UPPSALA
UNIVERSITET

UPTEC STS 21025

Examensarbete 30 hp

Maj 2021

Multimodal Convolutional Graph Neural Networks for Information Extraction from Visually Rich Documents

Kevin Ajamlou

Max Sonebäck

Civilingenjör i System i Teknik och Samhälle

UPPSALA
UNIVERSITET

Multimodal Convolutional Graph Neural Networks for Information Extraction from Visually Rich Documents

Kevin Ajamlou
Max Sonebäck

Abstract

Monotonous and repetitive tasks consume a lot of time and resources in businesses today and the incentive to fully or partially automate said tasks, in order to relieve office workers and increase productivity in the industry, is therefore high. One such task is to process and extract information from *Visually Rich Documents* (VRD:s), e.g., documents where the visual attributes contain important information about the contents of the document. A lot of recent studies have focused on information extraction from invoices, where graph based convolutional nerual networks have shown a lot of promise for extracting relevant entities. By modelling the invoice as a graph, the text of the invoice can be modelled as nodes and the topological relationship between nodes, i.e., the visual representation of the document, can be preserved by connecting the nodes through edges. The idea is then to propagate the features of neighboring nodes to each other in order to find meaningful patterns for distinct entities in the document, based on both the features of the node itself as well as the features of its neighbors.

This master thesis aims to investigate, analyze and compare the performances of state-of-the-art multimodal graph based convolutional neural networks, as well as evaluate how well the models generalize across unseen invoice templates. Three models, with two different model architecture designs, have been trained with either underlying ChebNet or GCN convolutional layers. Two of these models have been re-trained, and compared to their predecessors, using the over-smoothing combatting technique DropEdge. All models have been tested on two datasets - one containing both seen and unseen templates and a subset of the previous dataset, containing only invoices with unseen templates.

The results show that multimodal graph based convolutional neural networks are a viable option for information extraction from invoices and that the models built in this thesis show great potential to generalize across unseen invoice templates. Moreover, due to an inherent sparse nature of graphs modeled from invoices, DropEdge does not yield an overall better performance for the models.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala

Handledare: Emil Fleron (Violet AI Labs) Ämnesgranskare: Anders Brun

Examinator: Elísabet Andrésdóttir

Populärvetenskaplig Sammanfattning

Enligt en studie från 2017 [1] lägger kontorsarbetare världen över i snitt **552 timmar om året** på administrativa eller repetitiva arbetsuppgifter, samtidigt som **75 %** av de tillfrågade anger att de gärna hade överlåtit allt eller vissa delar av detta arbete till automatiserade processer. En sådan arbetsuppgift som förekommer inom de flesta företag, och som innehåller både repetitiva och strukturerade delmoment, är informationsextrahering från *Visuellt Berikade Dokument*, eller *Visually Rich Documents*, VRD:s. Ett VRD är ett dokument där layouten på dokumentet innehåller stora delar av dokumentets information och där stora delar av förståelsen för innehållet skulle begränsas om dokumentet komprimerades till en löpande text och där de visuella attributen inte togs i beaktande. Ett exempel på detta är fakturor, vilket också är den typ av VRD som kommer utgöra basen för detta examensarbete. Givet ett textfält, eller en *entitet*, med textinnehållet "1049" är det svårt att veta om detta är ett totalbelopp, ett momsbelopp eller ett fakturanummer om vi inte tar i beaktande vad som också står i textrutorna över, under såväl som till höger och vänster om entiteten. Det blir alltså viktigt att analysera den visuella kontexten såväl som det faktiska textinnehållet av ett givet textfält för att förstå dess betydelse.

Violet AI Lab är ett företag som fokuserar på att leverera AI-drivna lösningar till andra verksamheter. En sådan lösning de utvecklat är en informationsextraheringsalgoritm som extraherar olika entiteter, till exempel totalbelopp, OCR-nummer och förfallodatum, från fakturor. Deras nuvarande algoritm använder sig av en samling traditionella maskininlärningsmodeller tillsammans med heuristiska regler för att sammankoppla nyckel-värdepar och på så vis extrahera viktig information från dokumenten. Idén för detta examensarbete uppstod när ny forskning pekat på att *multimodala* tillvägagångssätt visat stor potential inom detta fält.

En maskininlärningsmodell som tar flera *modaliteter*, det vill säga informationskanaler av olika karaktär, till exempel ljud, bild eller text, i beaktande sägs vara *multimodal*. Genom att använda ett multimodalt tillvägagångssätt går det alltså att bibehålla både den visuella såväl som den textuella kontexten av ett VRD för att processera och tolka dokument. Vidare är *neurala nätverk* en familj av maskininlärningsmodeller som bygger på att sammankoppla nätverk av artificiella neuroner och träna dessa till att hitta betydelsefulla mönster i den underliggande datan.

Genom att modellera fakturorna som *grafer*, där varje ord representeras av en *nod* och där noderna har *kanter* till sina närmaste grannar kan både textinnehållet och den visuella kontexten användas för att träna dessa modeller. Attributen för en nod *blandas samman*, eller *konvolveras* med attributen från närliggande noder för att på så vis skapa en ny betydelsefull representation av noden baserat på dess lokala kontext. Denna nya representation kan sedan användas av modellerna för att prediktera om den givna noden tillhör en av de i förväg bestämda klasserna som ska extraheras, till exempel *totalbelopp* eller *fakturanummer*. För att kunna träna ett neuralt nätverk på graf-data krävs en speciell typ av nätverksdesign, något som kallas för *Graph Neural Networks* (GNN:s). En viss typ av GNN, vars huvudfokus är att just konvolvera attributen mellan noder, kallas för *Convolutional* Graph Neural Networks, eller *ConvGNN:s*. Inom ConvGNN-familjen har just *spektrala* ConvGNN:s visat mycket stor potential inom informationsextrahering från VRD:s, därför kommer modellerna i detta examensarbete att baseras på två olika spektrala ConvGNN-lager, nämligen *ChebNet* och *Graph Convolution Network* (*GCN*). I linje med ny forskning kommer vi i detta examensarbete bygga tre olika multimodala neurala nätverk baserade på dessa två konvolveringslager.

ConvGNN:s är fortfarande ett relativt ungt forskningsområde och det finns således vissa problem som forskningsvärlden fortfarande arbetar hårt med att lösa. Ett sådant problem är att de nya nod-representationerna som skapas vid en konvolvering börjar likna varandra mer och mer för varje konvolvering, även för noder som tillhör olika klasser. Detta fenomen, som kallas *over-smoothing*, gör det svårt att bygga djupa ConvGNN-arkitekturer och således också svårare att hitta komplexa samband mellan noder som ligger långt från varandra i grafen. En teknik för att motverka over-smoothing är *DropEdge*, som bygger på att slumpmässigt, med en given sannolikhet, ta bort vissa av de kanter som finns i grafen under träning av modellen. För att undersöka vilken inverkan DropEdge har på modellerna i detta examensarbete kommer två av modellerna att tränas om med DropEdge implementerat och där

olika sannolikheter kommer att testas.

Något som tidigare varit populärt för att extrahera information från VRD:s har varit *mall-baserade* modeller. Dessa bygger på att strukturen för dokumentet hårdkodas in i extraktionsalgoritmen, där text på en given position i dokumentet svarar mot en given entitet som ska extraheras. En stor nackdel med detta är dock att denna typ av design skalar väldigt dåligt inom verkliga applikationsområden, då ett dokument som bygger på en ny mall potentiellt skulle kunna störa extraktionsprocessen. En stor fördel med att använda sig av ConvGNN:s istället är att forskning pekar på att dessa modeller är mall-agnostiska, det vill säga att ConvGNN-baserade modeller även fungerar på dokument som bygger på mallar som inte förekommit under träning. För att undersöka detta kommer modellerna i detta examensarbete att testas på en delmängd av fakturor som är baserade på mallar som antas inte förekomma under träning.

Tidigare forskning inom informationsextrahering från fakturor med hjälp av ConvGNN:s har använt sig av förhållandevis små datamängder, där storleken på datamängderna har legat inom spannet av 3000 - 4000 fakturor. Till detta projekt har vi fått tillgång till 83 672 unika fakturor från 18 272 olika utfärdare. Uppgiften har varit att extrahera 13 olika klasser som förekommit i varierande grad på fakturorna i datamängden. En 14:e klass, *undefined*, har skapats för att kategorisera alla ord som inte tillhör någon av de förstnämnda 13 klasserna. De olika modellerna har sedan testats på en testmängd bestående av 4136 fakturor, samt en delmängd av denna bestående av 393 fakturor med mallar som inte förekommer i träningsmängden.

Våra resultat visar att en multimodal ConvGNN-baserad lösning kan vara ett gångbart alternativ för informationsextrahering från fakturor. Den bäst presterande modellen, en ChebNet-baserad modell bestående av 4 konvolveringslager med ett avslutande linjärt lager, uppvisade ett makrogenomsnitt F_1 på 0.7119, precision på 0.8259 och recall på 0.6255. Dessa siffror ökade dock med 15.5 %, 1.8 % samt 28.7 % när de 4 minst förekommande klasserna inte räknades med i resultaten. Samma modell presterade även bäst på delmängden av fakturor med osedda mallar och hade där ett makrogenomsnitt F_1 på 0.6015, precision på 0.6816 och recall på 0.5382. Vidare såg vi en genomgående prestationssänkning för modellerna efter applicerandet av DropEdge i takt med att dess effekt ökade i styrka. Vissa undantag var för en GCN-baserade modellerna som fick en svag ökning i F_1, precision och recall för probabiliteterna $p = 0.10$ och $p = 0.20$.

Acknowledgements

This master thesis project has been carried out at Uppsala University in collaboration with Violet AI Lab, a Stockholm based machine learning company specialized in artificial intelligence and machine learning applications. We would first like to extend our thanks and utmost appreciation to Emil Fleron and Mikael Nelsson, our supervisors at Violet AI Lab, for providing us with excellent help, as well as thank all the company's employees who helped us throughout the process, in particular Anna Rydin and Johannes Koch. We would also like to thank our subject reviewer, Anders Brun, for aiding us when aid was needed. Lastly, we would like to thank Péter Nagy, CEO of Violet AI Lab, for giving us the opportunity to write our master thesis in collaboration with the company. Without aforementioned's help and contributions, this project would not have been feasible.

Contents

1 Introduction

1.1 Background

Visually Rich Documents (VRD:s) such as banking documents, receipts and invoices, play an integral role in our day to day lives, especially in the world of business. The defining characteristics of a VRD is the importance of visual cues and layout information embedded into the document, where any removal of such visual features would reduce the information quality of the document as a whole. VRD processing, such as extracting due dates from invoices or other types of information extractions, is often a monotonous task historically achieved through manual labor. Automatic assistance of said tasks could potentially lead to a great reduction in both time consumed as well as money spent for companies which handle and process VRD:s. However, automatic handling of VRD:s becomes a complicated subject matter because of the entanglement of information in the visual cues of the document and any representation of a VRD that only leverages textual information results in loss of information. Furthermore, any template-based model, i.e., where the schema of the document is hard-coded into the extraction process, would scale poorly as there is no standardization for where certain values must appear and any previously unseen template, e.g., an invoice from a new vendor, would potentially break the model [2].

In recent years, *multimodal* approaches have become a hot research topic [2], [3], [4]. A multimodal model is simply a model which can process data from multiple modalities, such as audio, video and linguistic data. For VRD:s, it is important to take both the layout as well as the textual information into consideration when extracting information. In the field of information extraction from invoices, a tactic which has shown promise in recent research [2], [3], [5] is to preserve the layout information as well as the textual content of the invoice by modelling the invoice as a network graph. The graph is in turn fed to a *Convolutional Graph Neural Network* (ConvGNN) to extract the relevant information. There has been a surge of different ConvGNN designs as of late, although the goal is shared among these models: to generate a representation of each node through aggregation of a nodes own features with the features of its neighbors, thus informing each node of its local context [6]. In the context of information extraction from invoices, these newly generated node features can then be used in order to predict the class of a node, e.g., whether a node contains the total amount, due date, invoice number and so on.

A particular type of ConvGNN:s that has shown a lot of potential in the field of information extraction from invoices are the *spectral based* ConvGNN:s [3]. Spectral based ConvGNN:s are based heavily on spectral graph theory and aggregates node features by treating the features as signals and convolving them in the spectral domain. However, transforming features to the spectral domain is a costly operation which involves eigen-decomposition of the graph Laplacian, which is why ConvGNN:s such as *ChebNet* [7] and *Graph Convolutional Networks* (GCN) [8], the two graph convolution layer designs that will be inspected in this thesis, resorts to approximating the spectral transformation and convolution through Chebyshev polynomials instead.

However, the field of spectral graph convolutions is still young and there are some problems with the current strategies that the scientific community is struggling to solve. One such problem is the problem of over-smoothing, where the feature representations of nodes from different classes become indistinguishable from each other as the network grows deeper and more complex [9]. Rong et al. [10] propose the idea of randomly removing some of the edges from a given graph to relieve some of the over-smoothing, a technique that they fittingly call *DropEdge*.

This thesis aims to evaluate and compare the predictive power of ChebNet and GCN for information extraction of invoices by labeling nodes in graph representations of invoices as one of 14 different classes. We also want to see if the DropEdge technique has any effect on the final results. The training is done on a labeled invoice dataset consisting of 83 672 unique Swedish invoices from 18 272 different vendors. The size of this dataset is unprecedented, as previous studies in the field of convolutional graph neural networks have been limited to at most a couple of thousand invoices [3], [5]. Each entity (e.g., the due date, the VAT amount et cetera) of the invoices has been labeled by a mixture of automated processes and human supervision as well as intervention, where needed. This data was sourced by a third party before the inception of the project idea, thus making the actual sourcing not a part of this thesis.

1.2 Purpose

The purpose of this thesis is to evaluate and analyze the performances of state-of-the-art graph based neural networks on the task of extracting information from invoices, as well as evaluate how well the networks generalize for unseen invoice templates. Three models are built using either underlying ChebNet or GCN convolution layers. The models are trained and compared with respect to their *macro average recall, precision* and F_1*-score*. The two deepest models are also retrained using DropEdge with different drop probabilities, and will be compared to their previous results.

1.3 Organization of the Work

This master thesis has been done in collaboration with the Stockholm based machine-learning applications company Violet AI Labs. The idea for writing a thesis about multimodal information extraction from invoices came from this project's supervisors, who wanted to investigate how the company's current machine-learning models compares to state-of-the-art approaches on said task.

During the course of the project, weekly follow up meetings have been conducted with the project supervisors and machine learning engineers at the company, where focus have been on discussing the current status of the project as well as further ideas and improvements to implement. Several presentations for the company's employees have been carried out as well, both for machine-learning engineers and data scientists, as well as for laymen and employees not working directly with machine-learning technologies. The presentations have consisted of describing the current statuses of the project, the theories on which our thesis are based on as well as how we, the authors, have chosen to carry it out.

2 Related Work

The following section describes related research done in the field of information extraction from VRD:s, with three different approaches being presented.

2.1 Attend, Copy, Parse: End-to-end Information Extraction from Documents

Palm [4] proposes in his research paper "Attend, Copy, Parse: End-to-end information extraction from documents" an end-to-end approach for information extraction from VRD:s (end-to-end meaning string outputs extracted from PDF or document image input). Palm introduces an "Attend, Copy, Parse" architecture, utilizing a deep neural network model that can be trained directly on end-to-end data. Since end-to-end data is easy to come by, the author argues that using his proposed system eliminates the need for the expensive work of word-level labeling, which state-of-the-art word classification methods for information extraction currently rely on. Palm evaluates his proposed architecture on a large dataset consisting of invoices and outperforms production system based state-of-the-art word classification. However, such an approach assumes text to only have sequential relations in the horizontal plane and does not take the 2-dimensional nature of invoices into account.

2.2 Graph Convolution for Multimodal Information Extraction from Visually Rich Documents

Unlike [4], Liu et al. [5] propose in their research paper "Graph Convolution for Multimodal Information Extraction from Visually Rich Documents" a different approach for information extraction from VRD:s (receipts), namely a multimodal approach using Convolutional Graph Neural Networks. In their model, they combine textual and visual information present in these VRD:s where graph embeddings are trained to summarize the context of text segments in the document. According to the authors, their system outperforms classic information extraction models by significant margins. The authors use a dataset consisting of government regulated invoices, meaning that they all follow the same template.

2.3 An Invoice Reading System Using a Graph Convolutional Network

Like [5], Lohani et al. [3] propose in their paper, "An Invoice Reading System Using a Graph Convolutional Network", a system for reading digitized invoice images. This approach highlights the most useful billing entities, such as total amount, VAT amount and net amount, without the need of any particular parameterization. According to the authors, the power in the system lies in the fact that the method generalizes to both seen and unseen templates and layouts of invoice, thus making it template agnostic. The authors achieve a complete invoice reading upto 27 classes of interest, without the need of any template configuration or information. It is however noteworthy that the authors never states the number of different vendors that are present in their dataset and it is hence difficult to determine the level of template agnosticism of their model.

2.4 Choice of Basis for Our Thesis

Comparing approaches and models is not a straightforward task, since the research done by the different teams involves different datasets, entities to classify and evaluation metrics. The basis for this thesis is the work done by [3], described in Section 2.3. The reasons for basing our work in [3]'s work, and not the work from the other two described in Sections 2.1 and 2.2, is mainly due to [3]'s model's ability to classify several different entities, its high achieved scores and its supposed ability to generalize across invoice templates.

3 Neural Network Anatomy

The following sections aim to provide a high-level fundamental understanding of machine learning and artificial neural networks in order to facilitate the understanding of the concepts our models are based on, which will be further explained in Section 4.

3.1 Artificial Neural Network Fundamentals

One of the true challenges in machine learning is to solve tasks that are easy for humans to perform but hard for humans to formally explain. In other words, problems humans solve intuitively that feel automatic, e.g., recognizing faces in images, spoken words or written text pose challenges for computers to solve as well. The machine learning solution to these problems can be described as the process of computers learning from experiences and understanding the world in terms of hierarchical concepts that build on top of simpler concepts. This process, in the field of machine learning, is referred to as deep learning. The word "deep" refers to the depth of layers, where a layer is the highest-level building block, in a so-called *artificial neural network* (ANN) [11].

3.1.1 The Artificial Neuron

In order to understand neural networks, one must first be acquainted with the most fundamental component of any ANN, the *artificial neuron*. The neuron is a node in the ANN which receives signals from the environment in the network, gathers all these signals and transmits a single signal of its own, an output, to other connected neurons. In order to compute the output, z, from the neuron, we need to have an input vector $x = [x_1, x_2, \ldots, x_N]$, a set of weights $w = [w_1, w_2, \ldots, w_N]$ and a bias b. The output z can then be computed with Equation 1 [12]:

$$z = \sum_{i=0}^{N} w_i x_i + b. \tag{1}$$

The firing and signal strength of a neuron is controlled by an activation function, to which the output z is passed [12], which will be further explained in Section 3.1.2. Figure 3.1.1 illustrates an artificial neuron.

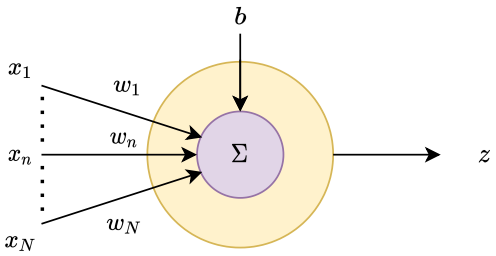


Figure 1: An artificial neuron. Before the signal reaches the node, every input x_n is multiplied by the corresponding weight, w_n.

An ANN is a layered network of neurons and consist of an input layer, a number of hidden layers and an output layer. The neurons in one layer are either partially or fully connected (FC) to the neurons in the next layer [11]. This will be further discussed in Section 3.1.3.

3.1.2 Activation Functions

After we have received an input to the neuron, we want the output from the nodes to be sensitive in certain areas, thus the use of an activation function. The application of an activation function defines which neurons to trigger in each layer, where only the neurons with relevant information are activated. The main utility of applying an activation function is to introduce non-linearity in the network [13], since this allows the model to learn more complex functions. Different types of activation functions can be used for this task, but this thesis will solely utilize the activation functions known as *Rectified Linear Unit* (ReLU) and *softmax*. The ReLU function is defined as

$$ReLU(z) = max(0, z), \tag{2}$$

where the output is equal to z for positive inputs and 0 for negative inputs, which is illustrated in Figure 2 [13].

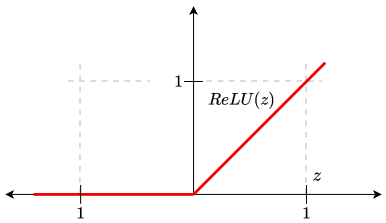


Figure 2: Plot of the ReLU function.

The softmax function normalizes the output from a neural network which converts them to values that sum up to one. The function is defined as:

$$\sigma_{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{3}$$

for $i = 1, \ldots, K$ and $\boldsymbol{z} = [z_1, \ldots, z_K] \in \mathbb{R}^K$ [13]. An illustration of this conversion is shown in Figure 3.

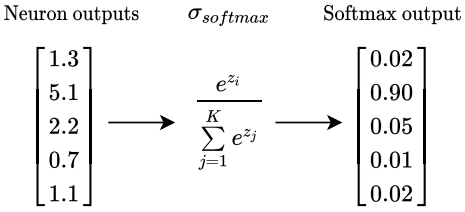


Figure 3: The result of applying softmax to the outputs from the neurons. The values from the softmax output sum up to 1.

3.1.3 Multilayer Perceptron

A *Multilayer Perceptron* (MLP), also referred to as a feedforward artificial neural network, is the archetypal deep learning model. The aim of an MLP is to approximate some function f^*, and it is suitable for classification problems where the inputs to the network are assigned a label or a class. Say we have a classifier $y = f^*(x)$ which maps an input x to a class y, the MLP defines the output of the network as $y = f(x; \theta)$, where θ represents the weights and biases w and b, and learns the value of θ that yields a function approximation as accurate as possible. Between every layer in the network, activation functions are utilized. The procedure of learning to map input values to output values is called *training* [11]. An MLP utilizes a technique called *backpropagation* when training the network, which will be further explained in Section 5.3.

In an MLP network, every unit (neuron) in one layer is connected to every unit in the next layer. In other words, the MLP network is a fully connected one [11]. An MLP consists of three or more layers, Figure 4 illustrates a typical MLP structure.

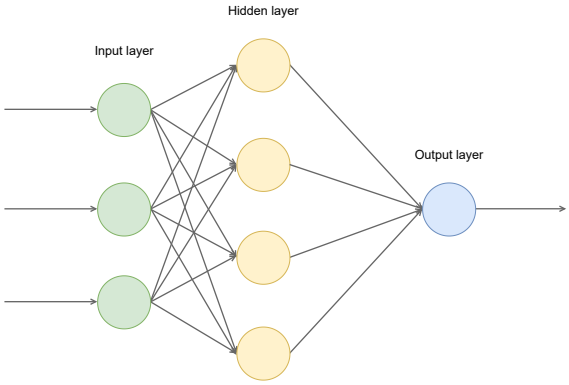


Figure 4: A simple MLP network consisting of an input layer, a hidden layer and an output layer. An MLP consists of three or more layers, meaning the number of hidden layers can differ from MLP to MLP. Modified image from [11].

An MLP is called a feedforward network due to the fact that there are no feedback connections where the outputs of the model are fed back into itself. Instead, all connections in the model move from input to output, layer by layer [11].

3.2 Convolutional Neural Networks

A *Convolutional Neural Network* (CNN) is a specialized kind of neural network, used for processing data that has a grid-like topology. As the name suggests, CNN:s are neural networks that, in place of general matrix multiplication, utilize convolution in at least one of their layers. [11]. Where traditional neural networks' output units interact with every input unit, convolutional neural networks have sparse interactions. This sparsity can be achieved through a *kernel* used to extract features from the input data. Using this approach, it is possible to store fewer parameters which in turn reduces memory requirements. This also improves statistical efficiency and requires fewer operations for computing the output. Figure 5 illustrates the difference in connectivity between the nodes, in regard to convolutional layers and traditional neural network layers [11].

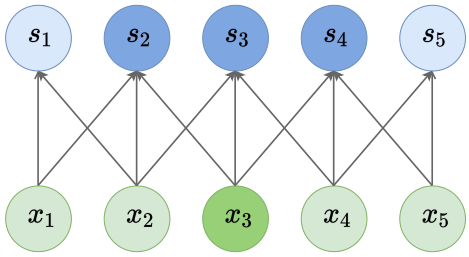


Figure 5: The top layer shows the sparse connectivity using convolution, the bottom layer shows the connectivity using matrix multiplication. With a kernel of 3, only 3 outputs are affected by input x. With matrix multiplication all outputs are affected by input x. Modified image from Goodfellow et al. [11].

CNN:s are typically composed of three different layers – a convolutional layer, a pooling layer and a fully connected layer. The task of the convolutional layer is to take an input consisting of a numerical matrix representation of said input and extract convolved features. Doing so utilizes a filter of a set size, which "walks" through the numerical representation, and the dot product of the entries of the filters as well as the input at a given position thus creating a new, convolved, feature. A simplified example of operations in the convolutional layer are seen in Figure 6. In this example, the 2×2 filter starts at the top left corner of the matrix, and strides one step at a time to the right. This continues until the filter reaches the edge of the input matrix and is then positioned at the same starting position but one step down, row wise. The iteration process stops when the filter reaches the lower right corner of the input matrix [13].

