uncertain how to label for your annotators to label manually.

In this thesis, we use two of those four features. We use pre-labeling and active learning. Sadly, Label Studio Community Edition doesn't support Active Learning loops for labeling but, it can be used to do that with a couple of small modifications.

Pre-labeling Pre-labeling is done using Tesseract OCR[15] as it would be too difficult and too much time-consuming for annotators to do transcriptions and bounding boxes manually. In the begging, the machine learning backend was set up to do interactive pre-annotations which were done after the user has created a bounding box on the image. Nevertheless, this approach was really slow even though transcriptions were better as Tesseract performs better per line OCR than when it has to do its own detections. Thus, we decided to sacrifice the quality of transcriptions (although there isn't that much significant difference in quality) for speed.

Active Learning Active learning in Label Studio Community Edition is very tricky as Label Studio doesn't support active learning loops. In this thesis, we managed to do this by the following steps. Firstly we have to retrieve the notion of how many tasks were annotated. This is done by webhooks for certain events which can be set up even in Label Studio Community Edition. Webhook is the way Label Studio tells us certain action has happened. In this thesis, we were interested in an action called „ANNOTATION_CREATED". We created a simple flask[5] API which handles the request that way it incremented a counter for a certain project id in Redis[6]. Then we have running another background service that checks these counters periodically and when they go above a certain number they start training the model for the given project. When training is finished by using Label Studio SDK we create predictions for unlabeled tasks and assign each of them a *Prediction Score*. In the end, we sort the task in Data Manager View that way the tasks with lower *Prediction Scores* come first. This way we annotate the examples until we are happy with the predictions made by our trained active learning model.

[5]https://flask.palletsprojects.com/en/2.3.x/
[6]https://redis.io/

5.1.3 Tessseract OCR

Tesseract OCR[15] open-source OCR engine that extracts printed or written text from images. It was originally developed by Hewlett-Packard, and development was later taken over by Google. This is why it is now known as "Google Tesseract OCR".

As of now, Tesseract already supports language recognition for more than 100 languages "out of the box". The most recent version of Tesseract (4.0) has an AI integration through LSTM Neural Network to detect and recognize inputs with a variety of sizes better.

One of Tesseract's great strengths is that it is compatible with many programming languages and frameworks using wrappers such as Pytesseract, also known as Python-Tesseract.

Tesseract supports various output formats: plain text, hOCR (HTML), PDF, invisible-text-only PDF, TSV, and ALTO (the last one - since version 4.1.0) which we decided to use for output to create our pre-annotations.

5.2 Repository Structure

The repository of this thesis is structured the way that all components from certain docker-compose services should be together, however, there are some exceptions. The repository consists of only 4 directories which are:

- **ml_backend** directory - this directory contains all the needed logic for the machine learning backend for Label Studio. The directory consists of two scripts in which `_uwsgi` is responsible for creating Rest API which handles the requests to our machine learning backend and stores them into the Redis queue from which they are processed one by one. The second script defines class `TesseractOCR` which inherits from class `LabelStudioMLBase` and is responsible for creating predictions for a given task.
- **dataset** - this directory contains all of the data processing. Here we create `Dataset` object from our datasets either for the basic FUNSD dataset which needs to be converted into corresponding graphs or our re-annotated dataset. Furthermore, we also created class called `DatasetFactory` to simplify selecting various dataset and limit the number of if-statements in scripts that uses these datasets like `trianing.py, eval.py, analyze_predictions.py`.
- **models** - here we store our simple Graph Convolutional Model along with `ModelFactory` which is responsible for loading saved models or creating modified models for finetuning.
- **utils** - lastly we have the utils directory which contains all our repository settings or just the tools whose main purpose was to be reused

The repository contains several other files in the root directory but we will describe them better in the next sections.

5.3 Docker Compose services

For simpler dependency management, mostly all of the compounds of the solution are containerized using Docker. As most of these compounds needed to be run uninterrupted

for several hours we deployed our solution to the virtual private server and using docker-compose services these compounds were managed. Docker-compose gives us better control of what services are currently running and provides us with easier access to logs for these containers. Services that ran uninterrupted for several hours were:

Label Studio Label Studio service wasn't part of our repository but it was a service we managed using docker-compose. We cloned Label Studio's repository to our VPS (virtual private server) and ran docker-compose with our specified settings. This was the first service we have been running during our thesis.