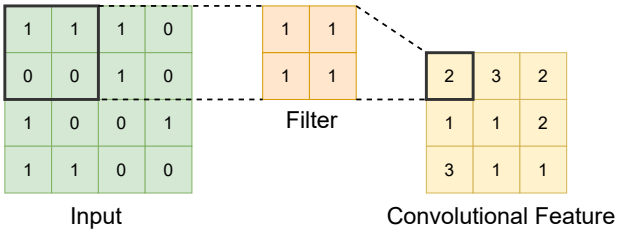


Figure 6: A simple example of a convolutional operation.

In the next layer, the pooling layer, specific functions known as pooling functions are performed to reduce the dimensionality of the network. In other words, the pooling function replaces the output at a certain location with a statistical summary of the nearby outputs [13]. Lastly, the fully connected layer flattens the results before being passed as output.

The deeper the network architecture, the more abstract features we allow the network to learn. For intuitive purposes, the learning process for CNN:s applied to, e.g., an image, can be summarized as follows:

- As the learning goes deeper in the network, the complexity in each filter increases; the first layers learn to detect basic features (e.g., corners and edges)
- The middle layers' filters are trained to detect parts of objects (e.g., a nose, an ear)
- The last and final layers learn to recognize full objects (e.g., a face) for different representations

4 Graph Neural Networks

The following sections build upon Section 3 and aim to further describe the concepts that our model architecture is based on and focus specifically on graph theory coupled with convolutional neural networks.

A *Graph Neural Network* (GNN) is a neural network model adapted for graphs as input. By processing the graph directly, GNN:s preserve the topology information within the graph which would otherwise be lost using traditional processing methods, as the graph would be squished and reshaped into vectors before being fed to a neural network. GNN:s can be combined with different types of extensions, e.g., convolutions, to better facilitate training and learning [14]. To better understand GNN:s and their extensions, it is vital to first understand the fundamentals of graph theory.

4.1 Graph Theory

The concept of a graph can sometimes vary; however this thesis will rely on the definition of a simple graph as explained by [15]:

A graph is constructed through two basic elements: *vertices* (also known as *nodes*) and *edges*. A graph G consists of the pair (V, E), where V is a finite set of vertices and E is an unordered set of pairwise subsets of V called edges, such that the edges in E describe the connections of vertices in V. The edges of a graph can either be directed or undirected depending on whether there exists a directional dependency between vertices or not. An example of an undirected graph consisting of 5 vertices and 6 edges can be seen in Figure 7.

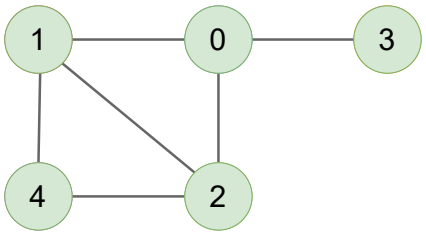


Figure 7: An undirected graph consisting of the vertices $V = \{0, 1, 2, 3, 4\}$ and the edges $E = \{\{0, 1\}, \{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 4\}, \{2, 4\}\}$.

A useful and elegant way to fully represent a graph is through the *adjacency matrix* [16]. If we let $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_N\}$ and the subset $\{v_i, v_j\}$ can be found in E such that $\{v_i, v_j\}$ forms an edge, we say that the vertices v_i and v_j are *adjacent*. From this, we can construct the adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ of G (N = the number of nodes in V), where the elements in $\boldsymbol{A}$ are given by Equation 4.

$$a_{ij} = \begin{cases} 1, & \text{if } v_i \text{ and } v_j \text{ are adjacent,} \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

From this, it follows that for an undirected graph which does not allow for loops, $\boldsymbol{A}$ is symmetric, and the trace of $\boldsymbol{A}$ is zero. The equation for the adjacency matrix for the example graph in Figure 7 can be found in Equation 5. Note that the blue numbers indicate the row and column-wise index positions.

$$\boldsymbol{A} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}. \tag{5}$$

Furthermore, we can also describe the number of edges which terminate at a given vertex, i.e., how many connections a given node in V is part of, by calculating the *degree matrix*. If we let $\boldsymbol{D} \in \mathbb{R}^{N \times N}$ denote the degree matrix of graph G, the elements in $\boldsymbol{D}$ are given by Equation 6.

$$d_{ij} = \begin{cases} deg(v_i), & \text{if } i = j, \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

Where $deg(v_i)$ denotes the degree of vertex v_i, i.e., the number of edges which contains vertex v_i. Equation 8.1.2 illustrates the degree matrix of the graph found in Figure 7.

$$\boldsymbol{D} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix} \end{array}. \tag{7}$$

Another important matrix representation of the graph, especially in the field of spectral graph theory and graph convolutions, is the *Laplacian matrix*, or simply the *Laplacian*. The simplest form of the Laplacian can be derived from the adjacency matrix and the degree matrix, such that $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$, where $\boldsymbol{L} \in \mathbb{R}^{N \times N}$. The elements in $\boldsymbol{L}$ are given by Equation 8.

$$l_{ij} = \begin{cases} deg(v_i), & \text{if } i = j, \\ -1, & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j, \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

Since $\boldsymbol{D}$ is a diagonal matrix and $\boldsymbol{A}$ is symmetric for undirected graphs, it naturally follows that $\boldsymbol{L}$ is also symmetric. Given the same adjacency and degree matrix used in previous examples, the corresponding unnormalized Laplacian is represented in Equation 9.

$$\boldsymbol{L} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 3 & -1 & -1 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 \\ -1 & 1 & 3 & 0 & -1 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & -1 & -1 & 0 & 2 \end{bmatrix} \end{array}. \tag{9}$$

The normalized Laplacian matrix is defined by Equation 10 [17].

$$\begin{aligned} \mathcal{L} &= \boldsymbol{D}^{-\frac{1}{2}} \boldsymbol{L} \boldsymbol{D}^{-\frac{1}{2}} \\ &= \boldsymbol{I}_N - \boldsymbol{D}^{-\frac{1}{2}} \boldsymbol{A} \boldsymbol{D}^{-\frac{1}{2}}. \end{aligned} \tag{10}$$

Here, $\boldsymbol{I}_N$ is the identity matrix of dimension $N \times N$. Recent works in graph convolutions and graph node classification such as [7] and [8] build upon spectral graph theory, which gives rise to the need of transforming the graph into the spectral domain. Through eigen-decomposition of the Laplacian we achieve Equation 11.

$$\mathcal{L} = \boldsymbol{U} \boldsymbol{\Lambda} \boldsymbol{U}^T. \tag{11}$$

$\boldsymbol{\Lambda} \in \mathbb{R}^{N \times N}$ is the diagonal matrix with its diagonal elements being the eigenvalues sorted in ascending order and $\boldsymbol{U} \in \mathbb{R}^{N \times N}$ is composed of the eigenvectors corresponding to the eigenvalues in $\boldsymbol{\Lambda}$ [7]. The

matrices $\boldsymbol{U}$ and $\boldsymbol{U}^T$ can then be used as transformation matrices between the vertex domain and the spectral domain, respectively [7].

4.2 Convolutional Graph Neural Networks

The *Convolutional Graph Neural Network* (ConvGNN) is a special type of GNN that can be seen as a generalization of a CNN [6]. As already mentioned in Section 3.2, CNN:s has the ability to learn local structures in the input data. It achieves this by exploiting the underlying grid-like structure of the data by learning filter weights which correspond to the Euclidean position of the pixels. However, graph data representations are not necessarily neatly structured onto a grid, rather they can be highly irregular and in non-Euclidean domain, removing the possibility to apply a CNN onto graph data out-of-the-box [7], see Figure 8.

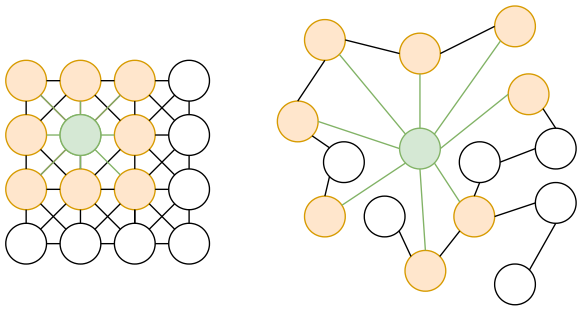


Figure 8: The pixels of an image can be represented as a graph (left) where the green node represents a centered pixel and the orange nodes are its neighbors. To capture this region in a CNN, we simply apply a 3×3 filter, as it is apparent that the neighborhood of the centered vertex naturally aligns with its position in Euclidean space. However, the neighborhood of a vertex in a more general graph (right) is not necessarily tied to the Euclidean space. Furthermore, the number of neighbors of a node in the rightmost graph may vary, while the number of edges from a node in the leftmost graph is fixed (except for nodes at the borders).

However, there are still multiple ways to achieve convolutions on graph data and generalizations of CNN:s has been a hot research topic in recent years [3], [7], [8], [18]. We want to note that there is some discrepancy regarding the umbrella term for Convolutional Graph Neural Networks, where they are sometimes referred to as Graph Convolutional Networks (GCN) [3], and sometimes referred to as ConvGNN, [6]. To avoid confusion between the model family of convolutional graph neural nets and the specific model presented by Kipf & Welling [8] which is also called GCN, we have chosen to use ConvGNN to denote the umbrella term. Whenever we refer to the specific model presented by Defferrard [7] or Kipf & Welling [8], we will use the term ChebNet and GCN, respectively.

Recent ConvGNN:s often have a similar structure, where the input of a ConvGNN model consists of [6]:

- A feature matrix $\boldsymbol{X} \in \mathbb{R}^{N \times F}$ where N is the number of vertices and F is the number of features for each node.
- A representation of a graph in matrix form, often the adjacency matrix $\boldsymbol{A}$ or some variation thereof.

The goal is then, for some graph $G = (V, E)$, to learn a function f of the features in G which produces a node-level output $\boldsymbol{Z} \in \mathbb{R}^{N \times F'}$, where F' is the number of node-level output features. If we let $\boldsymbol{H}^{(l)}$ be the output feature matrix at the l^{th} layer, then $H^{(0)} = \boldsymbol{X}$ and $\boldsymbol{H}^{(L)} = \boldsymbol{Z}$, where L is the number of layers. A single layer in the neural network can then be described as according to Equation 12.

$$\boldsymbol{H}^{(l+1)} = f(\boldsymbol{H}^{(l)}, \boldsymbol{A}). \tag{12}$$

Specific ConvGNN models, such as ChebNet [7] and GCN [8], then only differs in how $f(\cdot, \cdot)$ is chosen and parameterized [6].

As discussed in 3.2, the pooling layer is an important building block for most CNN:s and can indeed be implemented for graph convolutions as well [7]. However, this technique is used when classifying complete graphs, not individual nodes within graphs. Hence, we omit the pooling layer completely in this thesis.

4.2.1 Spectral Graph Convolutions

A certain family of ConvGNN:s of which both ChebNet and GCN belong to is the s*pectral based* graph convolution family. Spectral based graph convolutions are based on the idea that node features can be treated as signals and that convolutions of signals in the vertex domain can be treated as multiplication in the spectral domain [8], [7]. To operate in the spectral domain, the graph feature matrix $\boldsymbol{H}^{(l)} \in \mathbb{R}^{N \times F}$, where N is the number of nodes and F is the number of node features at layer l, first needs to be transformed through a change of basis. Recall that we can derive the normalized Laplacian matrix $\mathcal{L}$ from the adjacency matrix $\boldsymbol{A}$ of an undirected graph as defined by Equation 10, and that $\mathcal{L}$ can be eigen-decomposed to $\boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T$ as defined by Equation 11. The eigenvectors in $\boldsymbol{U}$ can be used to form a new basis in the spectral domain. The spectral transformation $\hat{\boldsymbol{H}}^{(l)}$ of input matrix $\boldsymbol{H}^{(l)}$ is defined by [19]

$$\hat{\boldsymbol{H}}^{(l)} = \boldsymbol{U}^T \boldsymbol{H}^{(l)}, \tag{13}$$

and the transformation back to the vertex domain is defined as

$$\boldsymbol{H}^{(l)} = \boldsymbol{U}\hat{\boldsymbol{H}}^{(l)}. \tag{14}$$

We can also understand the eigenvalues $\boldsymbol{\Lambda}$ of $\mathcal{L}$ as the frequencies of the graph. Hence, we can construct a filter g_θ where θ are learnable parameters, operating on the Laplacian as such [7]:

$$\begin{aligned} g_\theta(\mathcal{L}) &= g_\theta(\boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T) \\ &= \boldsymbol{U} g_\theta(\boldsymbol{\Lambda})\boldsymbol{U}^T. \end{aligned} \tag{15}$$

Through Equation 13, 14 and 15, convolving $\boldsymbol{H}^{(l)}$ with g_θ can now be expressed as

$$\begin{aligned} \boldsymbol{H}^{(l)} * g_\theta(\mathcal{L}) &= g_\theta(\mathcal{L})\boldsymbol{H}^{(l)} \\ &= \boldsymbol{U} g_\theta(\boldsymbol{\Lambda})\boldsymbol{U}^T\boldsymbol{H}^{(l)}. \end{aligned} \tag{16}$$

Simply put, the idea is to transform $\boldsymbol{H}^{(l)}$ to the frequency domain, convolve the frequencies through the filter operation and then convert the features back to the vertex domain. Different spectral based graph convolutions mainly differ in the formulation of filter g_θ [6].

4.2.2 ChebNet

The ChebNet convolution layer is designed using the following filter for graph convolutions [7]:

$$g_\theta(\mathcal{L}) = \sum_{k=0}^{K-1} \theta_k T_k(\tilde{\mathcal{L}}). \tag{17}$$

As both the eigen-decomposition of $\mathcal{L}$ and the change of basis between the vertex domain and the spectral domain needed in Equation 16 are costly operations, the proposed filter g_θ is parameterized as a polynomial function of $\mathcal{L}$ and is calculated as a *Chebychev polynomial approximation*. As proven by Hammond et al. [20], by using a Chebyshev polynomial approximation for computing $g_\theta(\mathcal{L})$, we bypass the need for eigen-decomposing $\mathcal{L}$ as well as the need for transforming the feature matrix between domains. The k^{th} order Chebychev polynomial of x, denoted as $T_k(x)$, is defined as

$$
\begin{aligned}
T_k(x) &= 2xT_{k-1}(x) - T_{k-2}(x) \\
T_0(x) &= 1 \\
T_1(x) &= x.
\end{aligned} \tag{18}
$$

Through this Chebyshev polynomial approximation of the convolution filter, we not only lower the time complexity of the convolution, we also allow for the filter to be applied locally. The parameter K denotes the locality of the filter, i.e., we center the filter with a radius of K hops around a node. In other words, K is analogous to the filter size of a CNN. By using a Chebyshev approximation of $g_\theta(\mathcal{L})$ with locality K, we reduce the time complexity of the entire filtering operation from $O(n^3)$ down to $O(K|E|)$ [7] [6], where $|E|$ is the number of edges in the graph. $\tilde{\mathcal{L}}$ is a scaled version of $\mathcal{L}$ such that

$$
\tilde{\mathcal{L}} = \frac{2\mathcal{L}}{\lambda_{max}} - \boldsymbol{I}_N. \tag{19}
$$

where λ_{max} is the largest eigenvalue of $\mathcal{L}$. This rescaling is necessary as the Chebyshev polynomials form an orthonormal basis on the interval $[-1, 1]$ and the original Laplacian lies within the interval $[0, \lambda_{max}]$. θ_k are learnable Chebyshev coefficients at hop k, such that $\theta_k \in \mathbb{R}^{F \times F'}$, where F is the number of features going into the model and F' is the length of the feature map going out of the model.

The spectral convolution of a feature matrix $\boldsymbol{H}^{(l)}$ can now be defined as

$$
\begin{aligned}
\boldsymbol{H}^{(l)} * g_\theta &= g_\theta(\tilde{\mathcal{L}})\boldsymbol{H}^{(l)} \\
&= [\hat{\boldsymbol{H}}_0^{(l)}, \hat{\boldsymbol{H}}_1^{(l)}, \ldots, \hat{\boldsymbol{H}}_{K-1}^{(l)}]\boldsymbol{\Theta}^{(l)},
\end{aligned} \tag{20}
$$

where

$$
\begin{aligned}
\hat{\boldsymbol{H}}_k^{(l)} &= 2\tilde{\mathcal{L}}\hat{\boldsymbol{H}}_{k-1}^{(l)} - \hat{\boldsymbol{H}}_{k-2}^{(l)} \\
\hat{\boldsymbol{H}}_0^{(l)} &= \boldsymbol{H}^{(l)} \\
\hat{\boldsymbol{H}}_1^{(l)} &= \tilde{\mathcal{L}}\boldsymbol{H}^{(l)},
\end{aligned} \tag{21}
$$

and where $\boldsymbol{\Theta}^{(l)} = [\theta_0^{(l)}, \theta_1^{(l)}, \ldots, \theta_{K-1}^{(l)}]$ is a vector of size K containing the learnable parameters for each hop at the l^{th} layer. Now, a single layer can be described as

$$
\boldsymbol{H}^{(l+1)} = \sigma(\sum_{k=0}^{K-1} \hat{\boldsymbol{H}}_k^{(l)}\theta_k^{(l)}), \tag{22}
$$

where $\boldsymbol{H}^{(l+1)} \in \mathbb{R}^{N \times F'}$. For a graphical illustration of how Equation 22 convolves the features of a single node with its $K = 3$ neighborhood, please refer to Figure 9.

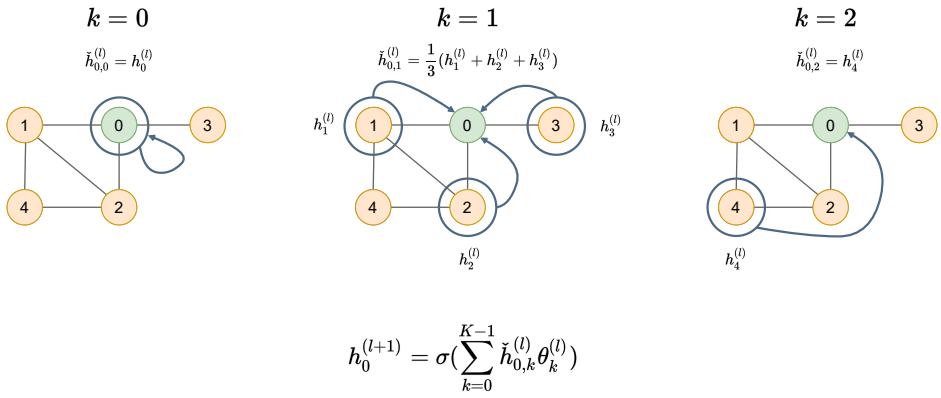


Figure 9: A Chebyshev convolution for $K = 3$ for one specific node (node 0). The adjacent nodes affecting the feature representation of node 0 are indicated with a circle. $\theta_k^{(l)}$ represent the $F \times F'$ dimensional weights at hop k.

4.2.3 GCN

Kipf & Welling [8] present the following architecture for a multi-layered GCN:

$$\boldsymbol{H}^{(l+1)} = \sigma(\tilde{\boldsymbol{D}}^{-\frac{1}{2}} \tilde{\boldsymbol{A}} \tilde{\boldsymbol{D}}^{-\frac{1}{2}}) \tag{23}$$

To understand this architecture, we once again consider graph $G = (V, E)$ where $V \in \mathbb{R}^N$, the corresponding adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ and some input matrix $\boldsymbol{H}^{(l)} \in \mathbb{R}^{N \times F}$ of node features at layer l. The GCN layer proceeds with a similar idea of a spectral filter as ChebNet, as defined by Equation 17. However, GCN limits the number of K-hops per layer to 1, such that the features of a centered vertex only convolve with features of nodes with which it shares an edge. The idea is then to stack multiple layers in succession to receive a set of useful convolutional filter functions instead of relying on the parametrization of K [8]. Furthermore, λ_{max} in Equation 19 is approximated to $\lambda_{max} \approx 2$. This simplification, in conjunction with the definition of $\mathcal{L}$ stated in Equation 10, simplifies Equation 19 as follows:

$$\begin{aligned} \tilde{\mathcal{L}} &= \mathcal{L} - \boldsymbol{I}_N \\ &= -\boldsymbol{D}^{-\frac{1}{2}} \boldsymbol{A} \boldsymbol{D}^{-\frac{1}{2}}. \end{aligned} \tag{24}$$

Under the simplifications made in Equation 24 as well as by limiting the number of hops to 1, the convolution of a feature vector $h_i^{(l)} \in \boldsymbol{H}^{(l)}$ for node i and the spectral filter g_θ can be defined as

$$\begin{aligned} h_i^{(l)} * g_\theta(\tilde{\mathcal{L}}) &= \theta_0 h_i^{(l)} + \theta_1 (\mathcal{L} - \boldsymbol{I}_N) h_i^{(l)} \\ &= \theta_0 h_i^{(l)} - \theta_1 \boldsymbol{D}^{-\frac{1}{2}} \boldsymbol{A} \boldsymbol{D}^{-\frac{1}{2}} h_i^{(l)}. \end{aligned} \tag{25}$$

Here, θ_0 and θ_1 are parameter vectors for $k = 0$ and $k = 1$. To reduce the risk of overfitting, [8] further approximate $\theta_0 = -\theta_1$ and reformulate the parameters as a single parameter $\boldsymbol{\theta}$, such that the convolution expression can be rewritten as

$$h_i^{(l)} * g_\theta(\tilde{\mathcal{L}}) \approx \boldsymbol{\theta}(\boldsymbol{I}_N + \boldsymbol{D}^{-\frac{1}{2}} \boldsymbol{A} \boldsymbol{D}^{-\frac{1}{2}}) h_i^{(l)}. \tag{26}$$

Since $\boldsymbol{D}^{-\frac{1}{2}} \boldsymbol{A} \boldsymbol{D}^{-\frac{1}{2}}$ has eigenvalues in the range $[0, 2]$, the expression is renormalized in terms of $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{D}}$ as such:

$$\boldsymbol{I}_N + \boldsymbol{D}^{-\frac{1}{2}} \boldsymbol{A} \boldsymbol{D}^{-\frac{1}{2}} \rightarrow \tilde{\boldsymbol{D}}^{-\frac{1}{2}} \tilde{\boldsymbol{A}} \tilde{\boldsymbol{D}}^{-\frac{1}{2}}. \tag{27}$$

Here, $\tilde{\boldsymbol{A}}$ is the adjacency matrix of the original graph G with added self-loops, such that $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \lambda \boldsymbol{I}_N$, i.e., every node is also set as a neighbor to itself. λ is a potentially trainable parameter which adjusts the importance of features for the centered vertex. However, Kipf & Welling [8] fixes $\lambda = 1$ in their report. $\tilde{\boldsymbol{D}}$ is the degree matrix of $\tilde{\boldsymbol{A}}$. This renormalization reduces the risk for numerical instabilities and vanishing/exploding gradients during backpropagation (see Section 5.3).

This far, the theory behind a GCN has been explained in terms of a single node feature vector $h_i^{(l)}$. The idea can however be generalized over a feature matrix $\boldsymbol{H}^{(l)}$ as such:

$$\boldsymbol{H}^{(l)} * g_\theta(\tilde{\mathcal{L}}) = \tilde{\boldsymbol{D}}^{-\frac{1}{2}} \tilde{\boldsymbol{A}} \tilde{\boldsymbol{D}}^{-\frac{1}{2}} \boldsymbol{H}^{(l)} \boldsymbol{\theta}^{(l)}, \tag{28}$$

where $\boldsymbol{\theta}^{(l)} \in \mathbb{R}^{F \times F'}$ is the trainable weight matrix at the l^{th} layer. By applying a nonlinearization function $\sigma(\cdot)$ to Equation 28, we end up with the Equation for the output feature map $\boldsymbol{H}^{(l+1)} \in \mathbb{R}^{N \times F'}$ stated in Equation 23.

5 Model Training

For a machine learning model to be of any real use it first needs to be trained on data. In the following section we will go through common techniques used for adjusting the weights and biases of a deep learning model. Sections 5.1 - 5.3 will explain how to measure the difference between a model's prediction and the "true" value as well as how this measurement can be used to update the model's parameters. Section 5.4 will explain the concept of epochs and batches and Section 5.5 will go through some pitfalls when training a model and how to counter them.

5.1 Loss Function

In order to determine how well a model fits the training data, it is essential to measure how the model's predictions compare to the true values, i.e., how big the difference is between the two. Functions that describe these differences are called *loss functions*, which takes both the predicted values and the true values as input, such that $L(\hat{y}(x_i;\theta), y_i)$ [13]. Here, L is the loss function, $\hat{y}(x_i;\theta)$ is the model output based on the feature vector x_i for a single data point i and the model parameters θ, while y_i denotes the true value of i. The loss function calculates the difference between the model output and the true value for *each individual* data point in the dataset.

An example of a loss function, which is also used in this thesis, is the *Multiclass Cross Entropy* (CE) loss function [21]. This is, as the name entails, a loss function for multiclass machine learning problems and operates on the logits produced by a softmax activation function. The logits from the softmax output can be interpreted as a form of certainty of membership for each class [13]. The CE loss function assigns a low loss when the certainty for the correct class is high, and a high loss when the inverse is true. We calculate the cross entropy by taking the negative log likelihood for each true label as such:

$$L(\hat{y}(x_i;\theta), y_i) = -\sum_{m=1}^{M} (y_{im} log(\sigma(\hat{y}(x_i;\theta))_m)). \tag{29}$$

Here, M is the total number of classes and $\sigma(\cdot)$ denotes the softmax activation function. If we imagine a multiclass classification problem with three possible classes and datapoint i belongs to the third class, then y_i would be a vector such that $y_i = [0, 0, 1]$. Imagine also that the model output $\hat{y}(x_i;\theta)$ for the same data point is equal to $\hat{y}(x_i;\theta) = [1, 3, 5]$ and $softmax(\hat{y}(x_i;\theta)) \approx [0.016, 0.117, 0.867]$. The loss would then be calculated as:

$$\begin{aligned} L(\hat{y}(x_i;\theta), y_i) &= -\sum_{m=1}^{3} (y_{im} log(softmax(\hat{y}(x_i;\theta))_m)) \\ &\approx -1 \times (0 \times log(0.016) + 0 \times log(0.117) + 1 \times log(0.867)) \\ &\approx 0.06. \end{aligned} \tag{30}$$

In this example, the model is fairly certain that data point i does indeed belong to the third class, leading to a low loss.

By averaging the loss over all data points, we achieve the cost function $J(\theta)$ which is defined as follows [13]:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(\hat{y}(x_i;\theta), y_i). \tag{31}$$

Here, N denotes the total number of data points in the training dataset. The cost function now gives us the average loss over all predictions for the whole dataset and functions as a measure for how well the model on average fits the training data.

5.2 Gradient descent

Training a model implies that we choose the parameters θ such that we minimize the cost function $J(\theta)$ in order to ensure that the difference between the model predictions $\hat{y}$ and the true values y become as small as possible. The parameter learning problem can then be defined as follows [13]:

$$\theta_{learned} = \arg\min_{\theta} J(\theta). \tag{32}$$

If θ consisted of a single unknown scalar ϑ such that $J(\theta) = J(\vartheta)$, a simple way of optimizing $J(\vartheta)$ would be to calculate the derivative $J'(\vartheta)$ and adjust ϑ with some step size $\gamma \geq 0$, also called the *learning rate*, in the negative direction of the slope [11]. ϑ can then be updated as follows:

$$\vartheta^{(k+1)} = \vartheta^{(k)} - \gamma J'(\vartheta^{(k)}). \tag{33}$$

Where k denotes the current iteration of ϑ. In neural networks however, θ is most often not a scalar but a matrix consisting of multiple parameters, in which case a simple derivative is not sufficient. However, the same concept can be reapplied using the partial derivatives of $J(\theta)$ with respect to all parameters in θ. By calculating the *gradient* $\nabla_\theta J(\theta)$, we get a vector consisting of the partial derivatives with respect to the parameters in θ, where the element on index position i in $\nabla_\theta J(\theta)$ corresponds to the parameter on index position i in θ [11]. Similarly to Equation 33 we can now update the parameters in θ through:

$$\theta^{(k+1)} = \theta^{(k)} - \gamma \nabla_\theta J(\theta^{(k)}). \tag{34}$$

The act of learning new parameters in θ by adjusting them in the negative direction of their partial derivative counterpart in $\nabla_\theta J(\theta)$ is called gradient descent [11].

5.3 Backpropagation

Before adjusting the parameters in accordance with the gradient, the gradient first needs to be computed. This is often done using the *backpropagation*, which builds on the concepts of the calculus chain rule [13]. The calculus chain rule [22] states that:

> If $z = f(y)$ is is a differentiable function of y and $y = g(x)$ is a differentiable function of x, then $z = f(g(x))$ is a differentiable function of x and $\frac{dz}{dx} = \frac{dz}{dy} \times \frac{dy}{dx}$.

This idea can be expanded and applied to the cost, weights and biases of a neural network. Consider a neural net with L layers, where each layer l consists of the following components:

$$\begin{aligned} z^{(l)} &= \boldsymbol{W}^{(l)} q^{(l-1)} + b^{(l)} \\ q^{(l)} &= \sigma(z^{(l)}), \end{aligned} \tag{35}$$

with the last output layer:

$$\begin{aligned} z^{(L)} &= \boldsymbol{W}^{(L)} q^{(L-1)} + b^{(L)} \\ q^{(L)} &= \sigma(z^{(L)}), \end{aligned} \tag{36}$$

where $q^{(L)}$ is the model prediction $\hat{y}$ and is evaluated by some cost function $J(\hat{y}, y)$. To compute the change in J with respect to the weights $\boldsymbol{W}^{(L)}$, we would through the chain rule achieve:

$$\frac{\partial J}{\partial \boldsymbol{W}^{(L)}} = \frac{\partial J}{\partial q^{(L)}} \times \frac{\partial q^{(L)}}{\partial z^{(L)}} \times \frac{\partial z^{(L)}}{\partial \boldsymbol{W}^{(L)}}. \tag{37}$$

By evaluating how changes in $\boldsymbol{W}^{(L)}$ affects $z^{(L)}$, how changes in $z^{(L)}$ affects $q^{(L)}$ and how changes in $q^{(L)}$ in turn affects J, we can measure how changes in $\boldsymbol{W}^{(L)}$ affects J. This can analogously be done for the biases in $b^{(L)}$ as well. From here, it is possible to recursively calculate the partial derivatives further down in the network, caching the already calculated derivatives to optimize calculation times [13]

[11]. The partial derivative of J with respect to the weights in layer $L-1$ would then be calculated as follows:

$$\frac{\partial J}{\partial \boldsymbol{W}^{(L-1)}} = \frac{\partial J}{\partial q^{(L)}} \times \frac{\partial q^{(L)}}{\partial z^{(L)}} \times \frac{\partial z^{(L)}}{\partial q^{(L-1)}} \times \frac{\partial q^{(L-1)}}{\partial z^{(L-1)}} \times \frac{\partial z^{(L-1)}}{\partial \boldsymbol{W}^{(L-1)}}. \tag{38}$$

Notice that the first two terms in 38 have already been calculated in 37 and the cached results can be reused to save on computation time. This is done for all weights and biases throughout the layers in the network, starting with the last layer and moving backwards through the network, reusing the partial derivatives that have already been calculated in previous layers to construct $\nabla_\theta J$ [13]. This, in essence, is the backpropagation algorithm.

5.4 Stochastic Gradient Descent & Mini-batching

For very large datasets, it might not be feasible or computationally efficient to calculate the gradient of the cost function for the whole dataset each time. However, by calculating the gradient over smaller subsamples of the dataset, also known as *mini-batches*, we can still get an approximation of the gradient over the whole dataset [13]. The cost function for a mini-batch would then be defined as:

$$J(\theta) = \frac{1}{N_b} \sum_{i=1}^{N_b} L(\hat{y}(x_i; \theta), y_i), \tag{39}$$

where N_b denotes the size of a mini-batch. The model parameters θ can subsequently be updated based on the gradient of a single mini-batch as follows:

$$\theta^{(t+1)} = \theta^{(t)} - \gamma \frac{1}{N_b} \nabla_\theta \sum_{i=1}^{N_b} L(\hat{y}(x_i; \theta^{(t)}), y_i), \tag{40}$$

A full sweep over the whole dataset, where θ has been updated based on the gradients of all subsequent mini-batches, is called an *epoch*. When splitting the dataset into mini-batches, it is important to ensure that the data points in the set are independent from each other [11]. In the case of invoices, it might be the case that the first 100 invoices are all from the same vendor, the next 100 invoices are from another vendor and so on, depending on how the data was collected. In that case, each mini-batch would be heavily biased towards a specific vendor which might have a negative impact on the algorithm's effectiveness [11]. To ensure that each mini-batch represents the whole dataset as well as possible, the dataset should be shuffled at the start of each epoch before being split into mini-batches.

In the context of graph neural networks, a mini-batch m of k invoices where m contains N nodes, m consists of [23]:

1. A feature matrix $\boldsymbol{X} \in \mathbb{R}^{N \times F}$
2. A label vector $\boldsymbol{y} \in \mathbb{R}^N$
3. An unlinked adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ consisting of adjacency matrices $\boldsymbol{A_1}, \ldots, \boldsymbol{A_k}$.

To illustrate this, imagine a mini-batch m containing invoices I, J and K, where each invoice is represented as a graph by feature matrix $\boldsymbol{X_i}$, label vector y_i and adjacency matrix $\boldsymbol{A_i}$. The graph representation of m would then be constructed as illustrated in Figure 10.

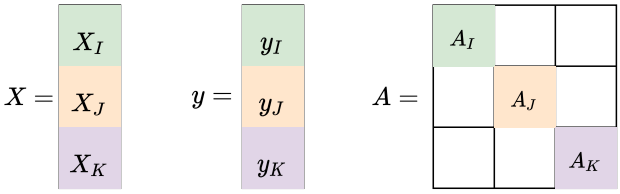


Figure 10: A representation of how 3 invoices I, J and K are stacked to produce a mini-batch. Notice how the adjacency matrices are stacked diagonally in $\boldsymbol{A}$, this ensures that nodes from one invoice can not propagate its node features to nodes in the next invoice.

In other words, the node features and node labels are simply concatenated in the node dimension. The adjacency matrices of invoice I, J and K are stacked diagonally such that the mini-batch functions as one large graph, consisting of multiple unlinked subgraphs.

5.5 Overfitting

As discussed in Section 5.3, a neural network is trained through the use of backpropagation. As a reminder, the backpropagation algorithm calculates the gradient of the cost function to find the direction at which the weights and biases lead to an increased cost of the model predictions. By adjusting the weights and biases in the opposite direction of the gradient with some step size, the model learns to minimize the cost of its predictions with respect to the data. With this in mind, it is tempting to believe that more training always leads to a better performing model, which is not always the case. If the model is left to learn from the data for too long, it might tune its parameters to fit the data too well, which leads to a very complex model that performs well on the training data but that generalizes badly over new, previously unseen, data. This is called *overfitting* [13] and in this section we will describe a few methods used in this thesis to reduce the risk of this, namely train/test/validation splitting, early stopping, L_2-regularization and DropEdge.

5.5.1 Train, Test & Validation Splits

A common tactic for combatting overfitting is to divide the available dataset into two randomly shuffled and non-overlapping subsets called the *training set* and the *validation set* [13]. A third randomly shuffled and non-overlapping set, known as the *test set*, is utilized in conjunction with the previous two. While the test set itself does not contribute to a reduction of model complexity, it does give a fair estimation of the final model performance. The following section will describe the use of these three sets.

The training set is used to train the model: this set contains the data which is fed to the model and used to adjust the model parameters in the backpropagation algorithm. The training set is commonly split into mini-batches as described by Section 5.4 and a full sweep over the batches in the training set is what constitutes an epoch.

After training a model for one full epoch, the model can be evaluated on some metric, e.g., the cost or any of the metrics mentioned in Section 6, using the validation set. The validation set gives an unbiased approximation of the model performance, since no data from the validation set has been used during the training process. Each time a model performs better on the validation set, a checkpoint of the best-performing model with respect to the validation set can be saved. By selecting a model based on metrics calculated on the validation set and not on the training set, the risk of overfitting the model to the training data is reduced [13].

After training and selecting the best-performing model, it is common practise to evaluate the model on some metric. Since the model is chosen specifically for its performance on the validation set, any evaluation of model performance will be biased if the validation set is re-used for this evaluation [13]. Hence, the test set is used as a final set of data points to get a fair evaluation of the model performance on unseen data. In other words, the test set itself does little to prevent overfit, but rather gives a fair estimation of the final model performance.

5.5.2 Early Stopping

Early stopping is a method often used in conjunction with the technique of selecting a model based on the validation set, which might further decrease the risk of overfitting. Early stopping builds upon the idea

that after some number of epochs, the training error and the validation error will start to diverge, i.e., the model starts to overfit to the training data and generalization over unseen data worsens. By setting a window size, also known as the *patience*, for the number of epochs which the validation performance has not increased, the training can be stopped before the set number of epochs has passed. By prematurely ending the training process after n consecutive iterations of not improving the validation performance, the risk of overfitting potentially decreases even further. However, implementing such a method requires some level of fine tuning, since stopping the training too early may result in a model that is too simple, while stopping the training too late may result in a model that still overfits to the training data [24].

5.5.3 L_2 Regularization

L_2-regularization, also known as *weight decay* or *ridge regression*, is another technique for reducing the risk of overfitting a model to the training data. L_2 Regularization introduces a penalty term $\|\theta\|_2^2$ to the cost function $J(\theta)$ where θ is the model parameters, such that:

$$J(\theta)_{regularized} = J(\theta) + \frac{\lambda}{2} \|\theta\|_2^2 . \tag{41}$$

Since the model parameters are learned through minimizing the cost function, the introduced penalty term forces the model to both keep the original cost function small as well as pushing the weights θ of the model towards zero, reducing the risk of the model overfitting its parameters to the training data. λ is called the *regularization parameter* and is a design choice [13]. The choice of $\lambda = 0$ is the same as omitting L_2 regularization completely, while larger λ penalizes larger values for θ more heavily.

5.5.4 DropEdge

Yet another method of battling overfitting is to fit all possible neural network models on the same dataset in order to average each model prediction, creating an ensemble of models. This ensemble is most useful when the individual models are different from each other, which in a neural network setting means they should either be trained on different data or have different architectures. Training several neural networks with different architectures is a daunting task, since finding optimal hyperparameters for each one of these is time consuming and with each network comes potentially a vast number of computations. In order to combat this, a single model can be implemented to simulate having a large number of neural network architectures by, given a certain probability, randomly dropping some units (i.e., neurons) in the network along with their corresponding connections between other network units. The method of utilizing this technique is called *dropout* and is used in neural networks to battle overfitting [25]. When developing graph convolutional neural networks, however, another main obstacle is the problem of over-smoothing the model. Over-smoothing means that node representations become more and more similar to each other, eventually resulting in indistinguishable nodes. More specifically, the representations of adjacent nodes in graph convolutions are pushed to mix with each other such that, in the extreme case of having an infinite number of layers, all representations of the nodes converge to a stationary point resulting in vanishing gradients. This makes the nodes unrelated to the input features and occurs through the effect of Laplacian smoothing, hence the name over-smoothing [26].

While dropout combats overfitting, a technique known as *DropEdge*, proposed by Rong et al. [10], is one used specifically for graph input to battle both overfitting and over-smoothing. Instead of dropping layerwise units as in dropout [25], DropEdge randomly removes edges from the graph by removing some of the adjacency matrix's connections according to a certain probability. This is done at each training batch stage, generating a "new" dataset for each stage. This results in a slower convergence speed but in turn tackles over-smoothing [10].

When applying DropEdge to a neural network, different random deformed copies are generated from the original graph, thus the randomness and the diversity of the input data is augmented and in turn prevents overfitting to a higher degree. [26].

DropEdge is only used during training, and at each batch DropEdge enforces Vp non-zero elements in the adjacency matrix, given a probability p, to be converted to zeroes, where V is the total amount of edges present in the adjacency matrix and p is the probability of dropping an edge. The original adjacency matrix is denoted as $\boldsymbol{A}$ and we get the resulting adjacency matrix $\boldsymbol{A}_{drop}$. $\boldsymbol{A}$ is then calculated as

$$\boldsymbol{A}_{drop} = \boldsymbol{A} - \boldsymbol{A}', \tag{42}$$

where $\boldsymbol{A}'$ is the sparse matrix expanded from $\boldsymbol{A}$ [26]. An example of applying DropEdge for the graph in Figure 7 can be seen in Figure 11.

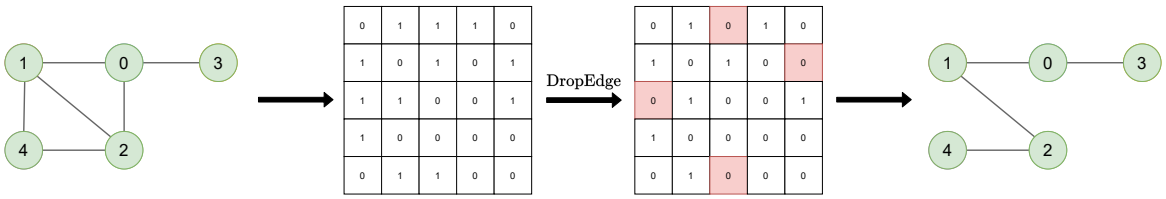


Figure 11: Example of illustration of applying DropEdge, with an arbitrary probability p, on the graph from Figure 7. Red squares indicate which edges that have been dropped. The far-right graph is the result after dropping the edges.

6 Evaluation Metrics

When building machine learning models, it is not only important to build a model but to modify it using certain feedback on the model's performance. Using an evaluation metric quantifies the performance of that predictive model into useful feedback. There are a number of important evaluation metrics to take into consideration when constructing these models, namely the *accuracy, precision, recall* and $F_1 score$ [13]. A common property of all these metrics is that they all fall in the range [0, 1] where a value close to 1 is good and a value close to 0 indicates a problem with the model. Before delving too deep into these metrics, however, it would be wise to first mention the confusion matrix.

6.1 Confusion Matrix

The confusion matrix is a useful table which describes the performance on a set of test data where the true values are known. The validation data is split into the four groups true negative, false negative, false positive and true positive, depending on y and $\hat{y}$ (the output predicted by the model). The four groups can be exemplified with a classifier which predicts the presence of cancer in patients: True positives (TP) indicate predictions where the patient has cancer, and in reality does have it; True negatives (TN) indicate predictions of patients not having cancer and where the patient indeed did not have cancer; False positives (FP), also called *Type I error*, indicate predictions of patients having the disease when they in fact do not have it; False negatives (FN), also known as *Type II error*, indicates predictions of patients not having the disease when they in fact do have it [13]. An illustration of a confusion matrix is shown in Table 1.

Table 1: Confusion matrix for a binary classifier. 1 denotes "has cancer" and -1 denotes "does not have cancer", in the cancer example. Columns represent the true value, while rows represent the predicted value.

	$y = -1$	$y = 1$	Total
$\hat{y}(x) = -1$	TN	FN	$N*$
$\hat{y}(x) = 1$	FP	TP	$P*$
Total	N	P	n

6.2 Accuracy

However, the confusion matrix is seldom used as a metric in itself but is rather used to derive other metrics from it. One such metric is the *accuracy*, i.e., how many of the predictions that were correctly classified out of the total dataset [13]. The accuracy can be calculated as:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}. \tag{43}$$

The accuracy metric, while popular, can often be misleading. Let us go back to the example of patients with cancer and let us assume that we have a heavily imbalanced dataset where each row represents a patient and only 0.1 % of all patients in the dataset have cancer. A model trained for classifying said patients as either "has cancer" or "does not have cancer" would still get an accuracy of 99.9 % by classifying all patients as "does not have cancer". Depending on the problem at hand and the cost for misclassification, accuracy might not be the best choice for evaluating the model.

6.3 Recall

Another useful metric is the *recall*, which is often used when the cost of false negatives is high, i.e., it is a measure of the Type II error. The recall metric is calculated as:

$$recall = \frac{TP}{P} = \frac{TP}{TP + FN}. \tag{44}$$

If we re-evaluate our previous model which predicts all patients as not having cancer on a dataset consisting of 10 000 patients, we can calculate the corresponding confusion matrix as seen in Table 2.

Table 2: Confusion matrix for a binary classifier. 1 denotes "has cancer" and -1 denotes "does not have cancer", in the cancer example. Columns represent the true value, while rows represent the predicted value.

	$y = -1$	$y = 1$	Total
$\hat{y}(x) = -1$	9990	10	10 000
$\hat{y}(x) = 1$	0	0	0
Total	9990	10	10 000

From the confusion matrix, we get that $TP = 0$ and that $P = 10$ which we can use to calculate the recall as

$$recall = \frac{TP}{P} = \frac{0}{10} = 0. \tag{45}$$

Even if the model would correctly predict cancer in one of the patients, the recall would still only be 0.1 which, in the case evaluating a model which predicts cancer in patients, probably is a better choice of metric compared to the accuracy.

6.4 Precision

Precision is a metric used when, instead of assigning a high cost to false negatives, we assign a high cost to false positives [13], i.e., Type I errors. This metric is favorable in situations such as spam detection, where an email classified as "spam" when in fact it is not may result in loss of important information. The precision metric is calculated as follows:

$$precision = \frac{TP}{P*} = \frac{TP}{TP + FP}. \tag{46}$$

6.5 F_1-Score

Analyzing recall and precision on their own could also lead to misleading results. Using the same dataset of patients as mentioned before, processed by a model which classifies all patients as having cancer would result in a perfect recall of 1, although with a less-than-desirable precision of 0.001. This tradeoff between recall and precision is very common, where an increase in one of these metrics leads to a decrease in the other [27]. One measurement of this trade-off is called the F_1*-score*, F*-score*, F*-measure* or simply F_1 [27] which may also be used as a performance metric. The F_1-score is the summary of the precision and recall by their harmonic mean:

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall}. \tag{47}$$

While the F_1-score might be harder to interpret than the metrics mentioned earlier, it is still a very useful metric since it, for datasets which have a heavy class imbalance in favor of the true negative class in the dataset, effectively captures the trade-off between the precision and the recall [13].

6.6 Macro Average

When evaluating performance across multiple classes, macro averaging is a useful evaluation metric method which averages some metric (e.g., F_1-score, recall or precision) over all classes and gives an equal weight to every class regardless of frequency [27]. This means that, for a classification problem with 3 classes $C \in \{X, Y, Z\}$, even though class X may contain 99 % of all data and class Y and Z contain 0.5 % of the data each, the precision or recall score from each class would still contribute equally to the macro average.

To compute the macro average of either precision or recall for classes $C \in \{X, Y, Z\}$, we first create a global contingency table, whose cell values are the sums of the related cells in the per class confusion matrices, in order to then use this table to compute the averaged performance scores, see Table 3.

Table 3: Contingency table of 3 separate classes X, Y and Z. Here, precision has been chosen as the relevant metric.

	TP	FP	FN	$precision$
Class X	TP_X	FP_X	FN_X	$precision_X$
Class Y	TP_Y	FP_Y	FN_Y	$precision_Y$
Class Z	TP_Z	FP_Z	FN_Z	$precision_Z$

To calculate the precision_{macro}, we simply compute the precision for each row in the contingency table and then average the per-class precision scores in order to compute a global mean. This process is analogous for recall as well. Consequently, the precision_{macro} and recall_{macro} are calculated according to Equation 48 and 50 respectively [27].

$$precision_{macro} = \frac{precision_1 + precision_2, \ldots, precision_n}{n}, \tag{48}$$

$$recall_{macro} = \frac{recall_1 + recall_2, \ldots, recall_n}{n}. \tag{49}$$

The macro average F_1-score is then calculated as previously shown but based on the precision_{macro} and recall_{macro} [27]:

$$F_{1macro} = 2 \times \frac{precision_{macro} \times recall_{macro}}{precision_{macro} + recall_{macro}}. \tag{50}$$

7 Data

Previous researchers in the field of convolutional graph neural networks have seldom had access to more than a few thousand invoices with labeled bounding boxes [5], [3]. For this thesis, 83 672 unique Swedish invoices from 18 272 vendors have been acquired through a third party. These vendors do not necessarily provide their own unique invoice templates, but it is highly likely that this is the case. This thesis will for the sake of simplicity regard the 18 272 different vendors to be equal to 18 272 different templates.

The data from these invoices are divided into two datasets: The first dataset, which will henceforth be denoted as the *OCR dataset* or *OCR data*, is the raw data sourced directly from the output of the OCR engine Google Vision, containing information such as the text content of a word, the bounding box of said word (i.e., the pixel position) and the page on which the word occurs. The second dataset, subsequently called the *label dataset* or *label data*, also consists of bounding boxes, page numbers and with the addition that the text value contained by the bounding box has been parsed and standardized. The label dataset only contains information about words belonging to the entity classes of interest. These entity classes are further elaborated upon in section 7.1.

Section 7.1 describes the structure of the raw data that has already been acquired, i.e., the OCR data and the label data. In Section 7.2 we review the cleaning process of the two datasets and how the two cleaned datasets are merged to create a single annotated dataset. Section 7.3 presents how the annotated dataset is used to model the data into a graph representation of an invoice through the form of an adjacency matrix. Section 7.4 describes how the annotated dataset is split into train-, test- and validation datasets, as well as some insights acquired from the train dataset.

7.1 Data Sources

The raw, unlabeled, OCR data is sourced from the OCR engine Google Vision and is a set of JSON-files, where each JSON-file corresponds to a single invoice and contains a set of fields containing data corresponding to the words present in the invoice. Each field consists of bounding box coordinates for a given word, the page of where the bounding box is present, which invoice ID the bounding box belongs to as well as the text content within the bounding box. A bounding box is defined by the x and y coordinate of its top left corner of the box, as well as its width and height. The coordinates are measured by their pixel position on the invoice image. Figure 12 shows what the bounding boxes look like overlaid on a dummy invoice, i.e., what the OCR engine has extracted from the invoice.