ML Backend Machine learning backend was the next service that was running. As mentioned in previous sections, it was responsible for creating pre-labeling in our dataset to help us to do annotations.

Training Training service for running for several hours and we always had known what's happening inside. Docker Compose logs helped us to debug all the issues and kept running the training process for several hours. Nevertheless, training was not the only process that was running inside this service. Using `supervisord`[7] package we managed to run alongside the training also Tensorboard GUI[8] for tracking the metrics of our training.

Active Learning and Active Learning Webhook These two services were responsible for active learning. Active Learning was a never-ending `while-true` cycle which always checked in Redis for a number of annotations and based on that it started training and predictions creation. Active Learning Webhook on the other was just a simple Flask API that incremented a counter in Redis every time an annotation was created.

Other than these services we were running two more services: **Redis** which supported ML Backend for task storing and active learning for handling the counters for annotation and **NGINX**[9] which was routing requests to our ML Backend API.

5.4 Training and Evaluation

To train our model it was used class `Trainer`. The trainer was responsible for training our model based on provided config. The given config contained several parameters, but there were three worth mentioning. Config contained the model on which we wanted to do the training, dataset, and so-called `train_tag` which was used to differentiate multiple runs from each other.

Training start with loading datasets, and storing model if required. We then train the given dataset on a required model for a specific number of epochs and evaluate each epoch on the validation dataset. Based on the given loss on the validation dataset we store the best model. Also as docker is stopping containers by sending them `SIGTERM`, we implemented graceful shutdown where the best model until that moment is also saved as we do not store the model each time the new best model is discovered but only storing it only into a variable. Models are stored in a file periodically e.g. 1000 epochs. To track

[7] http://supervisord.org/
[8] https://www.tensorflow.org/tensorboard
[9] https://www.nginx.com/

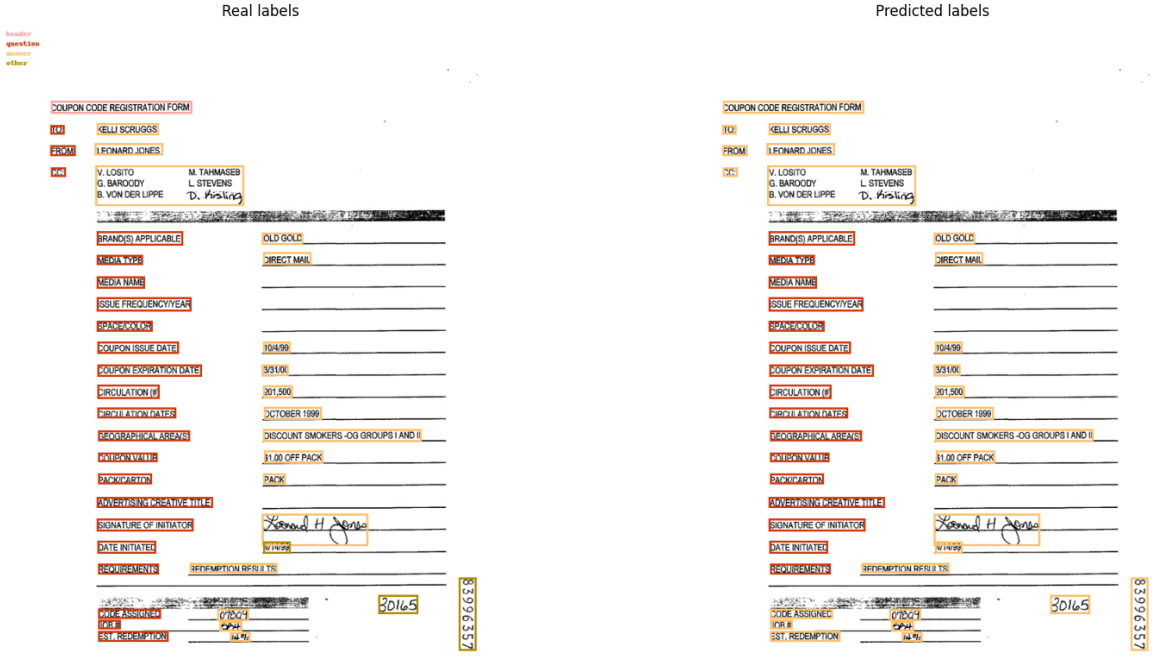

Figure 5.1: Example of output from analyze predictions tool.

how our training is progressing we also used Tensorboard which will be described in next subsection.