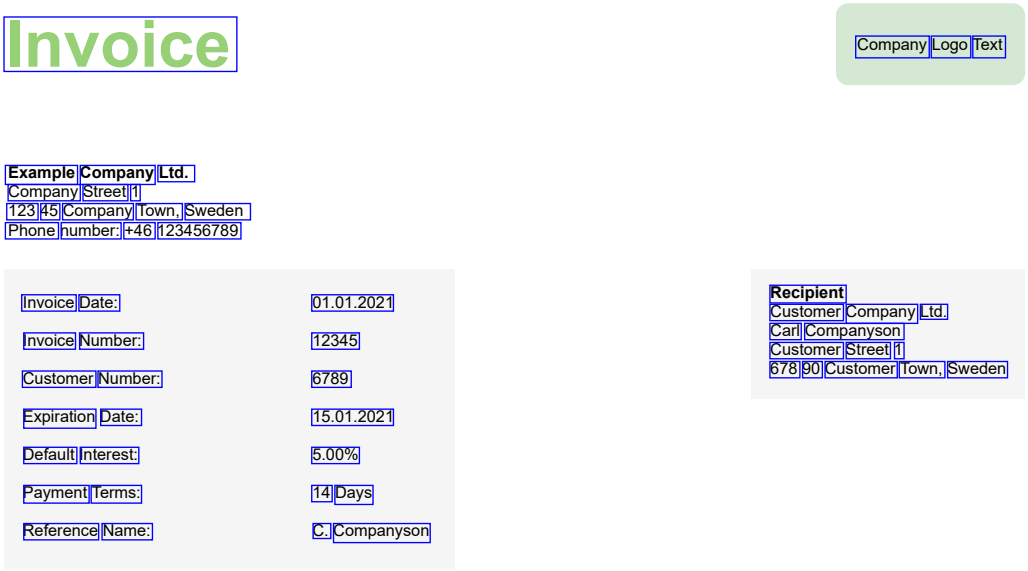


Figure 12: The extracted words and corresponding bounding boxes for each text value overlaid on a dummy invoice. The invoice in the figure is in English for exemplification purposes, while the invoices in the real dataset are in Swedish.

The label data also consists of a set of JSON-files with a similar structure as the OCR dataset, where each JSON-file contains information about a single invoice. In the label dataset, a single field consists of a bounding box, a semi-automatically annotated label (annotated through a combination of legacy entity extraction models and human supervision by a third party) corresponding to the bounding box, the page number for which the bounding box exists and a formatted value of the text content within the bounding box. The formatting rules for the value differ depending on the label. In the scope of this project, only the label and the bounding box coordinates (x and y coordinates as well as height and width) are utilized when training and evaluating the model, the formatted value is only used when cleaning the dataset. An overview of the different labels used by our model and their expected text content can be found in Table 4.

Table 4: An overview of the label data. Label is the name of the class, Explanation explains the purpose of the entity and the expected value/format denotes the expected data type or text content.

Label	Explanation	Expected Value/Format
amountCurrency	Currency for the amount	A string, eg “sek”, “kr”, “kronor”
amountFreightPack	Shipping cost	An integer or a float
amountNet	Total net amount	An integer or a float
amountRndDiff	Amount to be rounded off from total amount	An integer or a float
amountTot	Total amount incl. net and VAT amount	An integer or a float
amountVat	Total VAT amount	An integer or a float
dueDate	Date of invoice expiration	Any representation of a date, eg., 2007/09/13, 13-09-18, “den 19 februari 2021”
invDate	Date of invoice creation	see *dueDate*
invNo	Number/ID of invoice	A sequence of digits and/or letters and special characters
ocrNo	Payment reference	An integer
orderNo	Order number	An integer
referenceName	A name for the invoice reference	A name/an alphanumeric code
type	The type of the document	A string, e.g., “Faktura”, “Kreditfaktura”, “Påminnelse”

An example of label data for a dummy invoice can be found in Figure 13, where the fields have been drawn on the corresponding invoice. The text in orange is the label, i.e., its class, the text contained by the bounding box is the unparsed value of the field.

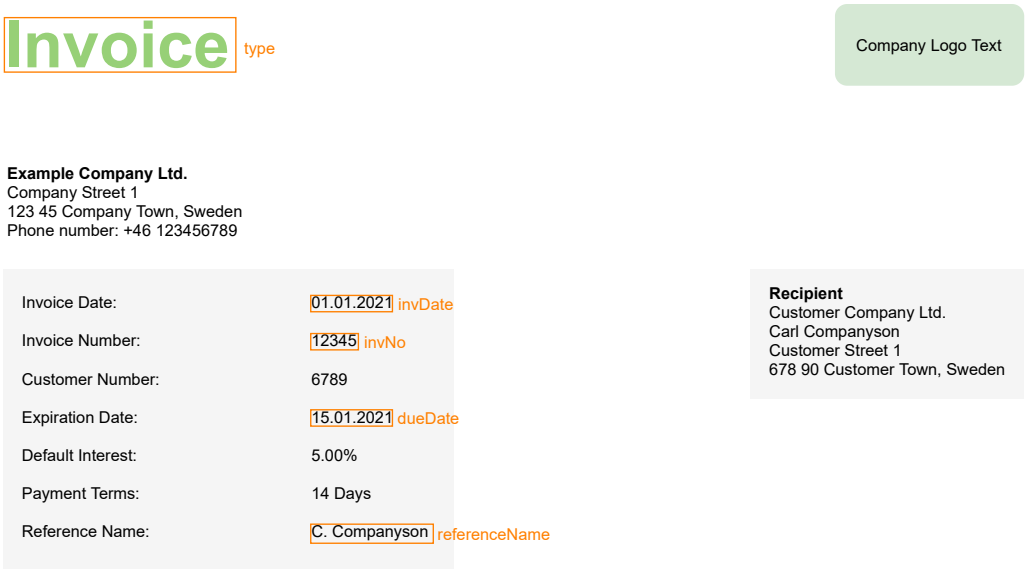


Figure 13: The label data overlaid on the dummy invoice.

7.2 Data Preprocessing

The following section will describe the data preprocessing steps required to prepare the *label data* and *OCR data* before feeding the data to the models. Section 7.2.1 will describe how the two datasets are cleaned and Section 7.2.2 describes how the two datasets are merged into a single dataset.

7.2.1 Data Cleaning

The process of cleaning both the label data and the OCR data is done much in the same way for both datasets. First, the data cleaning pipeline will be presented in chronological order, followed by a motivation of the different steps within the pipeline. All steps are applied on both datasets unless it is explicitly stated otherwise.

1. **Remove invoices with badly structured JSON data.** There is a clear structure within the vast majority of the OCR:ed JSON data but some invoices do not follow this convention. Removing those invoices makes the total number of invoices present in the dataset to 82 668.
2. **Remove small bounding boxes.** Bounding boxes which are only a few pixels high or wide exist in both the OCR data and the label data. These bounding boxes are often either artifacts produced from random noise interpreted as text in the OCR data or, in the case of the label dataset, a product of human error where labels which are non-existent in a specific invoice still have been registered into the dataset. Hence, all data points in both datasets, which have either a height or a width that is less than 0.2 % of the page height or the page width, are removed.
3. **Remove bounding boxes close to the document border.** Most of the valuable information of an invoice can be found within a certain area with some margin to the page border. At the same time, a lot of artifacts produced by the OCR engine occur near said borders. Faulty data points from the label dataset which contains wrongful information are also often found near the borders or in the corners of an invoice. Hence, only data points which have an x-value within the range $5\% < x < 95\%$ of the page width and have a y-value within the range $5\% < y < 95\%$ of the page height are kept.
4. **Lowercase text content** (Only for OCR data). To make sure that the casing of letters do not affect future feature calculations such as word embeddings, all text fields are lowercased.

5. **Strip special characters** (Only for OCR data). The OCR engine often interprets noise, such as ink blots or speckles on an invoice, as either of the special characters "* ! ;". Leading or trailing special characters are hence stripped from the text value of each data point.
6. **Remove data points with empty text content.** To make sure that there are no data points from the label datasets which have empty value-fields, said data points are removed. After step 5 in the data cleaning pipeline, some data points in the OCR data may also have empty text fields. These data points are removed as well.

7.2.2 Constructing the Annotated Dataset

As the OCR dataset does not contain any labels and the label dataset does not contain any information about unlabeled words, we need to merge the two datasets in order to create a dataset which is useful for training a model. This merged dataset will henceforth be called the *annotated dataset*. The annotated dataset is created by, for each invoice, extracting the labels and the bounding box coordinates for each row in the labeled dataset and then finding the corresponding bounding box or bounding boxes in the OCR dataset. If any number of bounding boxes in the OCR dataset overlap a bounding box in the label dataset by a certain percentage (in this case 30 %), the bounding boxes in the OCR dataset are considered to belong to the corresponding label from the label dataset. However, as the two datasets contain images in both PDF (Portable Document Format) and TIF (Tagged Image File Format) formats with different resolutions (200 and 300 dots per inch, DPI, respectively), looking at the overlap ratio between the bounding boxes from the two datasets becomes redundant since they operate in different scales in the x and y coordinates. To mitigate this problem, the x and y coordinates are normalized based on the height and width of the invoice image, read at the corresponding resolution for each dataset. By using normalized coordinates instead of pixel positions relative to the DPI resolution, the problem of mismatching scales is omitted. If a bounding box in the OCR data does not have a corresponding label from the label data associated to it, its label is set to *undefined*. Figure 14 illustrates all the bounding boxes and their corresponding labels overlaid on a dummy invoice. As the figure shows, the entity *referenceName* is divided into two bounding boxes, as a result from the OCR extraction, and thus both bounding boxes receive the label *referenceName*.

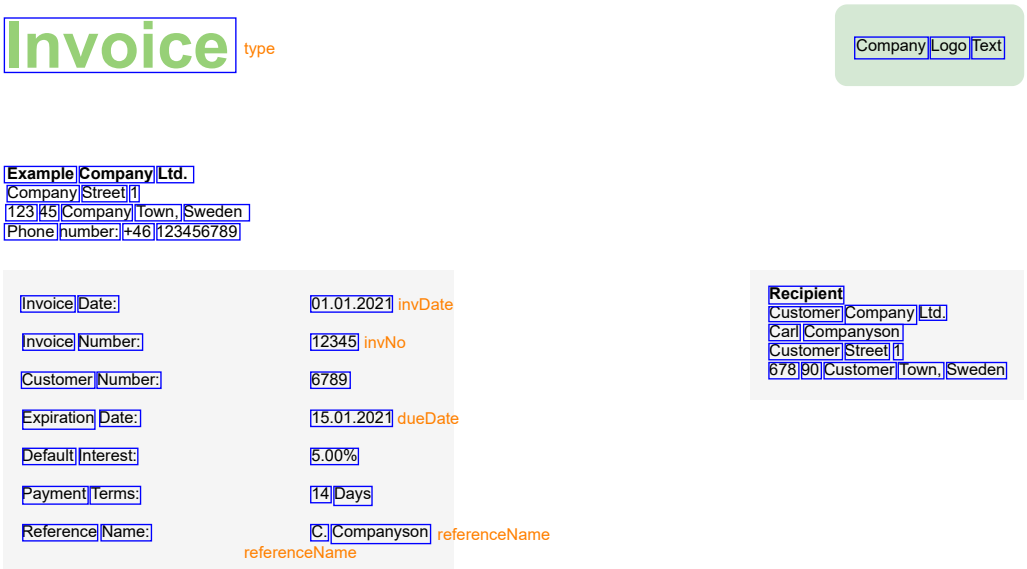


Figure 14: The annotated dataset overlaid on the dummy invoice. Each bounding box has a corresponding label, such as *invDate* or *dueDate*. The labels of bounding boxes labeled as *undefined* have been removed from this image in order to avoid occlusion, in actuality these labels would be present.

7.3 Graph Modelling

After cleaning the data and merging the two datasets of label data and OCR data into a single annotated dataset, the data needs to be modeled into graphs, where each word is represented as a node and each invoice corresponds to a single graph. The graph structure chosen for this thesis is a structure similar to the one proposed by [3]. To enforce that an invoice is read from left to right and top to bottom, we first construct lines of words. Algorithm 1 defines how lines are formed and is an extension of the line formation algorithm presented by [3].

Algorithm 1 Line Formation

1: Sort words based on Top coordinate and page number 2: Form lines as group of words which obeys the following:

Two words (W_a and W_b) are in the same line if:

W_a is on the same page as W_b AND
W_a is not in already in a line AND
W_b is not already in a line AND
$Top(W_a) \leq Bottom(W_b)$ AND
$Bottom(W_a) \geq Top(W_b)$

3: Sort words in each line based on $Left$ coordinate

To decrease ambiguity where a node can appear in many lines and in turn the need for processing the same node multiple times, we have added the constraint that a word can only belong to a single line. Since [3] only works with single paged invoices, we have also added the constraint that two nodes need to be on the same page to be able to belong to the same line. When building the graph, each line is processed from top to bottom, starting with the left-most word in each line, simulating a similar processing flow of the data as humans have when interpreting invoices written in many Western languages.

An undirected graph representation of an invoice is defined as $\boldsymbol{G}_{invoice} = (\boldsymbol{V}, \boldsymbol{E})$, where each vertex $v_i \in \boldsymbol{V}$ is a representation of a word W_i and each edge $e_{ij} = (v_i, v_j) \in \boldsymbol{E}$ describes a nearest neighbor relationship between two nodes, where edges are constructed through Algorithm 2.

Algorithm 2 Graph Modeling

1: Read words from each line starting from the topmost line going towards the bottommost line
2: For each word W_{source}, perform the following:

2.1: Check words that are on the same page, to the right and in vertical projection of W_{source}
2.2: Calculate the Horizontal Relative Distance, HRD, between W_{source} and the target words based on W_{source} rightmost coordinate and the target words leftmost coordinates
2.3: Select the word W_{target} with the shortest HRD as nearest $Right$ neighbor to W_{source}, provided that W_{target} does not already have a $Left$ neighbor at a closer or equal HRD and that $W_{source} \neq W_{target}$

2.3.1: In case W_{target} already has a Left neighbor W_{left} at a further HRD than to W_{source}, remove the neighbor relationship between W_{target} and W_{left}
2.3.2: In case W_{target} already has a $Left$ neighbor at a closer or equal distance to W_{source}, iterate over all target words, sorted in ascending order with respect to the HRD, until the criterion in 2.3 is met or until there are no more available nodes.

2.4 Repeat steps from 2.1 to 2.3 similarly for retrieving nearest neighbor words in the vertical direction by taking the horizontal projection of words on the same page as W_{source} as well as being below W_{source}, calculating the Vertical Relative Distance, VRD, and choosing words being further to the left in case of ambiguity
2.5 Draw edges between the source word and its 4 nearest neighbors if they are available

3: Connect the last word of a page (furthest down and to the right) with the first word (furthest up and to the left) on the next page by setting the last word as the Top neighbor to the first word on the next page

Since $\boldsymbol{G}_{invoice}$ is undirected, any time W_{target} is set as the $Bottom$ or $Right$ neighbor of word W_{source}, W_{source} is also set as the Top or $Left$ neighbor of W_{target}, respectively. As a given node is restricted to

having at most one neighbor in each of the major directions (i.e., top, bottom, left, right), each node has at most 4 neighbors, resulting in a sparse graph.

The criterion for a node being in vertical projection of W_{source} is defined as follows:

1. $Left(W_{source}) < Right(W_{target})$ AND
2. $Top(W_{source}) \leq Bottom(W_{target})$ AND
3. $Bottom(W_{source}) \geq Top(W_{target})$

Analogously, the criterion for a node being in horizontal projection of W_{source} is defined as:

1. $Bottom(W_{source}) < Top(W_{target})$ AND
2. $Left(W_{source}) \leq Right(W_{target})$ AND
3. $Right(W_{source}) \geq Left(W_{target})$

An illustration of the vertical and horizontal projection of a given source node can be seen in Figure 15.

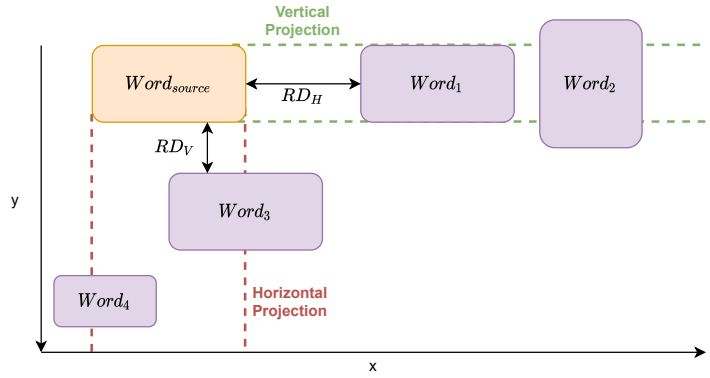


Figure 15: An illustration of possible *Right* neighbors ($Word_1$ and $Word_2$), as well as *Bottom* neighbors ($Word_3$ and $Word_4$) for $Word_{source}$. Here, RD_V and RD_H denote the Vertical Relative Distance and Horizontal Relative Distance, respectively.

The relative distance between two nodes are calculated as follows:

$$RD_H = Left(W_{target}) - Right(W source)$$
$$RD_V = Top(W_{target}) - Bottom(W_{source}).$$

The result is an adjacency matrix fully describing the structure of the graph. A visualization of how the nodes in an invoice graph representation are connected to their nearest neighbors can be found in Figure 16. A further demonstration of how different pages in an invoice graph are connected to each other can be found in Figure 17.

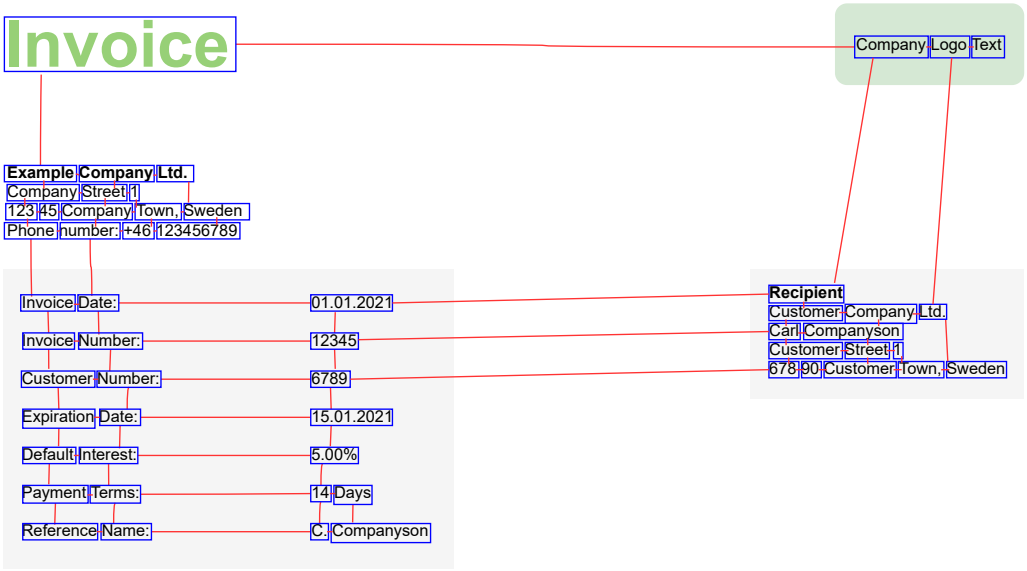


Figure 16: The resulting graph after applying the Algorithm 2 on a dummy invoice.

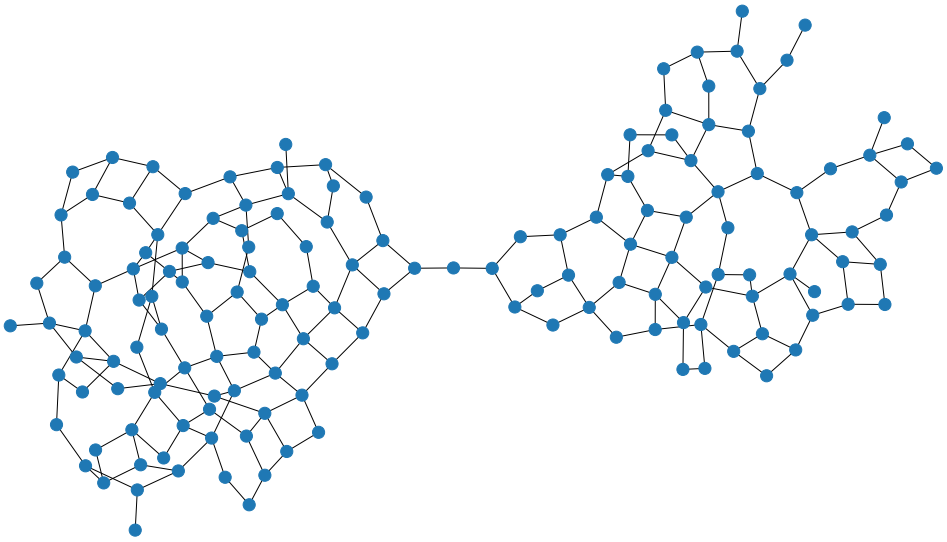


Figure 17: Graph representation of the nodes and edges present in a two-paged graph. The "waist" present in the middle of the two large clusters indicates a connection between two pages.

Although this graph structure is heavily inspired by Lohani et al. [3], there are a few key differences in our approach. Firstly, our approach allows for multi-page invoices of any size, while [3] assumes that each invoice only contains a single page. This is an important addition, as multiple pages and collective invoices are a common occurrence in the business world. Secondly, [3] places much more emphasis on the importance of top/left coordinate priority when creating edges. e.g., if a node in line l finds a nearest

bottom neighbor, then that edge is set and fixed throughout the graph modelling process even though there may be other nodes on subsequent lines that are substantially closer. Our approach relaxes this criteria and only enforces top/left priority in case of ambiguity, resulting in less edges between nodes which are not spatially close to each other. A comparison between the approach from [3] and our extension can be found in Figure 18.

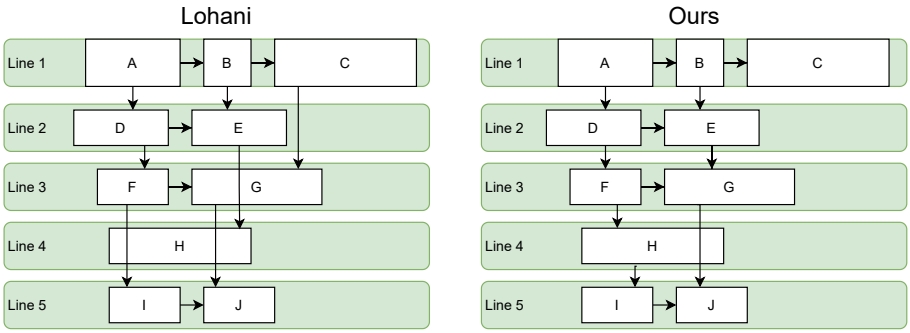


Figure 18: The left-most graph illustrates how nodes may be connected using [3]'s approach, while the right-most graph illustrates how nodes may connect using our extension. The figure illustrates the difference in vertical connections between nodes, however, the same differences can be found for horizontal edges as well. Note that the direction of the arrows only indicates source- and target nodes (i.e., arrows goes from the source node to the target node) and does not indicate any directionality of the edge.

7.4 Train, Test & Validation Datasets

After we have modelled the graph and calculated features associated with each node we can split the data into train, test and validation datasets. We perform a split into a ratio of 90%, 5% and 5% respectively, where we shuffle the data randomly based on invoice ID:s, thus keeping entire invoices intact within the dataset. From the split, we can gather insights about our training dataset, which we can see in Figures 19 and 20.

Figure 19 illustrates the relationship between all occurring 14 classes of interest. If we exclude the undefined class from the diagram, we gain a better understanding of how many data points of interest belonging to the different classes and how they stand in proportion to each other. Figure 20 demonstrates that the *amountTot* class stands for the majority of data points compared to the other 12 classes. The classes *amountNet*, *dueDate*, *invDate*, *amountVat*, *invNo* and *type* all have a somewhat equal amount of occurrences. The rest of the classes, *amountRndDiff*, *orderNo*, *referenceName* and *amountFreightPack* in particular, do not occur as often as the rest. In fact, the total number of occurrences of these "minor" classes combined are roughly as many as the *type* class.

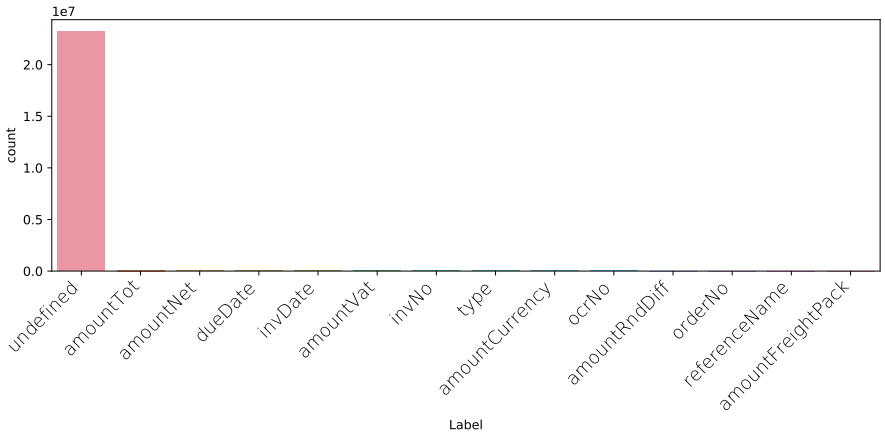


Figure 19: A bar plot illustrating the number of nodes belonging to each class in the training dataset.

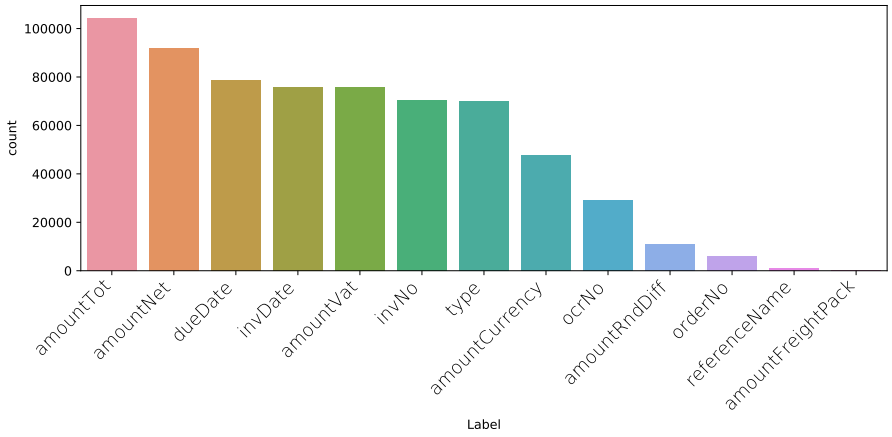


Figure 20: A bar plot illustrating the number of nodes belonging to each class in the training dataset. The label *undefined* has been omitted.

From these insights, it becomes evident that the class undefined alone stands for the vast majority of all data points. The class *undefined* occurs roughly 23.20 million times in a dataset containing around 23.86 million data points, which amounts to approximately 97%. Table 5 shows the ratio of all occurring classes within the training dataset.

Table 5: The ratio of occurrence between the classes, within the training dataset. In descending order.

Label	Ratio
undefined	97.23 %
amountTot	0.44 %
amountNet	0.39 %
dueDate	0.33 %
invDate	0.32 %
amountVat	0.32 %
invNo	0.30 %
type	0.30 %
amountCurrency	0.20 %
ocrNo	0.12 %
amountRndDiff	0.045 %
orderNo	0.026 %
referenceName	0.0045 %
amountFreightPack	0.0017 %

After the train, test and validation split we can gather further insights into the distribution of vendors between the different sets. Table 6 describes how many unique vendors to each are present in each respective set.

Table 6: Details on vendor occurrence between the dataset splits.

	Number of invoices	Number of unique vendors	Number of vendors unique to split
Training set	74 396	17 289	13 504
Validation set	4136	2719	394
Test set	4136	2691	386

By analyzing how many unseen vendors' invoices are present in the test and validation set respectively, we gather that roughly 15 % of all vendors in both the test and validation set are not present in the train

dataset. If we look at vendors not present in both the train and validation set, only in the test set, this number goes down to around 14 %, which corresponds to 393 invoices.

7.5 Discrepancies in the Data

By manually inspecting invoices, errors in the underlying data become apparent which may result in poor performance for all models. A common occurrence among many of the classes, although especially prominent for entities regarding amounts and dates, is that the values of said entities occur in multiple bounding boxes in an invoice where only one of the bounding boxes is labeled as the correct entity. The rest, even though they hold the same information, have a ground truth label of *undefined*. An example of this is Figure 21, where the total amount is found in multiple places on the same page of an invoice, but only one of the bounding boxes is labeled as *amountTot*. When inspecting multiple invoices from the same vendor, it also becomes apparent that there is no consistency in which fields that correspond to the ground truth. i.e., the node which corresponds to the ground truth node (and in extension also its neighborhood) varies greatly between different invoices, even when the invoices are from the same vendor. See Figure 21 as an example.

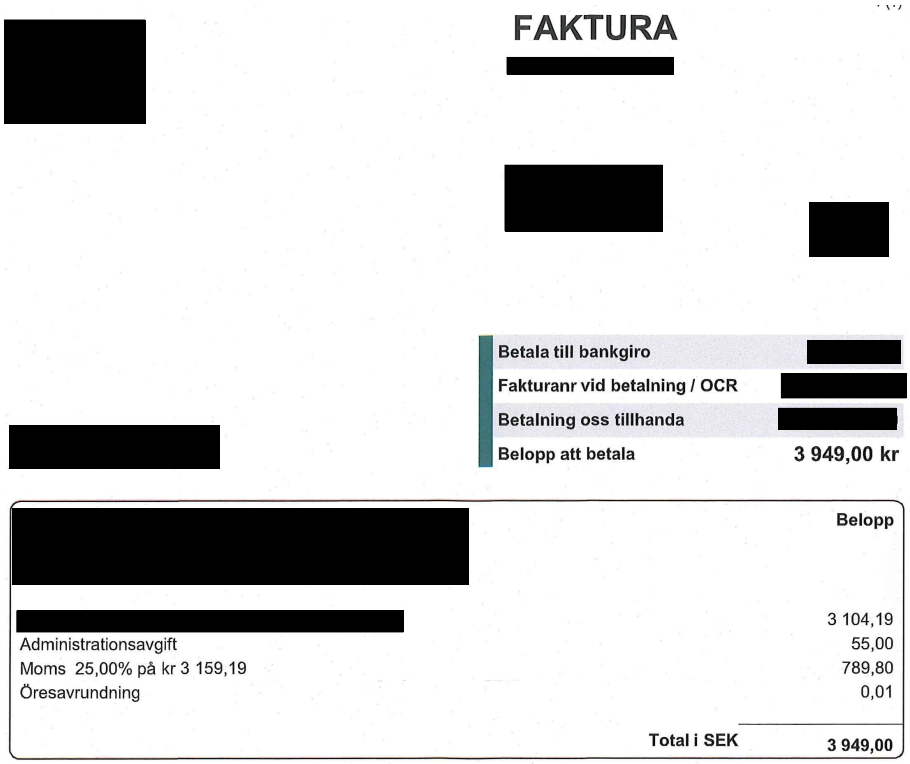


Figure 21: Example of an invoice where the total amount (i.e., "3 949,00") appears twice on the same page. Depending on the invoice, any of these two fields could be labeled *amountTot*, while the other is labeled as *undefined*.

Another problem with the underlying label data is that there are format inconsistencies and wrongly annotated nodes. This is a product of: bounding boxes from the label data sometimes overlapping more OCR bounding boxes than they should (see 22), label data and OCR data sometimes have mismatching scales (see Figure 23) or label data simply being wrong due to human error during the creation of the label dataset (see Figure 24).

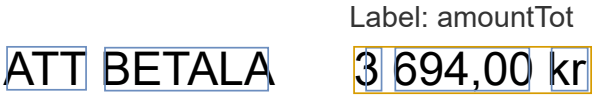


Figure 22: The bounding box in the label data (orange) for the *amountTot* entity on this invoice spans the three boxes containing "3", "694,00" and "kr", although the last box should be labeled as *amountCurrency* or perhaps *undefined*.

Figure 23: Because the scale of the OCR data (blue bounding boxes) and label data (orange bounding box) does not match, both the due date and the invoice number gets annotated as *invNo* during the creation of the annotated dataset.

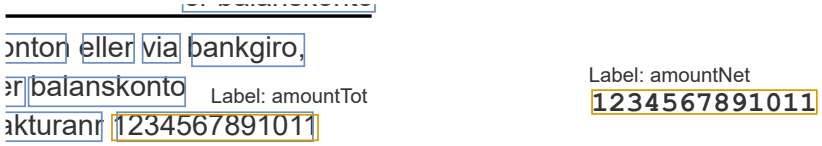


Figure 24: Extracted entities from different locations within an invoice. The OCR number is wrongly labeled as *amountTot* (left) and *amountNet* (right) because of errors existing in the label data.

There are also errors in the data stemming from the OCR engine when extracting the bounding boxes of words, which give rise to spurious nodes in the graph representation of an invoice. For some words, multiple bounding boxes of various sizes are spawned. This results in multiple overlapping bounding boxes which all attempt to represent the same word, see Figure 25. This gives an untrue graph representation of the invoice and because of the algorithm we use to annotate the OCR data in section 7.2.2, noise in the patterns for recognizing a certain label is introduced.

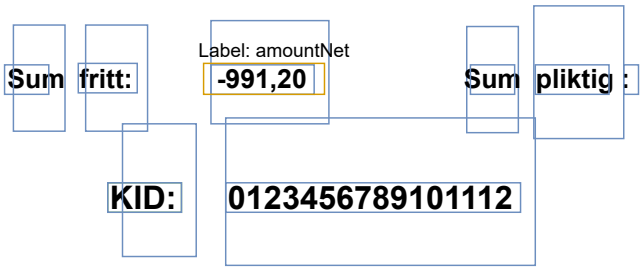


Figure 25: An example of when the OCR engine has produced multiple bounding boxes for each word in a given section. Because of how the algorithm for creating the annotated dataset is constructed, both of the blue bounding boxes surrounding "-991,20" will be labeled as *amountNet*.

There are also some instances where two entities are confused with each other in the label dataset, see Figure 26. Here, the label bounding box which should encapsulate the OCR bounding boxes belonging to the *amountTot* entity instead encapsulates the *amountNet* OCR bounding boxes. The opposite is true for the *amountNet* entity and will in this case be labeled as *amountTot*.

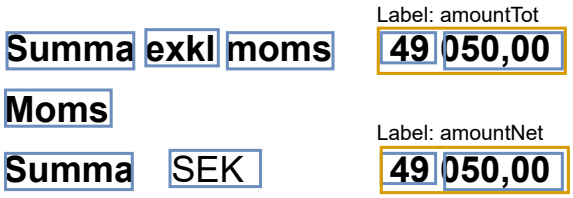


Figure 26: The bounding boxes for the labels *amountTot* (top orange bounding box) and *amountNet* (bottom orange bounding box) have been swapped in the label data.

Tabel 7 shows the result of manually inspecting 100 invoices and noting the occurrences for each discrepancy. The table shows the number of invoices these discrepancies are present in, not the number of discrepancies.

Table 7: Frequency of discrepancies on 100 manually inspected invoices.

Type of discrepancy	Number of invoices where the discrepancy is present
Entity is located on several places in one invoice and labeled once (see Figure 21)	73
Overlap (see Figure 22)	4
Entity is present in invoice but labeled as *undefined*, regardless of frequency of occurrence	78
Label data and OCR data are incorrectly scaled (see Figure 23)	1
The OCR-engine has produced several bounding boxes for the same node (see Figure 25)	16
Entity is incorrectly labeled as another entity, with the exception of undefined (see Figure 26)	12

The inspection has only been done on 100 invoices, i.e., a small fraction of the dataset, but the invoices have been randomly selected and provides us with an idea of the frequency of occurring errors within the data. These prevalent discrepancies can very well occur often enough to penalize our model's performance.

8 Feature Calculation

This section describes what features are associated with each node and how they are calculated. Section 8.1 describes the use of Byte Pair Encoding and Byte Pair Embedding to calculate features, Section 8.2 describes all node features chosen for the nodes in the graph to be fed to the model.

8.1 Byte Pair Encoding & Word Embeddings

In order to produce useful features for the text values present in an invoice, we need to represent said values in a useful manner. Word embedding is a word representation that allows for words with similar meaning to be similarly represented by preserving the syntactic and semantic similarities through projections in a continuous space of words. The individual words are represented as real-valued vectors in a predefined vector space, where each word is mapped to one vector and the corresponding vector values are learned in such a way that it resembles a neural network [28].

8.1.1 Byte Pair Encoding

For this thesis, the data compression algorithm *Byte Pair Encoding* (BPE) is used together with GloVe, in order to create the word embeddings used as features for the models (the utilization of GloVe together with BPE is explained in Section 8.1.2).

BPE compresses data through an iterative process by replacing the most frequently occurring adjacent byte pairs with bytes not present in the original data. The process ends when there are no more pairs occurring more than once [29]. For instance, let us say we have the text string `KHJERUKHJHJ`. The pair `HJ` is the most frequently occurring one and will be replaced by a byte not present within it, e.g., `A`. The resulting text string will be `KAERUKAA`, with the most frequently occurring byte pair `KA`. `KA` will be replaced by another unseen byte, e.g., `B`, which yields the string `BERUBA`. The resulting text string does not have any more frequently occurring byte pairs, which means that it cannot be compressed any further and thus the process ends. Decompression of the data can be done by performing the steps in reversed order [29]. The process is illustrated as follows:

Original data:	`KHJERUKHJHJ`
First iteration - swap `HJ` to `A`:	`KAERUKAA`
Second iteration - swap `KA` to `B`:	`BERUBA`

.

8.1.2 Byte Pair Embedding & GloVe

A utilization of BPE is through *Byte Pair Embedding* (BPEmb) which is based on the BPE algorithm and is a collection of pre-trained sub-word embeddings in 275 languages, trained on the contents of the website Wikipedia. More specifically, BPEmb is the combination of applying BPE tokenization and the *Global Vectors* method (GloVe) in order to create said subword embeddings [30]. In the context of this thesis, BPEmb is highly useful when coupled with OCR:ed information from invoices, since it is able to deduce the meaning of words correctly even if the words are incorrectly read by the OCR engine, due to the subwords present.

GloVe is a method used to obtain vector representations of words through unsupervised learning on global corpus statistics of word-word co-occurrences. GloVe is built upon the idea of being able to derive semantic relationships between words' co-occurrences, i.e., the probabilities of words co-occurring in a given corpus [31].

Deriving the word-word co-occurrences starts with constructing the so-called co-occurrence matrix $\boldsymbol{X}$, where $\boldsymbol{X}_{ij}$ denote the number of times the word j co-occurs with the word i. For instance, given the sentence *Steam is a gas. Ice is a solid.* the co-occurrence matrix can be constructed as follows [31]:

$$
\begin{array}{r} \\ \textit{Steam} \\ \textit{is} \\ \textit{a} \\ \textit{gas} \\ \textit{Ice} \\ \textit{solid} \\ . \end{array}
\begin{array}{c}
\begin{array}{ccccccc} \textit{Steam} & \textit{is} & \textit{a} & \textit{gas} & \textit{Ice} & \textit{solid} & . \end{array} \\
\begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 2 & 0 & 0 & 0 & 0 \\
0 & 2 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}
\end{array}
.
$$

Here, we see that, e.g., the word *is* co-occurs with the word *a* twice, and with the words *Steam* and *Ice* once. From the co-occurrence matrix we can derive the probability of the word j appearing in the context of the word i by denoting $P_{ij} = P(j|i) = X_{ij}/X_i$. If we let $i = ice$ and $j = steam$, the ratio of the word-word co-occurrence for these words can be analyzed by testing different probe words, denoted as k [31]. If a word k is related to the word *ice* but not to *steam* we get a smaller ratio than vice versa, i.e., a word k related to the word *steam* but not *ice*. Table 8 illustrates these co-occurrence probabilities for different k's.

Table 8: Probabilities for co-occurrence for the words $i = ice$ and $j = steam$. The choices of k have been selected from a large corpus. A higher value for $P(k|i)/P(k|j)$ indicates a stronger relation to the word k. Table and data derived from [31].

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

In short, the word embedding collection BPEmb is the result of applying BPE tokenization together with GloVe, and trained on the contents of Wikipedia. BPEmb allows for guessing the meaning of unknown or out-of-vocabulary words where BPE provides a subword segmentation without requiring tokenization.

8.2 Node Features

After the data has been modelled into a graph, each node in the graph will be associated with a certain number of calculated features, each of which are described in Table 9.

Table 9: The features associated with each node.

Feature	Comment
isAlpha	Boolean describing if the text is alphabetical or not.
isAlphaNumeric	Boolean describing if the text is a alphanumeric
isCurrency	Boolean describing if the text is a currency or not
isDate	Boolean describing if the text can be parsed as a date or not
isNumericWithDecimal	Boolean describing if the text is a number with decimals or not
RD_B	Vertical relative distance to the bottom neighbor node, defaults to 1 if no neighbor is present
RD_L	Horizontal relative distance to the left neighbor node, defaults to 1 if no neighbor is present
RD_R	Horizontal relative distance to the right neighbor node, defaults to 1 if no neighbor is present
RD_T	Vertical relative distance to the top neighbor node, defaults to 1 if no neighbor is present
Word embeddings	A vector of length 300 containing embedded values of the word

The choice of features and the way they are calculated has been done very much like the way proposed by [3]. The calculations of the word embeddings differ slightly, however. When utilizing BPEmb, a single word can be segmented into at least 1 and up to n subwords, where $n \to \infty$ for large enough words. Each subword is then represented by an embedding vector of length k, where k is a design parameter and set to 300 in this thesis. This results in that a word consisting of only one subword would be represented by an embedding matrix of size $(1, 300)$, and that a word segmented into 3 subwords would produce an embedding matrix with the shape $(3, 300)$. To enforce a fixed dimension for the embedding features, the column-wise mean is used as the embedding representation of the word. The choice of $k = 300$ is based on the research done by [3]. Another key difference from the method implemented by [3] is the size of the vocabulary used from BPEmb and the language used, which is the size 200 000 and the Swedish language. In general, a large vocabulary size results in frequent words not being split into segmentations, which in this case means that it minimizes the need of mean pooling the embedding vector to form a dimension of 300.

9 Models

This section aims to describe the architecture of the different models used in the thesis. Section 9.1 describes the ChebNet model and section 9.2 describes the two GCN models. Each of the models presented are trained with the same hyperparameters, which are shown in Table 10. Moreover, GCN-4 and ChebNet are both retrained using DropEdge before feeding the adjacency matrix to the network. Multiclass Cross Entropy is used to calculate the cost for each batch.

Table 10: Hyperparameter values for the models.

Hyperparameter	Value
Batch size	600
Learning rate	$1 \cdot 10^{-3}$
No. of epochs	2000
Patience	50
Weight decay	$1 \cdot 10^{-5}$

A general representation of the model architecture can be found in 27. Note that the number of input features are consistent between models (i.e., $F = 310$), although the dimensions of the hidden feature embeddings as well as the L number of stacked ConvGNN layers may differ. The model is fed a graph consisting of N nodes, where N may differ between invoices, and is represented by its adjacency matrix $\boldsymbol{A}$ and the corresponding feature matrix $\boldsymbol{X}$. $\boldsymbol{A}$ and $\boldsymbol{X}$ are processed by L stacked graph convolution layers, each followed by a ReLU activation function. The first layer maps the F input features of each node to C output channels. Each consecutive convolutional layer then increases the number of output channels by a factor of 2, such that the number of parameters $\theta^{(l)}$ at the l^{th} layer follows the pattern:

$$size(\theta^{(l)}) = K \times 2^{l-2}C \times 2^{l-1}C. \tag{51}$$

Where K is the number of hops and C is the initial filter size of the first convolution layer. Note that for GCN-based models, K is implicitly set to 1. After convolving the node feature representations L times, The feature map $H^{(L)}$ is fed to a FC layer which maps each node feature vector $h_i^{(L)}$ to a logit vector $o_i \in \mathbb{R}^P$, where P is the number of labels. The FC layer is followed by a softmax activation function, resulting in an output vector where each index position of the output vector corresponds to a label (including *undefined*), and the index position with the row-wise highest logit value is the predicted label for a given node. The entire architecture is illustrated in Figure 27.

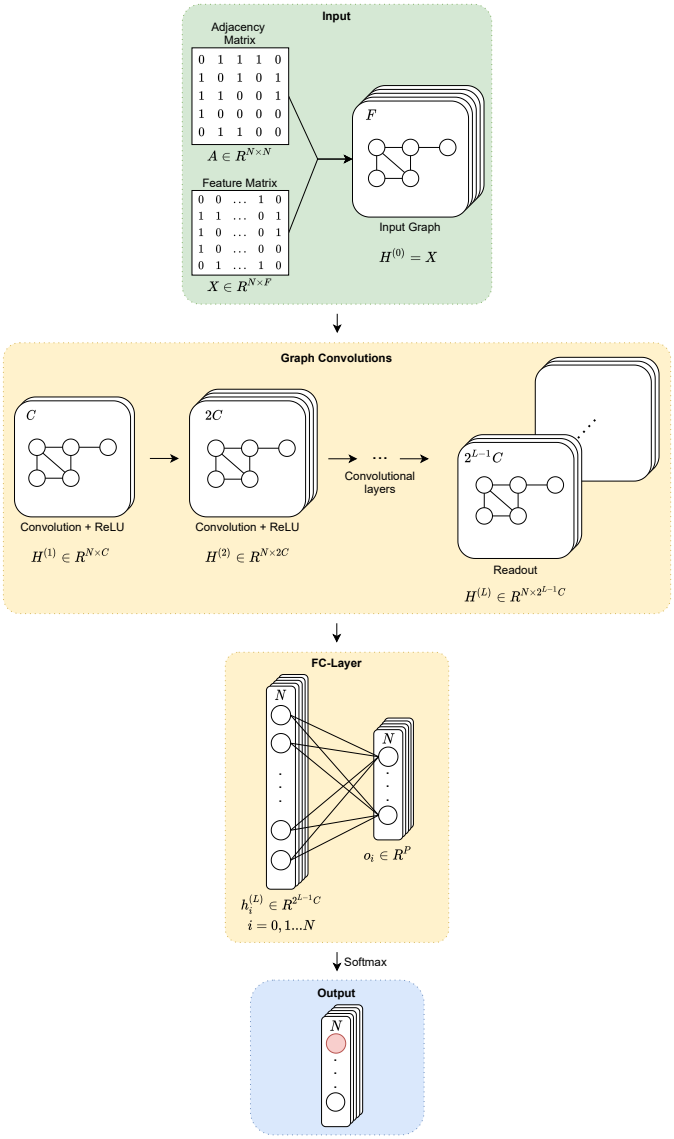


Figure 27: The general architecture of the models. A model is fed a graph representation of an invoice consisting of an adjacency matrix $\boldsymbol{A}$ and a feature map $\boldsymbol{X}$. The features of each node are then convolved L times and the resulting feature map is fed through a fully connected layer, followed by a softmax activation function. The resulting output is a matrix of size $\mathbb{R}^{N \times P}$ where P is the number of classes and the predicted class of each node is the class which corresponds to the index of the highest logit. The main difference between the models is the choice of convolution layers, as well as the choice for C and L.

9.1 ChebNet

The ChebNet model is the same model as proposed by [3] and consists of four ChebNet graph convolution layers (Section 4.2.2), where the Chebyshev filters' sizes increase with a factor of 2 per layer. The first layer has a filter size of 16, the second layer size 32 and so on, all the way up to a filter size of 128 in the fourth layer. We also set $K = 3$ in order to keep each convolution at most 2 steps away from the center vertex at each layer (recall from section 4.2.2 that the hyperparameter K determines the locality of the filter). Each layer is followed by a ReLU function. The fifth layer is a FC layer that reduces the dimension down from 128 to the number of classes, i.e. 14. Finally, a Softmax activation function provides us with a class label to each word in the invoice. Recall that we have 14 classes of interest, including the *undefined* class for irrelevant entities. The model has an added bias vector for each layer. For a parameter count of the model and its layers, please refer to Table 11.

Table 11: The parameter count for each layer as well as the total amount of trainable parameters in the ChebNet network.

Parameter	Layer 1	Layer 2	Layer 3	Layer 4	FC Layer
Weights	$3 \times 310 \times 16$	$3 \times 16 \times 32$	$3 \times 32 \times 64$	$3 \times 64 \times 128$	128×14
Bias	16	32	64	128	14
Total					49 182

9.2 GCN

Two different models have been built using a GCN based architecture. The first GCN model, henceforth called GCN-4, is very similar to the ChebNet model. GCN-4 consists of 4 GCN layers, where the size of each layer filter increases by a factor of 2 at every l^{th} layer, starting with a filter size of 16 and ending with the fourth layer with a filter of size 128. As in 9.1, each layer is also followed by a ReLU function. The fifth layer is a FC layer, reducing the number of dimensions from 128 down to the number of classes, i.e. 14. The FC layer is followed by a Softmax function.

As a benchmark model, as well as because of concerns regarding over-smoothing for deep GCN models [10], we have also built and trained a shallow GCN model, GCN-2. GCN-2 consists of two GCN layers with a filter size of 32 and 64, respectively, where each layer is followed by a ReLU function, as in previously mentioned models. The last layer is a FC layer which reduces the dimensions from 64 to 14 and is followed by a Softmax function.

Both models have a fixed $\lambda = 1$ as weight for the added self-loops. Each model has an added bias vector for each layer in the network. Please refer to Table 12 as well as Table 13 for an overview of the parameter count.

Table 12: The parameter count for each layer as well as the total amount of trainable parameters in the GCN-4 network.

Parameter	Layer 1	Layer 2	Layer 3	Layer 4	FC Layer
Weights	310×16	16×32	32×64	64×128	128×14
Bias	16	32	64	128	14
Total					17 758

Table 13: The parameter count for each layer as well as the total amount of trainable parameters in the GCN-2 network.