For our model evaluation, we used a simple script that loads the model using `ModelFactory` and uses the TorchMetrics[10] framework which has already implemented all the required metrics we wanted to use in this thesis.

Furthermore, to understand even more why is our model not performing as it should we implemented `analyze_predictions` tool which returns Real Labels vs Predicted Labels on the given picture from the test set. The output of this tool is shown in figure 5.1

5.4.1 Tensorboard

In machine learning, to improve something you often need to be able to measure it. TensorBoard is a tool for providing the measurements and visualizations needed during the machine learning workflow. It enables tracking experiment metrics like loss and accuracy, visualizing the model graph, projecting embeddings to a lower dimensional space, and much more.

Although Tensorboard is from the creators of Tensorflow[11] which is a competing framework from Google for machine learning, PyTorch decided to implement support for it and thus even though we use PyTorch in our project we were still able to use it.

For our use case, we were tracking loss on the training dataset and validation dataset to carefully examine progress on our training. A small illustration of how training process was monitored during training can be seen in figure 7.1

[10] https://torchmetrics.readthedocs.io/en/stable/
[11] https://www.tensorflow.org/

Figure 5.2: Example Loss vs Validation Loss in running instance of Tensorboard

Chapter 6

Re-annotation of FUNSD dataset

To ensure that our model contains information about the layout of text entities in the document, it was needed to provide a dataset that provides this information about the layout as precisely as needed. However, the original FUNSD dataset contains only the labels which more or less define only the relationship between key and value in the form, and two more that aren't really layout specific, even though one is called header, but it is just label which couldn't be linked to some other entity. NAF dataset which was also mentioned earlier doesn't properly define layout but only more granular define keys and corresponding values, for example pre-printed, hand-written, etc. Also, NAF dataset is too big, and re-annotating the whole dataset would require more time than doing so FUNSD dataset. The last reason to use the FUNSD dataset is the lower presence of handwritten text than in NAF, which as was mentioned earlier is hard to transcript by using Tesseract OCR [15] and as it will be noted in a while, Tesseract OCR was playing a big role in helping to annotate the dataset.

Due to this fact, the FUNSD dataset has been re-annotated based on suggestions from the previous chapter where semantic labeling was explained. Annotations were also created for a second task of specific data extraction. As these annotations, will not be used for the production model due to the fact active learning takes place based on user-specified data in a given task, we didn't have specific rules for annotations other than the semantic similarity between text entities with the same labels. This re-annotation process was done in a tool called LabelStudio[1] which was described in the previous chapter 5. Annotations have been defined based on strict rules and done with the help of Tesseract OCR[15]. The final re-annotated FUNSD[7] dataset was then used for training and evaluation for both tasks: semantic labeling and specific data extraction; which will be introduced results in the following chapter 7.

6.1 Annotators guidelines

To be able to guarantee that the model from this thesis will learn layout information properly, there have to be several rules on how annotations should be done. For the task of semantic labeling with decided to choose 8 labels:

- **Header**
- **SubHeader**

[1] https://labelstud.io/

- **TextKey**
- **TextValu**
- **TextKeyDescription**
- **PageDescription**
- **SubpageDescription**
- **Text**

and for the task of extraction of predefined data, we managed to specify 7 labels which were often present in provided form documents. These are:

- **SE_NAME**
- **SE_ADDRESS**
- **SE_DATE**
- **SE_AGE**
- **SE_BIRTHDAY**
- **SE_SIGNATURE**
- **SE_SEX**

When the annotator arrives to Label View, there already will be pre-labeled bounding boxes with transcriptions from Tesseract OCR[15]. The goal is to add labels to bounding boxes, adjust their size if needed and fix transcriptions. Also, there might be some weird bounding boxes from Tesseract that we want to remove. If some texts were not detected or we had adjusted another bounding box that was covering the part of the image where is text and it is not covering it now, we create the new bounding box manually add assign it to label and transcription respectively.

Now which bounding box should be assigned which label is a straight forward process but to be done consistently we need to define a clear specification of what each label should represent.

Header Header label is assigned to bounding boxes which should shortly summarize the whole scanned document. Usually, it's bold text with higher font than other elements that is situated on top of the page and which is one of the reasons why we included visual features and positional features as input to our model.

Subheader Subheader similar to the previous label with its features, usually font is a little smaller and positionally can be situated anywhere on the page. The only difference between Header and Subheader is that Header should summarize the whole page, but on the other hand, Subheader summarizes only part of the page. Subheaders are not further distinguished because in form documents, there can be a lot of subheader hierarchies and we don't want to create labels for each.

TextKey TextKey label is used for the key in the forms, everything that specifies what should be filled in the form is TextKey. However, one small detail is that it has to be *Key*, which means it should be a short value or description. For longer descriptions of the keys we defined new label.

TextKeyDescription TextKeyDescription is a label that has a certain relationship with TextKey. TextKeyDescription might be anything that further describes what should be written to a given TextKey. For example, there might be TextKey for the Personal ID but that could be anything here in the Czech Republic. It could be a birth number or just a number from the ID person has. TextKeyDescription can explain what it means more closely.

TextValue TextValue label is everything that is written on the other side of TextKey. It is the value that semantically corresponds with TextKey and positionally is also somewhere near.

PageDescription PageDescription closely correlates to the Header label. As its name suggests it describes the page as a whole, and as the header is known to be short summary of the page, page description is something that is much more specific about what the page should contain.

SubpageDescription SubpageDescription is a label that relates to the Subheader label as PageDescription relates Header label. It should be also a longer description of a certain part of the page is about.

Text Text is the default label. It is used when no condition which was described above applies to the TextEntity and we have no other choice than to choose this label.

6.2 Re-annotated FUNSD

Re-annotated FUNSD contains 76 fully labeled images with 51 of them containing labels for specific entities which were used to simulate active learning and then used for evaluation of our solution. These annotated examples have been split into three sets based on given percentages: (train(67,5%), validate(7,5%), and test(15%)). An example of a fully labeled image in Label Studio is shown in the figure 6.1