Parameter	Layer 1	Layer 2	FC Layer
Weights	310×32	32×64	64×14
Bias	32	64	14
Total			12 974

10 System Overview

This section aims to describe the different tools and frameworks used to build the entire system, as well as summarize the entire system from a high-level point of view.

10.1 Tools

Pandas [32] and Numpy [33] are two Python libraries used for processing tabular data and have been used extensively in this thesis for cleaning, preprocessing and merging the OCR and label data into the annotated dataset, as well as for feature calculations.

To model the tabular data in the annotated dataset into a graph structure, the library NetworkX [34] has been leveraged. NetworkX has been chosen for this task because of its high compatibility with Pandas, PyTorch [35] as well as Pytorch Geometric [23]. PyTorch is a tensor library which allows for deep learning and model training on both the CPU as well as the GPU. PyTorch Geometric is an extension library of PyTorch, aimed towards deep learning on irregular data such as graphs. Tabular data has been modelled into graphs through NetworkX, which in turn has been converted into its tensor representation through PyTorch Geometric to prepare the data for model training and GPU accelerated computing. PyTorch, in conjunction with PyTorch Geometric, has been used for building and training the neural network models. Scikit-Learn [36] has been used for calculating the metrics for model evaluation. Calculating the word embedding features, through BPEmb (8.1.2), has been done by implementing the BPEmb Python package, provided by the method's creators [30].

Monitoring and logging metrics and statistics from the training processes have been done with the use of TensorBoard [37]. While TensorBoard is technically a TensorFlow visualization toolkit, it works equally well for monitoring models made in Pytorch. A full list of the Python libraries with their respective versions used in this thesis can be found in the appendix.

Regarding hardware, processing and computation have been done on an NVIDIA GeForce RTX 2080 Ti (GPU) and an Intel Core i9-9820 3.30GHz (CPU).

10.2 System Data Flow

Sections 3 - 5 and 7 - 9 explain the main building blocks needed for each step in the extraction process. Figure 28 illustrates how these building blocks are connected to go from raw data to extracted entities of invoices. We have chosen to illustrate the data flow in two parts: the data flow during the training phase (where multiple invoices with labeled nodes are processed to train a model) , as well as the data flow during the production phase (where a single invoice with unlabeled nodes is processed by the model).

During the training phase, the OCR- and label datasets are first merged into a single annotated dataset (Section 7.2). The annotated dataset is then passed to the graph modeller (Section 7.3) as well as the feature calculator (Section 8) processes to model the invoices as graphs. Remember that a graph representation of a single invoice with N nodes consists of 1 adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ denoting the edges between nodes, 1 feature matrix $\boldsymbol{X} \in \mathbb{R}^{N \times F}$ where $x_i \in \boldsymbol{X}$ corresponds to the features of node i and 1 label vector $y \in \mathbb{R}^N$ where y_i is the label of node i. When all invoices have been modelled into graphs, the model is ready to be trained.

The *Batch Graph Generator* splits invoices from the train dataset into mini-batches of size 600, constructing one large graph representation out of 600 invoice graph representations (Section 5.4). These mini-batches are then sent, one mini-batch at a time, to the ConvGNN model, which produces a vector $\hat{y}$ of predicted labels for each node in the mini-batch. The Multiclass Cross Entropy Cost between $\hat{y}$ and label vector y is then calculated (Section 5.1) and used to update the parameters of the model through backpropagation (Section 5.3). The training stops when the model has not improved its performance on the validation set for 50 consecutive epochs.

The data in the production phase flows similarly to that in the training phase, with a few key differences. An invoice is modelled into a graph directly from the OCR data, as no label data is available. Hence, no label vector is produced during the feature calculation process. The invoice graph representation is then fed to the trained ConvGNN model, which in turn produces a vector $\hat{y}$ of predicted labels for each node in the invoice. Each node which is not labeled as undefined can be seen as an extracted entity.

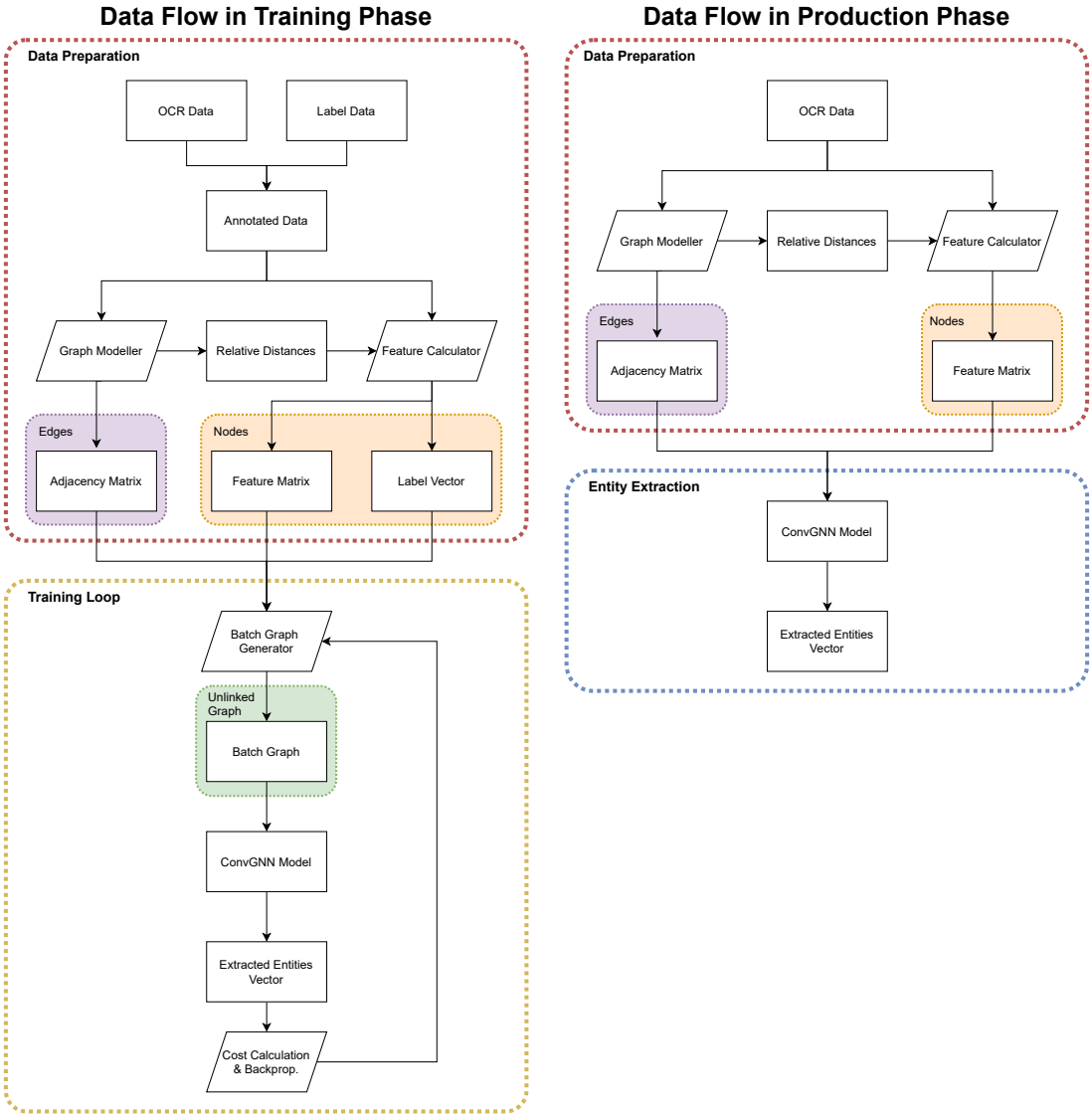


Figure 28: An overview of the complete system data flow during both the training and production phase. Rectangles indicate persistent storage, parallelograms indicate processes and arrows indicate the direction of the data flow. Coloured groups have been added to clarify how different stages of the pipeline relates to graph data representation of invoices.

11 Results

This section describes the results given by the models proposed in Section 9. For each model, the results present are the macro average for F_1-score, precision and recall, all evaluated on the test dataset. A full list of the F_1-score, precision and recall for each model and each individual label, as well as the confusion matrix for each respective model can be found in **Appendix A**. The F_{1macro}, precision_{macro} and recall_{macro} of each model can be found in Table 14. The confusion matrix for the best performing model can be found in Figure 29 as well as Figure 30 and the per-label metrics for the best model can be found in Table 15. The per-label performance for each model can be found in **Appendix A**.

Furthermore, this section will describe the results of the models on unseen invoice templates, see Table 16. The confusion matrix for the best performing model can be found in Figures 31 and 32. The per-label performance of the best performing model can be found in Table 17. That is, the model will be evaluated on invoices belonging to vendors which the model has not seen during training (as described in Section 7.4). The per-label performance on the unseen subset for each model can be found in **Appendix A**.

Table 14: Comparison of the macro average performance of the tested models. The highest metric scores have been highlighted in grey.

Model name	F_1	Precision	Recall
ChebNet	0.7119	0.8259	0.6255
ChebNet + DropEdge, $p = 0.10$	0.7088	0.8243	0.6217
ChebNet + DropEdge, $p = 0.20$	0.6709	0.7604	0.6003
ChebNet + DropEdge, $p = 0.50$	0.6253	0.7140	0.5563
ChebNet + DropEdge, $p = 0.80$	0.4963	0.6565	0.3989
GCN-2	0.5262	0.6354	0.4490
GCN-4	0.5309	0.6684	0.4404
GCN-4 + DropEdge, $p = 0.10$	0.5381	0.6514	0.4584
GCN-4 + DropEdge, $p = 0.20$	0.5069	0.6843	0.4025
GCN-4 + DropEdge, $p = 0.50$	0.3259	0.4932	0.2433
GCN-4 + DropEdge, $p = 0.80$	0.1095	0.2048	0.0747

Table 15: The corresponding performance metrics for each entity for the ChebNet model.

Entity	F_1	Precision	Recall
amountCurrency	0.7387	0.7918	0.6923
amountFreightPack	0.1250	1.0000	0.0667
amountNet	0.7360	0.7964	0.6842
amountRndDiff	0.3420	0.7333	0.2230
amountTot	0.8172	0.8258	0.8088
amountVat	0.8170	0.8348	0.7881
dueDate	0.8110	0.8171	0.8050
invDate	0.8681	0.8641	0.8723
invNo	0.8166	0.8565	0.7802
ocrNo	0.7108	0.7215	0.7003
orderNo	0.4696	0.8682	0.3218
referenceName	0.1639	0.5556	0.0962
type	0.9132	0.9040	0.9227
undefined	0.9948	0.9938	0.9958
Macro average	0.7119	0.8259	0.6255
Validation loss			0.0320

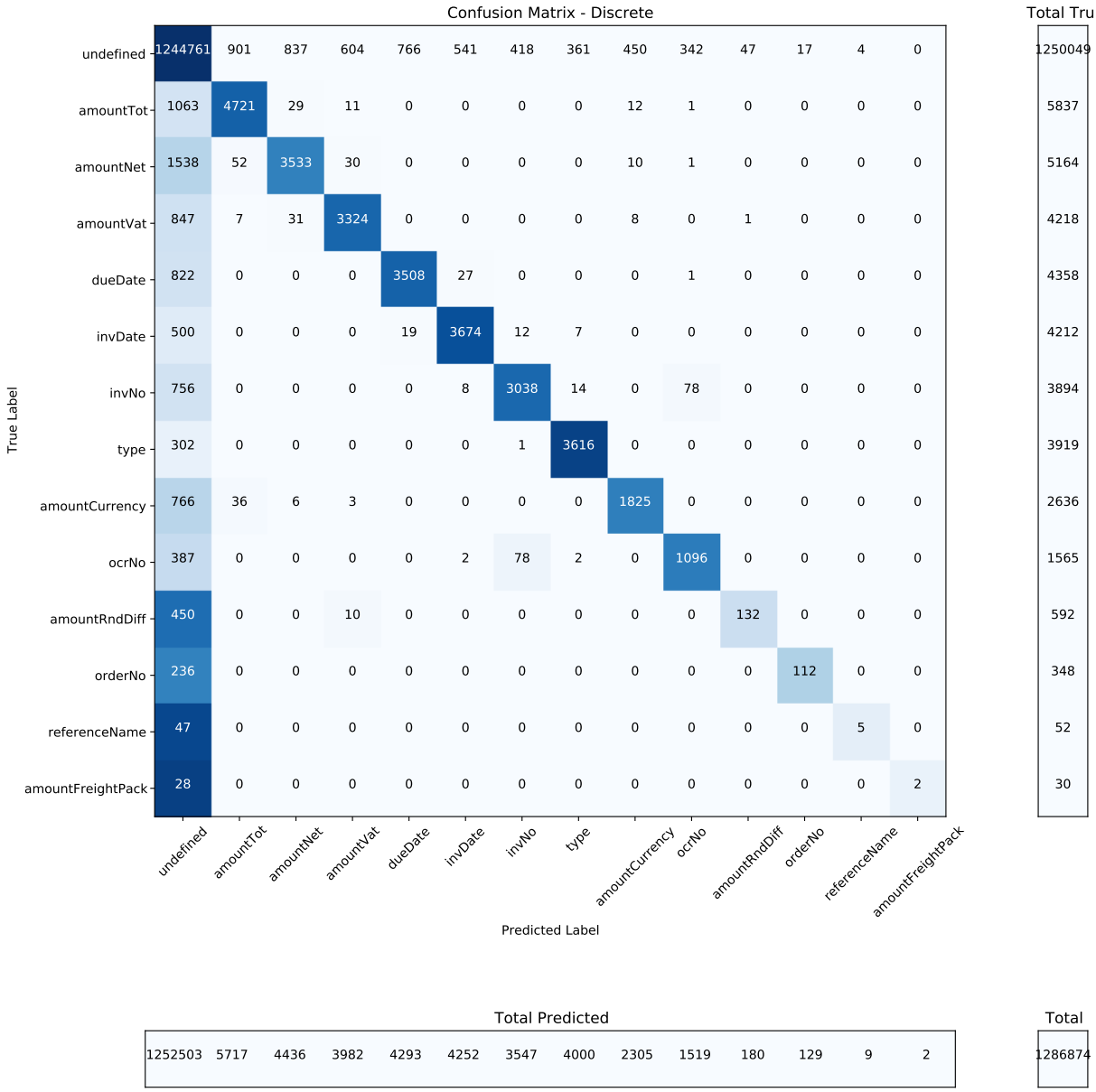


Figure 29: Discrete confusion matrix for the ChebNet model performance. The diagonal shows the per-class accuracy for each class. For instance, 4721 labels that were of the class *amountTot* have been classed as such. Meanwhile, 1063 labels that belong to the class *amountTot* have been classed as *undefined*.

Figure 30: Normalized confusion matrix of the ChebNet model, evaluated on the test set. The diagonal shows the per-class accuracy for each class where decimals are rounded to the nearest hundredth. For instance, 81 % of the labels that were of the class *amountTot* have been classed as such. Meanwhile, 18 % of the labels that belong to the class *amountTot* have been classed as *undefined*. The last 1% is distributed among other labels as can be seen in Figure 29.

Section 7.4 illustrates how approximately 9% of the test dataset contains invoices with unseen templates. The performance of the different models on the subset of invoices with unseen templates in the test set can be found in Table 16. Table 17 and Figures 31 and 32 show the best model's performance on these invoices alone. A comparison of the best performing model's performance on the complete dataset versus the performance on invoices with only unseen templates can be found in Table 18.

Table 16: Comparison of the macro average performance of the tested models on unseen templates. The best performing model has its metrics highlighted.

Model name	F_1	Precision	Recall
ChebNet	0.6015	0.6816	0.5382
ChebNet + DropEdge, $p = 0.10$	0.5789	0.6343	0.5325
ChebNet + DropEdge, $p = 0.20$	0.5633	0.6123	0.5215
ChebNet + DropEdge, $p = 0.50$	0.5252	0.5773	0.4818
ChebNet + DropEdge, $p = 0.80$	0.4496	0.5563	0.3773
GCN-2	0.4727	0.5613	0.4083
GCN-4	0.4602	0.5493	0.3959
GCN-4 + DropEdge, $p = 0.10$	0.4824	0.5594	0.4241
GCN-4 + DropEdge, $p = 0.20$	0.4536	0.5486	0.3867
GCN-4 + DropEdge, $p = 0.50$	0.3328	0.5179	0.2451
GCN-4 + DropEdge, $p = 0.80$	0.1090	0.2010	0.0748

Table 17: The corresponding per-class performance metrics for each entity for the best performing model, ChebNet, calculated on invoices with unseen templates.

Entity	F_1	Precision	Recall
amountCurrency	0.6378	0.7108	0.5784
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.6811	0.7870	0.6004
amountRndDiff	0.0417	0.5000	0.0217
amountTot	0.7912	0.8399	0.7479
amountVat	0.8000	0.8314	0.7709
dueDate	0.7284	0.7739	0.6879
invDate	0.7954	0.7903	0.8005
invNo	0.7273	0.8025	0.6649
ocrNo	0.6691	0.6475	0.6923
orderNo	0.2000	1.0000	0.1111
referenceName	0.0000	0.0000	0.0000
type	0.8674	0.8698	0.8650
undefined	0.9917	0.9895	0.9940
Macro average	0.6015	0.6816	0.5382

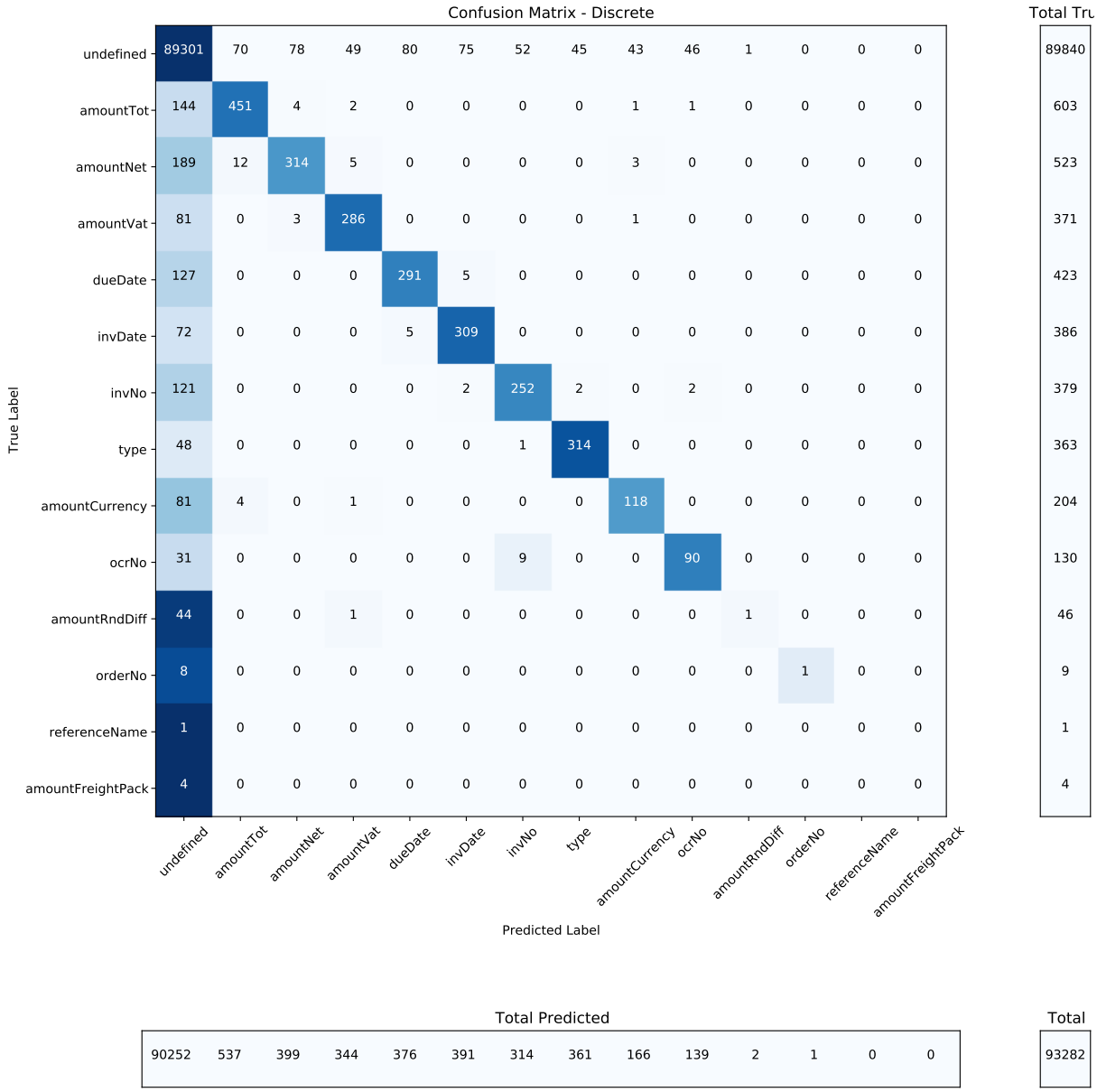


Figure 31: Discrete confusion matrix for the ChebNet model's performance on only the unseen invoice templates.

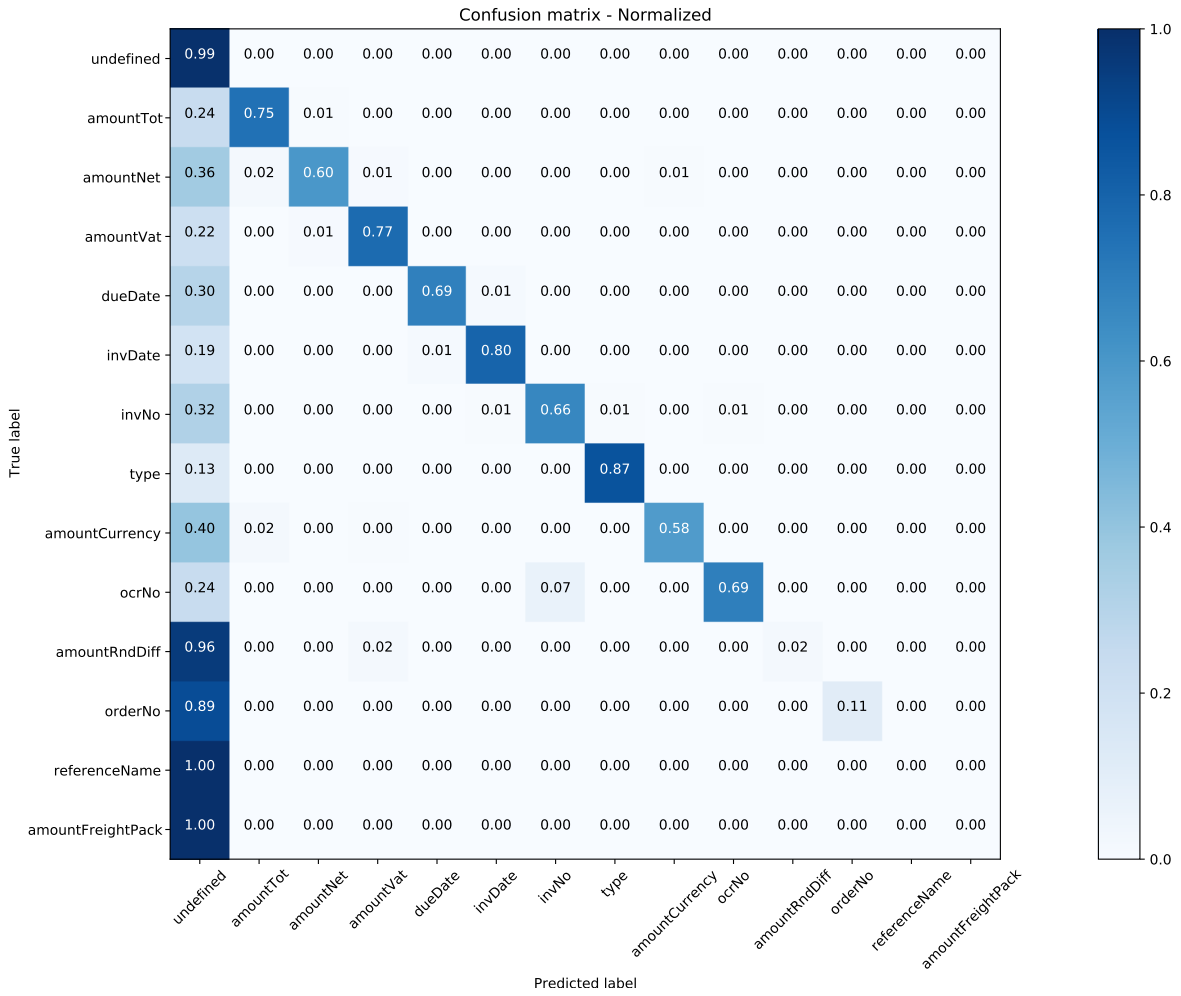


Figure 32: Normalized confusion matrix for the ChebNet model's performance on only the unseen invoice templates, rounded to the nearest hundredth. The per-class accuracy can be found in the diagonal of the matrix.

Table 18: Comparison of F_1-scores for each label as well as the macro average of the model when evaluated on the complete dataset versus invoices with unseen templates. The rightmost column illustrates the relative change, where the F_1-score for the complete dataset has been used as reference.

Label	F_1 - Complete Dataset	F_1 - Unseen Templates	Relative Change
amountCurrency	0.7387	0.6378	−13.66%
amountFreightPack	0.1250	0.0000	−100%
amountNet	0.7360	0.6811	−7.46%
amountRndDiff	0.3420	0.0417	−87.81%
amountTot	0.8172	0.7912	−3.18%
amountVat	0.8107	0.8000	−1.32%
dueDate	0.8110	0.7284	−10.19%
invDate	0.8681	0.7954	−8.38%
invNo	0.8166	0.7273	−10.94%
ocrNo	0.7108	0.6691	−5.87%
orderNo	0.4696	0.2000	−57.41%
referenceName	0.1639	0.0000	−100%
type	0.9132	0.8674	−5.02%
undefined	0.9948	0.9917	−0.31%
Macro average	0.7119	0.6015	−15.51%

12 Discussion

The following sections will analyze and discuss the results shown in Section 11 as well as identify underlying errors within the invoice dataset and the logit outputs from the ChebNet model, which can explain the behavior of all different models. Furthermore, while recall_{macro}, precision_{macro} and the F_{1macro} score have been published for all the models, we denote the precision_{macro} and the F_{1macro} metrics to be the most important metrics for deciding which model is best suited for information extraction of invoices. In practice, it is better to get no prediction output for a given label (i.e., the model labels the node as *undefined*) than to get a nonsense-prediction. However, the F_{1macro} score is still taken into account to make sure that the model is somewhat well-rounded.

12.1 Performance Difference Between Models

When analyzing Table 14 it becomes evident that the ChebNet model without DropEdge has the highest performance for all metrics. Not only does the ChebNet model outperform the other models on per-class F_1-score, precision and recall, it also showcases a lower validation loss, as well as a higher F_{1macro}, precision_{macro} and recall_{macro}. One explanation could be that, for the ChebNet model, the features of a centered vertex convolve with features of nodes that are further away compared to the GCN models. In other words, the ChebNet performs more K-hops per layer compared to GCN-2 and GCN-4, making it possible for the model to learn both local and more global patterns in invoices. As the deepest GCN model, GCN-4, has an equal amount of stacked layers as the ChebNet model, the GCN models are fairly constrained in terms of the size of convolutional neighborhoods as compared to the ChebNet model. Furthermore, the ChebNet model has a larger amount of trainable parameters at each layer (almost three times as many, see Table 11, 12 and 13), making it more complex and potentially better at learning complex patterns in the underlying training data. The combination of a lower K and a simpler network architecture can explain why the GCN models do not perform as well as the ChebNet model.

Regarding GCN-4 and GCN-2, we can see that the deeper GCN-4 performs slightly better than its shallower counterpart. This is in line with the previous reasoning regarding the impact of a larger K-locality for a model as GCN-4 convolves a centered node's features with nodes that are up to 4 hops away, compared to 2 hops for the GCN-2 model. This can further be explained by the fact that GCN-4 is a deeper network and thus is more able to capture underlying structural data. Moreover, even though GCN-2's first layer is twice the size of GCN-4's first layer, which may result in a greater feature representation of the original input nodes, the level of depth in GCN-4 as compared to GCN-2 can compensate for this narrower layer, since GCN relies on stacking multiple layers to account for a fixed K. However, it is possible that the narrowness of the first convolutional layer in GCN-4 contributes to the fact that we do not see an even greater increase in performance. Reducing the number of features from 310 down to 32 is already a reduction of almost 90 %, by halving that to 16 we end up with a feature representation of circa 5 % of the original input node features.

Another reason as to why GCN-4 shows only a slight increase in performance as compared to GCN-2 may be because of over-smoothing. Recall that over-smoothing occurs when node representations become more and more similar to each other and, in the extreme case, results in indistinguishable node features. This can happen when representations of nodes converge to a stationary point through the effect of, e.g., having too deep of a network. GCN-4 is not particularly deep, it is however deeper than GCN-2. However, if over-smoothing was an occurring problem for GCN-4, DropEdge should in theory have made a difference in performance for said model. This is further discussed in Section 12.2.

Another insight from training the different models is that none of the models have diverging train and validation loss curves. Figure 33 shows the train and validation loss curves for the ChebNet model (train and validation loss curves for the rest of the models can be found in **Appendix A**).

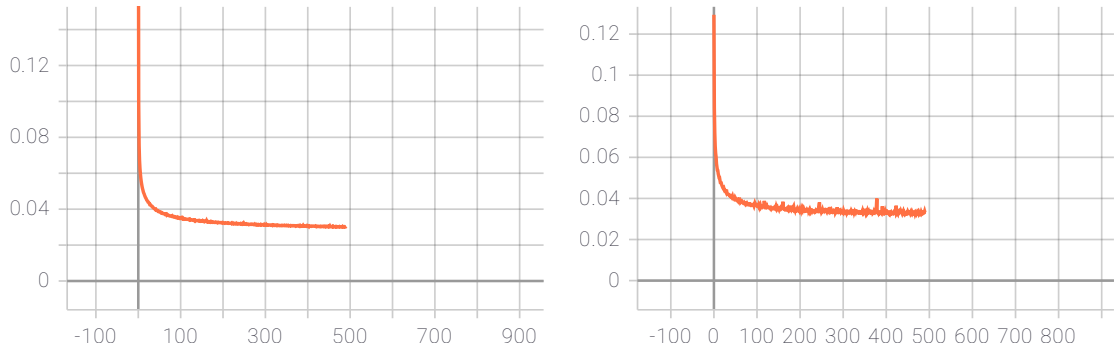


Figure 33: The train (left) and validation (right) loss curves for the ChebNet model. The y-axis shows the calculated loss while the x-axis shows at which epoch the loss has been calculated.

As Figure 33 shows, the curves follow each other closely and do not diverge. The same phenomenon can be found when inspecting the loss curves of the other models as well. This may imply that none of the models are complex enough to accurately describe the training data, as it never seems to overfit to it. While we do utilize L_2-regularization to prevent overfitting, the regularization term is quite small and we did still expect some level of overfitting to occur. If the models are in fact too simple, it may be the case that the regularization techniques in place damage the performance of the models instead of improving it. It may be worthwhile to test running the models both completely without L_2-regularization as well as with an increased patience to see if it has a positive impact on the performance. Moreover, early stopping is also utilized to prevent overfitting and the behavior of the loss curves can imply that our patience for early stopping is too narrow and that increasing it could yield a lower validation loss. Furthermore, if the models have oversimplified network architectures, a wider and/or deeper network design could in theory achieve better results.

12.2 DropEdge

Recall from Section 5.5.4 that DropEdge is a technique used to combat both overfitting and over-smoothing by enforcing some non-zero elements in the adjacency matrix, at random with a probability p, to be converted to zeroes, thus dropping some connections between the nodes. Doing so introduces sparsity in the network, which when applied can reduce overfitting and over-smoothing.

Table 14 in Section 11 shows the difference in performance between the different models, both with and without applying DropEdge. The results clearly demonstrate that neither ChebNet or GCN-4 in general benefited from utilizing DropEdge, since the performance for both of these models decreased as the effects of DropEdge was increased, i.e., as p increases, the model performance drops. The exception is DropEdge with $p = 0.10$ for the GCN-4 model, which has a slight increase for the recall_{macro} and in turn a slightly higher F_{1macro} score as well, as compared to GCN-4 without DropEdge. However, as the precision_{macro} has dropped for the DropEdge model, it is still arguably a worse model than the standard GCN-4 in the context of information extraction from invoices. This suggests that, for the current graph data representation and model architecture for ChebNet and GCN-4, DropEdge makes the network too sparse as the graph representation we have chosen for our invoice data (see Section 7.3) is already very sparse even before applying DropEdge. This can lead to the network not being able to capture the underlying structure in the data while utilizing DropEdge, thus underfitting the model.

Another explanation to why we don't see any performance gain from DropEdge can be related to the implementation of early stopping. As mentioned in Section 9 the patience for each model during training is set to a size of 50, meaning that training will stop after 50 consecutive epochs in which the validation loss has not improved. Applying DropEdge results in slower convergence and hurts the performance during the earlier stages of training, and by setting a patience window that is too small could disallow the network to converge to the lowest possible validation error, as stated in 5.5.4. Furthermore, as the models run a risk of being too simple to properly capture the patterns in the underlying data, the overfit reducing properties in DropEdge may lead to a reduced model performance.

12.3 Impact of Data Discrepancies

In Section 7.5 we discuss some of the identified discrepancies in the data which could have a negative effect on the performance of the models. In this section, we have analyzed how the ChebNet model performs on the invoices discussed in Section 7.5 and hypothesize on how the aforementioned discrepancies may impact the results seen in e.g., Figure 30 in Section 11.

When revisiting the invoices illustrated in Figure 24 and 25 in Section 7.5 and feeding them to the trained ChebNet model, we can indeed see that the model struggles with the aforementioned entities. In Figure 34, we can see that the model fails to classify the *ocrNo* entity to the left but does indeed classify the *ocrNo* entity to the right as the correct label. However, since the ground truth label is incorrect, the model is still penalized.

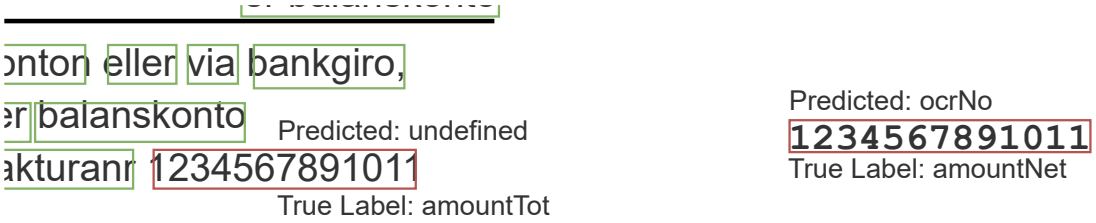


Figure 34: The OCR number is wrongly labeled as *amountTot* (left) and *amountNet* (right) because of errors existing in the label data which have a negative effect on the model performance.

In Figure 35, we can see that the model struggles to correctly extract the *amountNet* entity. This may be caused by the multiple bounding boxes spawned for each individual word. By having multiple bounding boxes for a single word, the surrounding neighborhood of the *amountNet* entity becomes noisy, which may have a negative impact on the node feature representations after convolving the node signals.

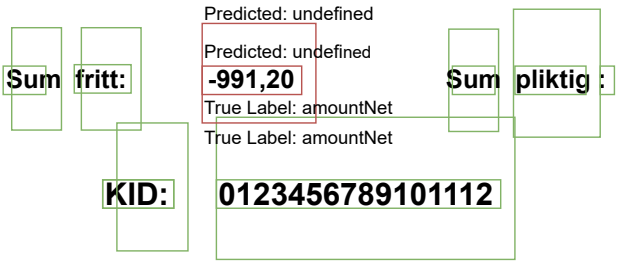


Figure 35: An example of when the OCR engine has produced multiple bounding boxes for each word in a given section. The model believes that all nodes should be labeled as *undefined*. However, the two red nodes both have the ground truth label *amountNet*.

There are also multiple cases where the discrepancy described in 21 had a negative effect on the model performance metrics. Figure 36 is an example of when the correct value has been extracted by the ChebNet model, but the model is still evaluated as being wrong since the incorrect node was classified as *amountNet*, even though it contains the correct text value.

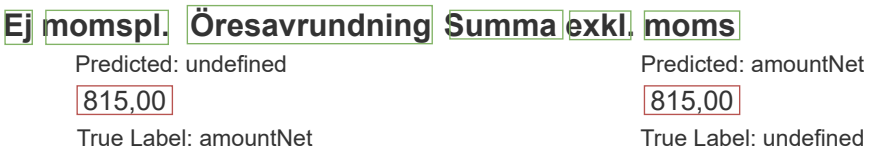


Figure 36: The model correctly extracts the *amountNet* value of 815,00, but predicted the wrong node as belonging to the aforementioned class. Green bounding boxes indicate nodes that are correctly classified as *undefined*.

By inspecting the confusion matrices in Figure 29 and Figure 30 in Section 11, one can see that there are some classes being confused for each other by the model, one such example is *amountTot* and *amountNet*. While the two classes are fairly similar in nature and would explain a lot, if not most of this confusion, another plausible explanation for this is the discrepancy illustrated in Figure 26, Section 7.5. As we can see in Figure 37, the two red boxes at the top of the figure are correctly classified by the model as *amountNet*, but would be penalized during training for this since the nodes have the wrong ground truth label *amountTot*. The same is true for the bottom and right-most two nodes, but where the total amount is actually labeled as *amountNet*.

Figure 37: The model correctly classified the two top-right nodes as *amountNet* and the two bottom-right nodes as *amountTot*. However, the nodes have the wrong ground truth label. Green nodes are correctly classified as *undefined*.

The same kind of issue in the data can also be found between the classes *invNo* and *ocrNo*, which could explain part of why 5 % of all *ocrNo* nodes are classified as *invNo*, as well as why 2 % of the *invNo* nodes are labeled as *ocrNo*.

Furthermore, in Figure 30, Section 11 we see that roughly 1 % of all occurring *amountCurrency* labels are classified as *amountTot*, the type of aforementioned mislabeling illustrated in Figure 22 (Section 7.5) is a plausible explanation for this phenomenon.

12.4 Per-Class Performances

As already established in Sections 7.5 and 12.3, there exists noise in the data of various degrees and of different character for the different classes. Arguably, the model still performs well on classes which have a relatively high number of occurrences in the data (e.g., *amountTot*, *dueDate* and *invDate*), while it performs poorly on classes with fewer occurrences (e.g., *amountRndDiff*, *referenceName* and *orderNo*).

Investigating the confusion matrix for the ChebNet model (Figure 30, Section 11) we see that even the best model showcases a subpar performance for the classes *amountRndDiff*, *orderNo*, *referenceName* and *amountFreightPack*, compared to the other class performances, where the highest F_1-score of these four classes is roughly 0.47 (*orderNo*). We also see that the nodes the model failed to classify correctly and that have a true label that belong to either of these four aforementioned classes, have been classified as *undefined* almost exclusively. If we also look at Figure 19 from Section 7.4, we recall that the class *undefined* accounts for a vast majority ($\sim$ 97 %) of all class occurrences. Furthermore, Figure 20 from the same section aptly illustrates how imbalanced the dataset is in regards to the four aforementioned classes. Out of more than 80000 invoices, only a small fraction of these contain nodes belonging to *amountRndDiff*,

orderNo, *referenceName* and *amountFreightPack*. Given these insights, our hypothesis is that our model performs poorly on these four classes because there simply is not enough data for the model to be able to capture the underlying structure of nodes belonging to said classes. Section 12.5 delves deeper into the reasons for the way these nodes are classified. However, if we omit the aforementioned four least frequently occurring classes from Table 15 and instead calculate the macro averages for the remaining 10 classes based on the performance metrics of the ChebNet model, we see an increase of the F_1-score by +15.5 % (from 0.7119 to 0.8224), of the precision by +1.7 (from 0.8259 to 0.8406) and of the recall by 28.7% (from 0.6255 to 0.8050). Table 19 shows the new macro averages for the evaluation on these 10 classes.

Table 19: Per-class performance metrics for the 10 most frequently occurring entities within the training set, evaluated using the ChebNet model. The entities are in descending order in regard to their frequency of occurrence. The four *least* frequently occurring entities have been omitted when calculating the new macro averages and have been struckthrough.

Frequency	Entity	F_1	Precision	Recall
97.23 %	undefined	0.9948	0.9938	0.9958
0.44 %	amountTot	0.8172	0.8258	0.8088
0.39 %	amountNet	0.7360	0.7964	0.6842
0.33 %	amountVat	0.8107	0.8348	0.7881
0.32 %	dueDate	0.8110	0.8171	0.8050
0.32 %	invDate	0.8681	0.8641	0.8723
0.30 %	invNo	0.8166	0.8565	0.7802
0.30 %	type	0.9132	0.9040	0.9227
0.20 %	amountCurrency	0.7387	0.7918	0.6923
0.12 %	ocrNo	0.7108	0.7215	0.7003
~~0.045 %~~	~~amountRndDiff~~	~~0.3420~~	~~0.7333~~	~~0.2230~~
~~0.026 %~~	~~orderNo~~	~~0.4696~~	~~0.8682~~	~~0.3218~~
~~0.0045 %~~	~~referenceName~~	~~0.1639~~	~~0.5556~~	~~0.0962~~
~~0.0017 %~~	~~amountFreightPack~~	~~0.1250~~	~~1.0000~~	~~0.0667~~
Macro average		0.8224	0.8406	0.8050

As Table 19 suggests, there may very well be a correlation between the frequency of occurrence of a class and the ChebNet model's performance on these classes.

From Figure 20 in Section 11, it also becomes clear that the *amountTot* and *amountNet* classes occur in the vast majority of invoices, while classes such as *dueDate*, *invDate* and type occur more infrequently (albeit relatively often compared to other classes). The *amountVat* class occurs as often as *dueDate*, *invDate* and *type*. Despite the high occurrence of the *amount* classes, our model is better at classifying *dueDate*, *invDate* and *type*. This may be linked to the complex nature of the *amount* classes' data as well as the errors present within them, as described in Section 7.5. A total/net/VAT amount value can occur in several different formats and places on an invoice, such as "12345,00", "12 345,00" or "12345" et cetera, whereas a date value follows standardized formats such as "2021.01.02", "02.01.2021" or "2 januari 2021" which is easily picked up by the date parser in the feature calculator process. In addition, if there are dates present within an invoice, it will most likely be an invoice or due date, whereas the amount values are non standardized format numbers which can be similar to values unrelated to currency amounts, e.g., article quantities or page numbers. In other words, the amount node values' are possibly not as distinguishable and unique as compared to e.g., date nodes. Furthermore, as we have also seen in Section 7.5, the total/net/VAT amount values can occur in several different places on one unique invoice. Moreover, where on the invoice the labeled value is placed differs from invoice to invoice. This results in different neighborhood relationships based on the location of the labeled value. On the contrary, an invoice usually has one due date and one invoice date, and standardized values and neighbors on top of that, which yields a lower variance in unseen data and thus may make it easier for the model to learn and classify these representations.

It is worth to note that a lot of nodes are misclassified as *undefined*, as can be seen in any of the

confusion matrices, e.g., Figure 29. While it would be ideal to have a model that perfectly classifies each node correctly, the next best option is to have a model that defaults to *undefined* when it is not sure of its predictions. From a business perspective, it is often better to get no prediction at all (e.g., a node that is misclassified as *undefined*) than to get a nonsensical one (e.g., a node with the true label *amountTot* that is misclassified as an *amountNet*). For instance, say we want to extract the *amountNet*, *amountTot* and *amountVat* values for accounting purposes. Then, it is better to not extract an entity (i.e., classify the entity as *undefined*) than to incorrectly extract, say, a net amount as a total amount, since incorrectly extracted entities can have severe consequences for this purpose.

12.5 Logit Analysis

The logit output of the models can be interpreted as how certain the models are that the classifications hold true. By analyzing the logit output of the best performing model we can more aptly gain an understanding of the beliefs it has. In this section, we will focus more on the less frequent classes. The full list of logit plots for each label can be found in **Appendix B**.

In Figure 38, we can see that a majority of *amountRndDiff* nodes are classified as *undefined* with a high degree of certainty. Through manual inspection of 100 randomly selected invoices from the validation dataset, we can see that while a majority of the inspected invoices have nodes which contain information about the rounding difference, they are not present in the label dataset and are hence labeled as *undefined*. Out of the 100 inspected invoices, 59 % of the invoices have nodes which should be classified as belonging to the *amountRndDiff* class. This can be compared to the training set which contains 74 396 invoices, out of which only 10 450, or roughly 14 %, of the invoices have fields labeled as *amountRndDiff*.

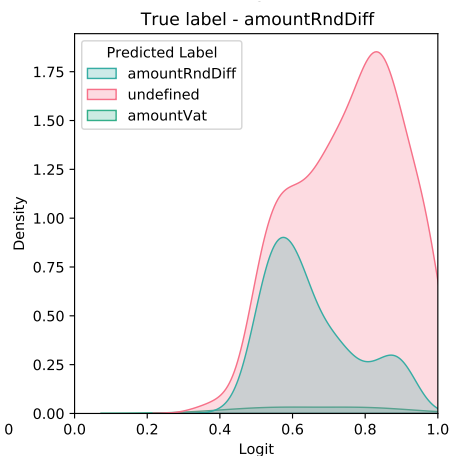


Figure 38: Kernel Density Estimation plot of predicted label logits, given the true label *amountRndDiff*. Most nodes belonging to this class are classified as *undefined*, where the majority of these classifications have a logit value of > 0.7.

While the manual inspection of 100 invoices is too small of a number to give a true estimation of the number of invoices that actually contain information about the rounding difference, if our analysis is any indication of the true distribution of this class, this means that the model is actively trained to classify nodes from the *amountRndDiff* class as nodes belonging to the *undefined* class in the majority of cases. This would explain both the high concentration of classifications with the *undefined* label for *amountRndDiff* nodes, as well as the high value of the logits when doing so. A similar pattern can be found when analyzing the logits of the *orderNo*, *referenceName* and *amountFreightPack* as well, see Figure 39, although further inspection of the aforementioned labels are needed to be able to draw a similar conclusion regarding issues with the annotation of these labels.

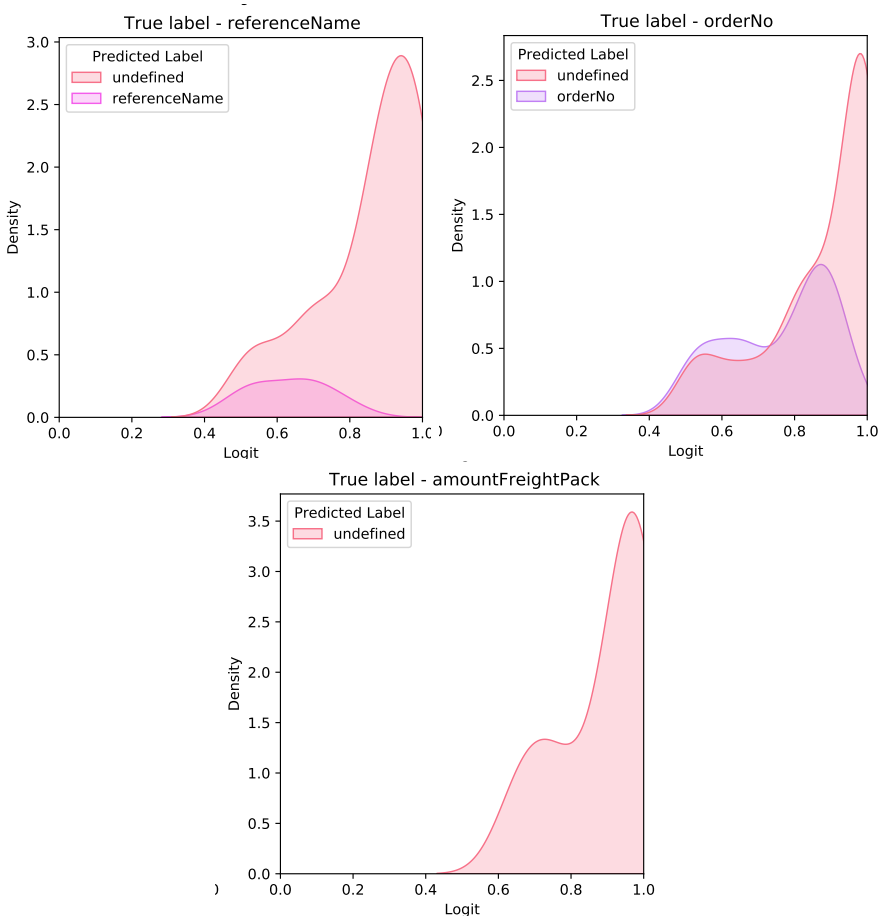


Figure 39: Kernel Density Estimation plots of predicted label logits, given the true label *referenceName*, *orderNo* and *amountFreightPack*, respectively. Most nodes belonging to these classes are classified as *undefined*, where the majority of these classifications have a logit value of > 0.6.

12.6 Level of Template Agnosticism

In Section 2 it is stated that one of the main reasons for choosing to implement Lohani et al.'s [3] architecture for this project was due to their model's ability to generalize across templates, i.e., its high level of template agnosticism. Recall from Section 7.4 that roughly 14% of the vendors in the test set are not present in either the train or validation set, meaning that the invoices by these vendors have templates that are assumed to remain unseen during training. It should be noted, however, that the whole test dataset contains more than 4000 invoices, while the subset of invoices with unseen templates consists of less than 400. We are aware that the small size of the subset may indeed bring results of less statistical significance and that a larger number of invoices with unseen templates would give a better approximation of the model performance.