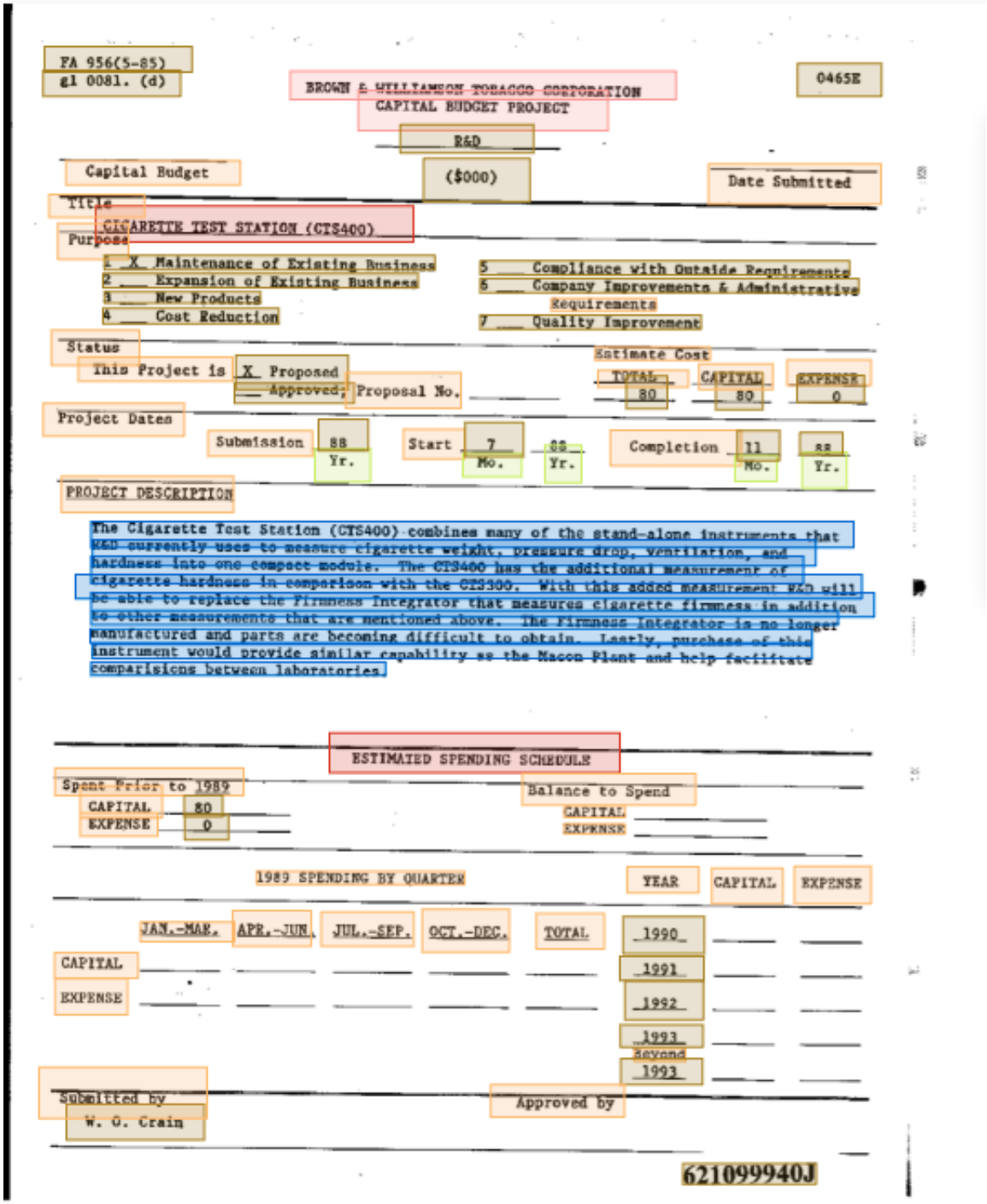

Figure 6.1: Overview of how the labeled image looks like in Label Studio for our reannotated FUNSD dataset

Chapter 7

Experiments

In this chapter we describe how our solution was trained and after that evaluated on a FUNSD dataset. Then we compared it to other approaches mentioned previously in this thesis which also were evaluated on the FUNSD dataset. After that, we evaluated the re-annotated FUNSD dataset we have created and visualized in our `analyze_predictions` tool. Lastly, we explain how we evaluated the active learning scenario on our reannotated dataset and how we evaluated it afterward in a real scenario where we wanted to retrieve specified data from a given dataset.

7.1 FUNSD

In the beginning, we trained our model on FUNSD dataset in two-ways. As we are using bounding box corners for each node as part of the input features we decided that using relative and absolute might make a difference in results. We assumed that the relative positions might convey more reliable information about where the given element is located even when comparing two nodes (bounding boxes) from two different images which have different sizes. We trained our model for 7 thousand epochs with absolute bounding boxes and for 3 thousand epochs with relative as can be seen in figure 7.1

Figure 7.1: Training(top) vs Validation(bottom) loss for absolute bounding box features and relative bounding box features. Absolute bounding boxes: blue(training), red(validation); relative bounding boxes: pink(training) green(validation)

Method	GT OCR info	#params	Recall	Precision
LayoutLM_{BASE}	word boxes + transcription	113M	75,97	81,55
LayoutLM_{LARGE}	word boxes + transcription	343M	75,96	82,19
LayoutLMv2_{BASE}	word boxes + transcription	200M	80,29	85,39
LayoutLMv2_{LARGE}	word boxes + transcription	426M	80.24	85,19
Word-FUDGE	word boxes	17M	69,37	75,30
FUDGE no GCN	none	12M	63,64	66,67
FUDGE	none	12M	64.90	68,23
GCN absolute bbox	word boxes + transcription	54M	47,86	47,50
GCN relative bbox	word boxes + transcription	54M	50,04	49,00

Table 7.1: Text entity detection/Semantic labeling on the FUNSD dataset

Model	Dataset	Recall	Precision
Our GCN model	Re-annotated FUNSD	48,27	48,51

Table 7.2: Text entity detection/Semantic labeling on re-annotated FUNSD dataset

As we can see from the graphs, loss during training of relative bounding boxes went much steeper down than for absolute bounding boxes. Also, we can see that loss both on training and validation stabilized around the same value around the 1000th epoch for both absolute and relative bounding boxes and it didn't improve even after another thousand epochs.

We evaluated our models for the accuracy of semantic labeling of nodes, as well as for Precision and Recall metrics as it was done in FUDGE[4]. The results we retrieved can be seen in table 7.1

As we can see our model did not perform very well in comparison to other solutions such as LayoutLM or FUDGE. This can be due to the fact we did not do any huge pretraining like LayoutLM did or our model being too simple which could not compare to bigger and more complex models like FUDGE. Nevertheless, as we can see using relative bounding boxes performed better also on the test set of the FUNSD dataset so we decided to continue with these features in the next testing.

7.2 Re-annotated FUNSD

In the next steps, we evaluated our model on the FUNSD dataset that we have re-annotated. As mentioned earlier we use relative bounding boxes while creating graphs from images. We trained our model for thirty-thousand epochs when we noticed increasing loss on the validation set as can be seen in fig 7.2. As we can see from the graph, the loss has been decreasing the whole time which led us to think that model is still training well. However, after evaluation, our model seems to be struggling with entities that had a small representation in the dataset. Evaluation of our model on given re-annotated FUNSD dataset can be seen in the table 7.2.

Also, we have calculated accuracy for each class in our dataset where we have learned that our model struggled to learn entities that are not often presented in the dataset. Exact values how our model performed on our dataset can be found in table 7.3

Accuracy on Class	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Our GCN model	23,08	0.00	33,64	69,53	0.00	0.00	0.00	7,14

Table 7.3: Accuracy of our model for each class: 0. Header; 1. Subheader; 2. TextKey; 3. TextValue; 4. TextKeyDescription; 5. PageDescription 6. SubpageDescription 7. Text

Accuracy on Class	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Finetuned GCN 5	0.00	0.00	0.00	0.00	0.00	0.00	3.7	94.33
Finetuned GCN 10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1

Table 7.4: Accuracy on the model which was trained with 5 examples and 10 examples; 0. SE_NAME, 1. SE_SIGNATURE, 2. SE_AGE, 3. SE_ADDRESS, 4. SE_BIRTHDAY, 5. SE_SEX, 6. SE_DATE, 7. unspecified

As it can be seen from table 7.3, our model learned to recognize `TextValues` and `TextKeys` with slightly better performance than others. Other labels it did not recognize at all.

Figure 7.2: Training loss vs Validation loss on re-annotated FUNSD dataset

7.3 Active Learning

As we describe earlier active learning is used to retrieve specific data from various documents by learning our model to recognize this data. We do so by using the pre-trained model on a re-annotated FUNSD dataset so we do not need to annotate a large amount of data, but a couple of examples should be enough. We wanted to evaluate our model on our dataset, however, this would not produce the same results as active learning would. Thus we decided to slice the dataset to the amount that we implemented in active learning to start the training. We did two sets of slicing where the first should represent the first annotation of examples. The second slicing should represent retraining our model after the first training has been done. Both of these training yielded very strange results as they learn to „recognize“ only the unspecified label. These results can be seen in the next table 7.4

As it is shown in table 7.4, our model learned to predict only labels for unspecified elements. This might be due to the reason that we have most of our annotated full of these unspecified elements as it was often the case that scanned documents didn't have specified elements that we wanted to extract. Thus slicing our dataset resulted in a very small number of samples for specified elements and it's understandable that our model did not learn anything.

Thus we decided to evaluate our model in a real scenario of active learning. For this purpose, we downloaded Brazilian Identity Document Dataset[16] which should contain more similar specific entities in each document. We specified two entities for our scenario: `Name` and `Date of Expiry`. Results of our experiment are shown in the enclosed video with this thesis mentioned in A. To summarize this video, our model failed to predict labels correctly even during real-time testing and still predicted unspecified labels for each text entity.

To conclude all of our experiments, we have been able to learn our model some nuances on layout datasets like FUNSD or our re-annotated dataset. However, active learning which could be potentially greatly usable in the real world in any type of document processing scenario did not succeed as expected.

In the future, there might be potential improvements in this process with a couple of adjustments such as more complex models or better features even on the edges of the graph. Also, the main aim to improve the results of this thesis will be to improve accuracy on benchmarking datasets as this knowledge is transferable to our data extraction task. Another objective that is mainly related to active learning is that a better OCR engine could also contribute to improving results as these OCR predictions are used to build graphs used for our model input.

Chapter 8

Conclusion

In conclusion, in this thesis, we have explained how graph neural networks work and how they compare to methods in basic machine learning. Furthermore, we described several solutions which are dealing with document understanding. In addition, we introduced datasets that are used in these solutions and explained why we decided to work with only one of them.

In previous chapters, we also proposed a solution that should handle two specific tasks. The first one was the semantic labeling of text entities and the second was the extraction of specified data. Next, we described how the processing of the annotation of a new dataset went. And finally, we evaluated our solution on a basic FUNSD dataset on which other solutions were evaluated and compared with others. Sadly, the results of our solution were not good as expected. However, we continued using this model for the evaluation of our newly created dataset. The training of our model looked much better on our dataset because losses on the train set and validation set were much better compared to the previous dataset training. Ironically, the results still did not improve and they were similar to results from the previous evaluation on the FUNSD dataset. As this evaluation was not the main purpose this thesis was aiming for. We tried finetuning our model again on our dataset but only for limited examples as we wanted to mimic behavior during active learning. We explained previously in the thesis how active learning should be used using our model and how our model is fine-tuned during the active learning process. Assuming our base model wasn't performing very well on the basic nor re-annotated datasets, we did not expect radical improvement in the active learning scenario. As a result, our model could not compete with current state-of-art approaches and thus there's a need for more work to improve its performance.

The future plans for this work are aiming to use a more complex model rather than using a simple 3-layer Graph Convolutional Network. An idea of using Graph Attention Networks for this solution might be a good following step for improving our model, however, this would require creating edge features for our graph. This could be achieved by adding distance as a feature to our edges. Another idea that might help improve the performance of our model might be using embeddings from LayoutLM as they were used for input for a simple linear layer network, thus we could improve their performance thanks to the usage of the Graph Neural Network.

Bibliography

[1] Appalaraju, S., Jasani, B., Kota, B. U., Xie, Y. and Manmatha, R. DocFormer: End-to-End Transformer for Document Understanding. *CoRR*. 2021, abs/2106.11539. Available at: https://arxiv.org/abs/2106.11539.

[2] Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V. F. et al. Relational inductive biases, deep learning, and graph networks. *CoRR*. 2018, abs/1806.01261. Available at: http://arxiv.org/abs/1806.01261.

[3] Davis, B. L., Morse, B. S., Cohen, S., Price, B. L. and Tensmeyer, C. Deep Visual Template-Free Form Parsing. *CoRR*. 2019, abs/1909.02576. Available at: http://arxiv.org/abs/1909.02576.

[4] Davis, B. L., Morse, B. S., Price, B. L., Tensmeyer, C. and Wigington, C. Visual FUDGE: Form Understanding via Dynamic Graph Editing. *CoRR*. 2021, abs/2105.08194. Available at: https://arxiv.org/abs/2105.08194.

[5] Devlin, J., Chang, M., Lee, K. and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR*. 2018, abs/1810.04805. Available at: http://arxiv.org/abs/1810.04805.

[6] Dunn, M., Sagun, L., Higgins, M., Güney, V. U., Cirik, V. et al. SearchQA: A New Q&A Dataset Augmented with Context from a Search Engine. *CoRR*. 2017, abs/1704.05179. Available at: http://arxiv.org/abs/1704.05179.

[7] Jaume, G., Ekenel, H. K. and Thiran, J. FUNSD: A Dataset for Form Understanding in Noisy Scanned Documents. *CoRR*. 2019, abs/1905.13538. Available at: http://arxiv.org/abs/1905.13538.

[8] Jure Leskovec. *Standford CS224W: Graph Neural Networks* [URL: http://web.stanford.edu/class/cs224w/slides/06-GNN1.pdf]. October 2021.

[9] Kipf, T. N. and Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR*. 2016, abs/1609.02907. Available at: http://arxiv.org/abs/1609.02907.

[10] Lewis, D. D., Agam, G., Argamon, S. E., Frieder, O., Grossman, D. A. et al. Building a test collection for complex document information processing. *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. 2006.

[11] Redmon, J. and Farhadi, A. YOLOv3: An Incremental Improvement. *CoRR*. 2018, abs/1804.02767. Available at: http://arxiv.org/abs/1804.02767.

[12] Reimers, N. and Gurevych, I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, November 2019. Available at: https://arxiv.org/abs/1908.10084.

[13] Ren, S., He, K., Girshick, R. B. and Sun, J. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *CoRR*. 2015, abs/1506.01497. Available at: http://arxiv.org/abs/1506.01497.

[14] Simonyan, K. and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*. 2014, abs/1409.1556. Available at: http://arxiv.org/abs/1409.1556.

[15] Smith, R. An Overview of the Tesseract OCR Engine. In: *ICDAR '07: Proceedings of the Ninth International Conference on Document Analysis and Recognition*. Washington, DC, USA: IEEE Computer Society, 2007, p. 629–633. ISBN 0-7695-2822-8. Available at: http://www.google.de/research/pubs/archive/33418.pdf.

[16] Soares Álysson, Neves Junior, R. das and Bezerra, B. BID Dataset: a challenge dataset for document processing tasks. In: *Anais Estendidos do XXXIII Conference on Graphics, Patterns and Images*. Porto Alegre, RS, Brasil: SBC, 2020, p. 143–146. DOI: 10.5753/sibgrapi.est.2020.12997. ISSN 0000-0000. Available at: https://sol.sbc.org.br/index.php/sibgrapi_estendido/article/view/12997.

[17] Vassileios Balntas, D. P. and Mikolajczyk, K. Learning local feature descriptors with triplets and shallow convolutional neural networks. In: Richard C. Wilson, E. R. H. and Smith, W. A. P., ed. *Proceedings of the British Machine Vision Conference (BMVC)*. BMVA Press, September 2016, p. 119.1–119.11. DOI: 10.5244/C.30.119. ISBN 1-901725-59-6. Available at: https://dx.doi.org/10.5244/C.30.119.

[18] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L. et al. Attention is All you Need. In: Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R. et al., ed. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017, vol. 30. Available at: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[19] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M. et al. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR*. 2016, abs/1609.08144. Available at: http://arxiv.org/abs/1609.08144.

[20] Xie, S., Girshick, R., Dollár, P., Tu, Z. and He, K. Aggregated Residual Transformations for Deep Neural Networks. *ArXiv preprint arXiv:1611.05431*. 2016.

[21] Xu, Y., Xu, Y., Lv, T., Cui, L., Wei, F. et al. LayoutLMv2: Multi-modal Pre-training for Visually-Rich Document Understanding. *CoRR*. 2020, abs/2012.14740. Available at: https://arxiv.org/abs/2012.14740.

[22] Xu, Y., Li, M., Cui, L., Huang, S., Wei, F. et al. LayoutLM: Pre-training of Text and Layout for Document Image Understanding. *CoRR*. 2019, abs/1912.13318. Available at: http://arxiv.org/abs/1912.13318.

Appendix A

The contents of enclosed DVD

The DVD enclosed with this thesis contains:

- `repository/` - directory which contains all of the code from this thesis
- `repository/README.md` - manual for installation and setup of our solution
- `data/` - contains re-annotated FUNSD dataset
- `tech_report/` contains source code of this thesis
- `video.mov` - short video presenting result of this thesis