Table 16 in Section 11 once again shows that the ChebNet model has the best performance with respect to all three metrics on the unseen template subset. However, the overall performance on the unseen template subset has decreased when compared to the whole dataset, as can be seen in Table 18 in the same Section. While the model still performs fairly well on more commonly occurring classes, exhibiting a relative decrease in F_1 performance in the range 1.32 - 13.66 %, it becomes apparent that the model experiences the heaviest losses in performance for predictions over the less frequent classes (i.e., *ocrNo*, *amountRndDiff*, *orderNo*, *referenceName* and *amountFreightPack*) with a relative decrease of up to 100 %. The poor performance on these classes is not surprising, however, as the model already performs poorly on these classes even when evaluating the model on the whole test set, as discussed in Section 12.4.

We can also see a trend where, when applying DropEdge to the model, the performance of the model

decreases as the effects of DropEdge increases. This is somewhat surprising as DropEdge should reduce some of the overfitting of the model and we would have thought that this would result in a slightly better performance on invoices with unseen templates. A similar reasoning for explaining the performance drop on the whole dataset when applying DropEdge could be applied here as well, i.e., because of the relatively small number of learnable parameters (roughly 49 000) and regularizing an already overly simplified model could indeed impact the performance of the model negatively. Furthermore, the already sparse connectivity in the input graph could lead to DropEdge dropping vital edges for node classification. It is interesting however, that the GCN-4 with an added DropEdge probability of $p = 0.10$ shows a better performance with regards to all metrics when compared to its DropEdge-less counterpart. It also shows an increase in both F_{1macro} and recall_{macro} when compared to GCN-2. To be able to more accurately determine if DropEdge has a positive effect or not on invoices with unseen templates, it would be beneficial to perform a study where the test set contained a larger fraction of invoices with unseen templates to be able to draw clearer conclusions on the effects of DropEdge in this scenario.

13 Future Work

There are a number of changes and additions to the system that can be implemented in future work in order to improve the models' overall performances. As aforementioned, the amount of errors in the data due to mistakes made by humans are considerable. Errors such as mislabeled text values are easy, albeit time consuming, to mitigate, and if these errors are taken care of, better data would be fed into the network which would most likely yield better results.

Another improvement would be to tune certain parameters. The hyperparameters, e.g., the learning rate and the patience window, can be fine tuned to optimal values via the help of methods such as grid search (i.e., performing a search through a subset of the hyperparameter space of a learning algorithm). We can also add more/other features to each node. For instance, other booleans such as *isFirstName*, *isCity*, *isZipCode* et cetera, may provide the network with a better understanding of each text value.

Furthermore, instead of comparing predicted labels to the true labels of a given node, a third improvement could be to compare the predicted labels' text value against the true labels' either formatted value or raw text value, which can prove to be a more fair performance evaluation. Recall from Section 7.1 that the true label's value is a processed, parsed and possibly human-corrected text value (e.g., the text value "10 450,00" is formatted as "10450.00"). Comparing the inherent text values of the classes to the true, formatted, value can therefore provide a better and more fair understanding of the model performances. This will require some post processing heuristics in order to match the predicted label's text value against the formatted value. This would change the evaluation to being based on whether the correct *information* has been extracted instead of being based on whether the correct *node* has been extracted or not. However, as shown in Section 7.5, even though a true label is set to e.g., *amountNet* its actual text value can be something else entirely, e.g., an OCR number as in Figure 24 in Section 7.5. Hence, this approach could come with its own fair share of problems.

To gain an even better understanding of the level of template agnosticism of the model, it would be interesting to enforce a larger fraction of the vendors in the different dataset splits to be unique. As of today, only 393 invoices in the test set have templates not present in the validation and train splits. To increase this number and in turn get results with a greater level of statistical significance, especially for the less frequent classes, one could use a stratified split over different vendors to control the distribution of templates between the splits, instead of the random split used today.

Yet another alternative for possibly improving the overall performances would be to utilize batch normalization. Batch normalization is a technique that can stabilize the learning process by standardizing the inputs to a layer for each mini-batch. This can have the effect of accelerating the training by, in some cases, halving the amount of needed epochs while simultaneously reducing the generalization error [38].

Our architecture might just be too simple and not complex enough (which may be a reason for DropEdge not improving the models' performances) and having a more complex architecture, either by making the network wider or deeper (or both), could yield more favorable results.

14 Summary & Conclusion

The purpose of this thesis is to evaluate and compare three deep learning models, using two different architecture designs. Three different convolutional graph neural network models - ChebNet, GCN-2 and GCN-4 - have been trained, evaluated and compared on the same task: extracting information from invoices. Two of these three, ChebNet and GCN-4, have also been extended with the overfitting and over-smoothing battling technique DropEdge and re-trained with different dropping probabilities. The raw data for this task has been sourced with the help of an OCR engine and consists of text present in the invoices as well as corresponding bounding box positions for each value. The data has been processed, cleaned and turned into graph representations for each invoice, in order to be fed into a convolutional graph neural network.

The models have all been trained on a set of 74 492 invoices and tested on 4140 invoices. The results show that the best performing model when looking at the evaluation metrics F_{1macro}, $\mathrm{precision}_{macro}$ and recall_{macro} is the ChebNet model with a F_{1macro} of 0.7119, a $\mathrm{precision}_{macro}$ of 0.8259 and a recall_{macro} of 0.6255. The ChebNet model yields promising results even on invoices with unseen templates, especially on the most frequently occurring classes. Furthermore, GCN-4 proves to yield better performance than its shallower and less complex counterpart, GCN-2, albeit only with a small margin.

Analysis of the data shows underlying, inherent errors in the data which might have a significant impact on the outcome of the models. Wrongly labeled nodes, spurious nodes created by OCR artifacts, and missing node labels are a few of the errors found in the data which have a negative impact on the performance of the model, especially for the more infrequent occurring classes.

The reason for the overfitting and over-smoothing battling technique DropEdge not improving performance across the networks might have to do with the fact that the networks already are sparse and relatively simple in nature. Utilizing DropEdge introduces even more sparsiness which leads to underfitting the model, thus making the network unable to capture underlying trends in the data as deftly as without applying DropEdge.

The conclusion we gather is that convolutional graph neural networks prove to be useful for information extraction from visually rich documents, even for VRD:s with unseen templates. Moreover, due to the sparse nature of invoice documents as well as the low complexity of the network architectures, applying DropEdge for small networks operating on sparse graphs does not yield better performance in general, with the exception of GCN-4 with $p = 0.10$.

Lastly, there are some improvements that can be made to further enhance, as well as give further insights, to the different models' performances. These include tuning hyperparameters, applying batch normalization, adding new or other features, enforcing stratified sampling over vendors for better insight into the level of template agnosticism, and, most importantly, fixing the present errors in the data.

References

[1] DJS Research, "Global productivity study." https://info.unit4.com/rs/900-SZD-631/images/Unit4_Market_Research_Infographic_HiRes_nobleed_v3.pdf, 2017.

[2] F. Krieger, P. Drews, B. Funk, and T. Wobbe, "Information extraction from invoices: A graph neural network approach for datasets with high layout variety," 03 2021.

[3] D. Lohani, B. Abdel, and Y. Belaïd, "An Invoice Reading System using a Graph Convolutional Network," in *International Workshop on Robust Reading*, (PERTH, Australia), Dec. 2018.

[4] R. B. Palm, F. Laws, and O. Winther, "Attend, copy, parse – end-to-end information extraction from documents," 2021.

[5] X. Liu, F. Gao, Q. Zhang, and H. Zhao, "Graph convolution for multimodal information extraction from visually rich documents," 2019.

[6] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, p. 4–24, Jan 2021.

[7] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," 2017.

[8] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017.

[9] D. Chen, Y. Lin, W. Li, P. Li, J. Zhou, and X. Sun, "Measuring and relieving the over-smoothing problem for graph neural networks from the topological view," 2019.

[10] Y. Rong, W. Huang, T. Xu, and J. Huang, "Dropedge: Towards deep graph convolutional networks on node classification," in *International Conference on Learning Representations*, 2020.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, pp. 5, 82–92, 151–153, 168–169, 204–212, 277–282, 330–332, 335–337. Adaptive Computation and Machine Learning series, MIT Press, 2016.

[12] A. Engelbrecht, *Computational Intelligence: An Introduction*, pp. 17–18. Wiley, 2007.

[13] A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön, *Machine Learning - A First Course for Engineers and Scientists*, pp. 40–43, 51–53, 60, 62–68, 75–78, 93, 98–100,113–114, 117–118, 120–122, 124–129. 2021.

[14] F. Scarselli, M. Gori, A. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, pp. 61–80, 2009.

[15] E. Williamson, *Lists, Decisions and Graphs*, p. 148. S. Gill Williamson, 2010.

[16] N. Biggs, *Algebraic Graph Theory*, pp. 7–11. Cambridge: Cambridge University Press, second ed., 1993.

[17] F. R. K. Chung, *Spectral Graph Theory*, pp. 2–3. American Mathematical Society, 1997.

[18] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," 2014.

[19] D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst, "Signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular data domains," *CoRR*, vol. abs/1211.0053, 2012.

[20] D. K. Hammond, P. Vandergheynst, and R. Gribonval, "Wavelets on graphs via spectral graph theory," *Applied and Computational Harmonic Analysis*, vol. 30, no. 2, pp. 129–150, 2011.

[21] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, pp. 209–210. Berlin, Heidelberg: Springer-Verlag, 2006.

[22] R. Larson and B. H. Edwards, *Calculus*, p. 134. 2016.

[23] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[24] Y. Yao, L. Rosasco, and A. Caponnetto, "On early stopping in gradient descent learning," *Constr. Approx*, pp. 289–315, 2007.

[25] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[26] C. Yang, R. Wang, S. Yao, S. Liu, and T. Abdelzaher, "Revisiting over-smoothing in deep gcns," 2020.

[27] Y. Yang, "An evaluation of statistical approaches to text categorization," *Journal of Information Retrieval*, vol. 1, pp. 67–88, 1999.

[28] S. Ghannay, B. Favre, Y. Estève, and N. Camelin, "Word embedding evaluation and combination," in *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, (Portorož, Slovenia), pp. 300–305, European Language Resources Association (ELRA), May 2016.

[29] P. Gage, "A new algorithm for data compression," *The C Users Journal archive*, vol. 12, pp. 23–38, 1994.

[30] B. Heinzerling and M. Strube, "BPEmb: Tokenization-free pre-trained subword embeddings in 275 languages," in *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, (Miyazaki, Japan), European Language Resources Association (ELRA), May 2018.

[31] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (Doha, Qatar), pp. 1532–1543, Association for Computational Linguistics, Oct. 2014.

[32] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020.

[33] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[34] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008.

[35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[37] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[38] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

Appendix A

Results for GCN-2

Table 20: The corresponding performance metrics for each entity for the GCN-2 model, evaluated on the test set.

Entity	F_1	Precision	Recall
amountCurrency	0.6064	0.7530	0.5076
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.5191	0.7017	0.4119
amountRndDiff	0.1083	0.4932	0.0608
amountTot	0.6998	0.7965	0.6241
amountVat	0.5860	0.7479	0.4817
dueDate	0.6969	0.7525	0.6489
invDate	0.7249	0.7545	0.6975
invNo	0.5940	0.7188	0.5062
ocrNo	0.5375	0.6500	0.4581
orderNo	0.0865	0.7273	0.0460
referenceName	0.0000	0.0000	0.0000
type	0.8295	0.8125	0.8472
undefined	0.9917	0.9881	0.9953
Macro average	0.5262	0.6354	0.4490
Validation loss			0.0530

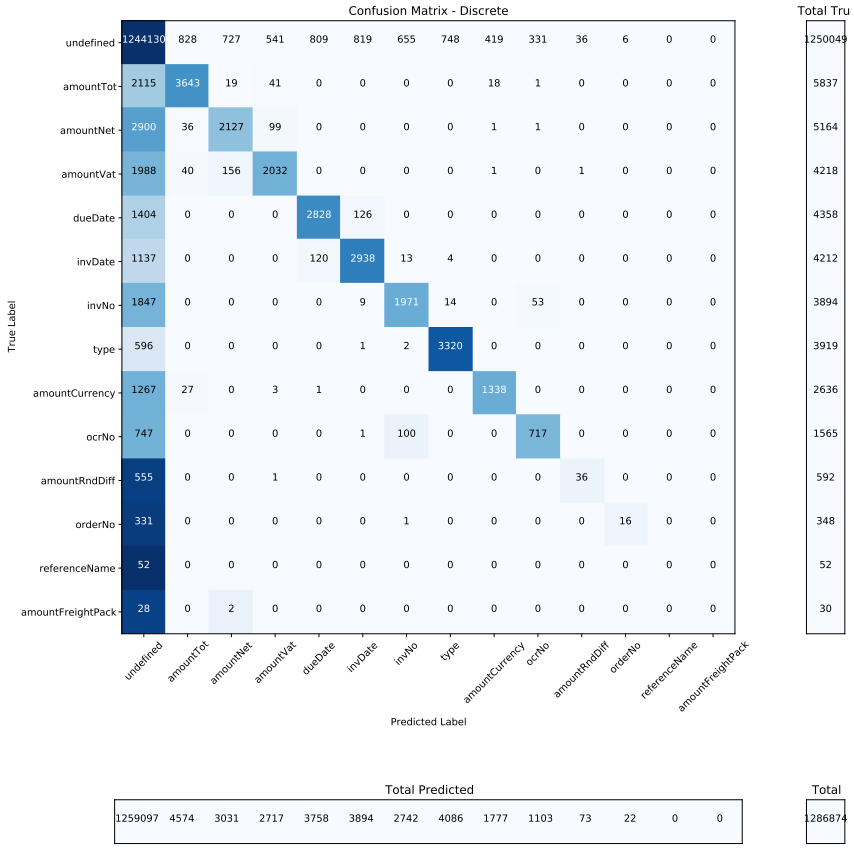


Figure 40: Discrete confusion matrix for the GCN-2 model on the whole test set.

Figure 41: Normalized confusion matrix for the GCN-2 model on the whole test set.

Table 21: The corresponding per-class performance metrics for each entity for the GCN-2 model, calculated on invoices with unseen templates.

Entity	F_1	Precision	Recall
amountCurrency	0.5477	0.7355	0.4363
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.4245	0.6653	0.3117
amountRndDiff	0.0400	0.2500	0.0217
amountTot	0.6402	0.8307	0.5207
amountVat	0.6044	0.8165	0.4798
dueDate	0.6287	0.6943	0.5745
invDate	0.7000	0.7316	0.6710
invNo	0.4825	0.6532	0.3826
ocrNo	0.5752	0.6771	0.5000
orderNo	0.0000	0.0000	0.0000
referenceName	0.0000	0.0000	0.0000
type	0.8226	0.8214	0.8237
undefined	0.9884	0.9825	0.9944
Macro average	0.4727	0.5613	0.4083

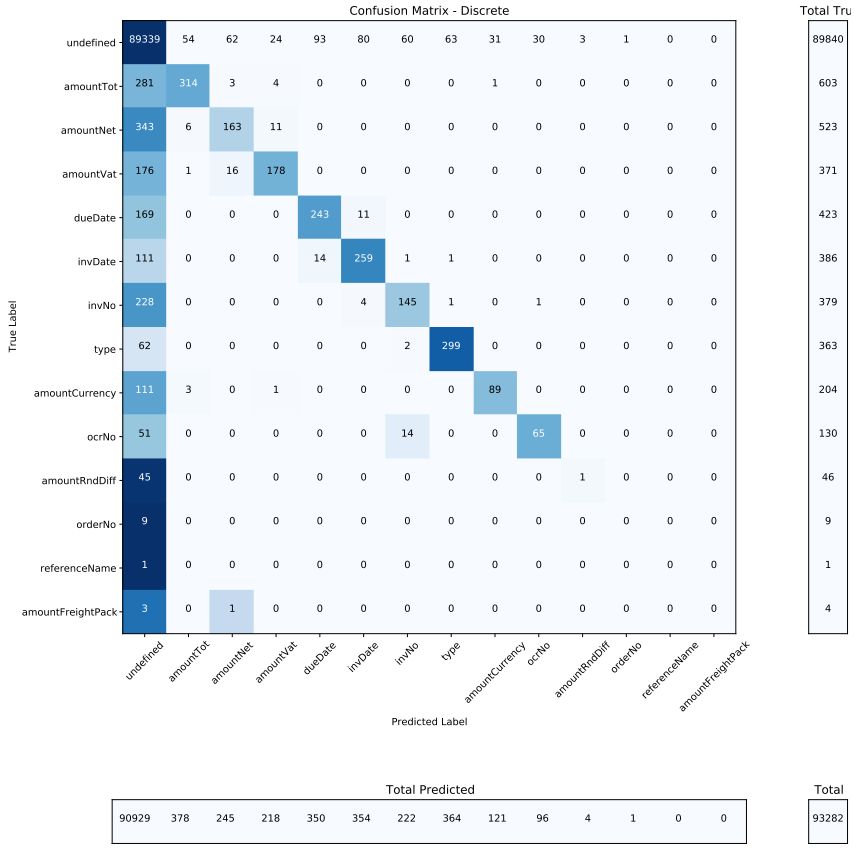


Figure 42: Discrete confusion matrix for the GCN-2 model on the unseen templates subset.

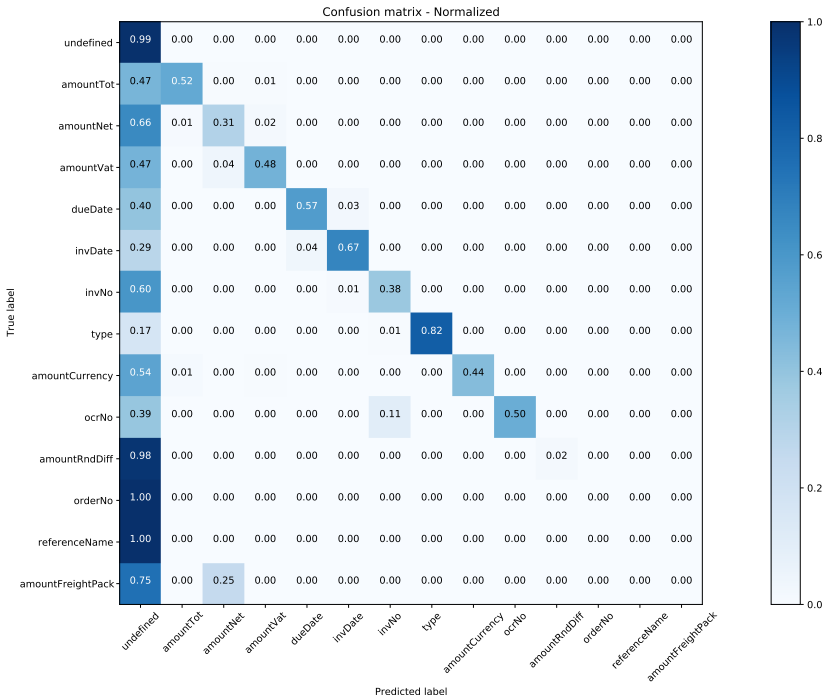
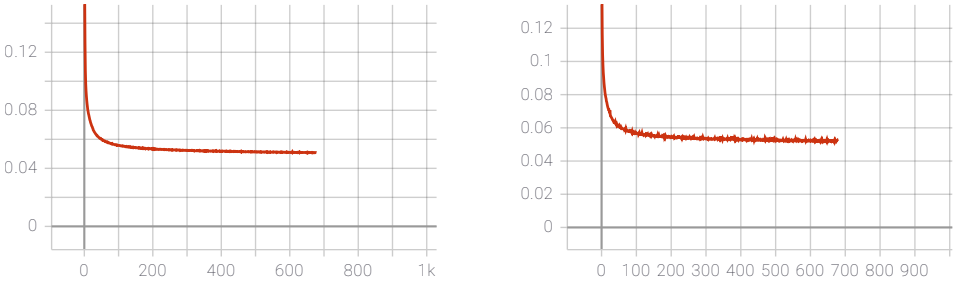


Figure 43: Normalized confusion matrix for the GCN-2 model on the unseen templates subset.



(a) Train loss (b) Validation loss

Figure 44: The train (a) and validation (b) loss curves for the GCN-2 model. The y-axis shows the calculated loss while the x-axis shows at which epoch the loss has been calculated.

Results for GCN-4

Table 22: The corresponding performance metrics for each entity for the GCN-4 model, evaluated on the test set.

Entity	F_1	Precision	Recall
amountCurrency	0.6212	0.7760	0.5178
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.5272	0.7104	0.4191
amountRndDiff	0.0297	0.6429	0.0152
amountTot	0.6978	0.7760	0.6339
amountVat	0.6164	0.7176	0.5403
dueDate	0.7035	0.7612	0.6540
invDate	0.7373	0.7983	0.6849
invNo	0.5754	0.7478	0.4676
ocrNo	0.5051	0.6555	0.4109
orderNo	0.0771	0.9333	0.0402
referenceName	0.0000	0.0000	0.0000
type	0.8163	0.8492	0.7859
undefined	0.9918	0.9880	0.9956
Macro average	0.5309	0.6683	0.4404
Validation loss			0.0550

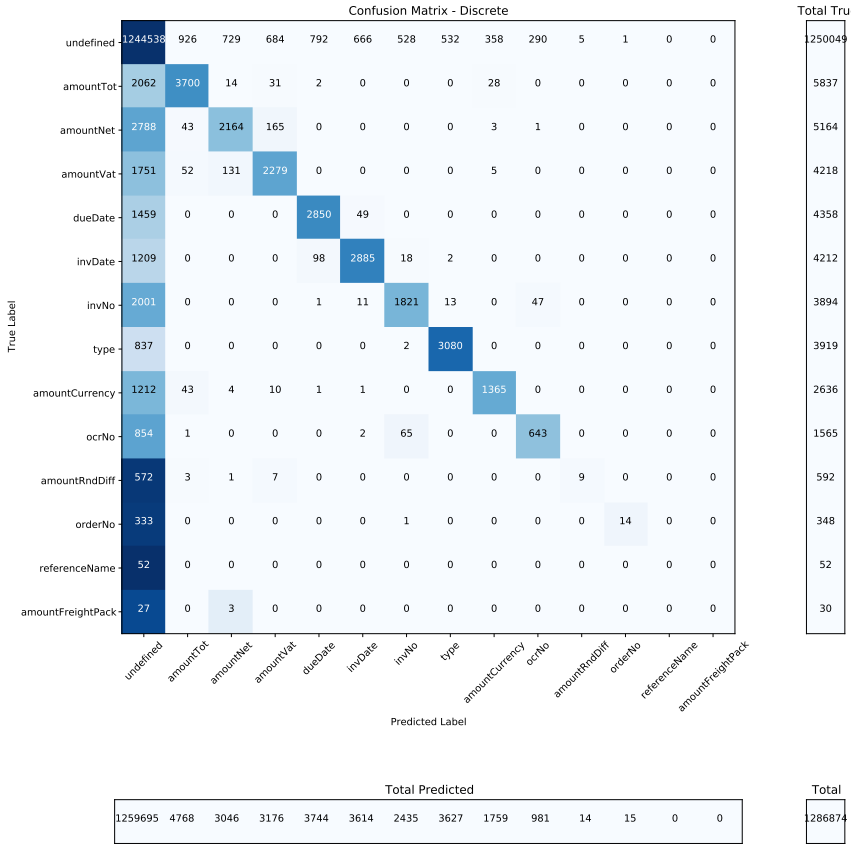


Figure 45: Discrete confusion matrix for the GCN-4 model on the whole test set.

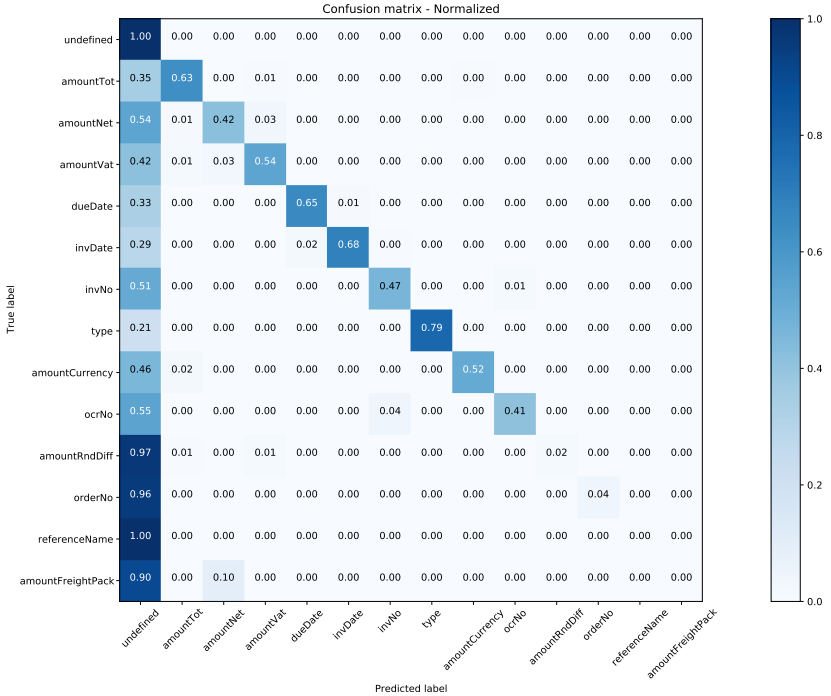


Figure 46: Normalized confusion matrix for the GCN-4 model on the whole test set.

Table 23: The corresponding per-class performance metrics for each entity for the GCN-4 model, calculated on invoices with unseen templates.

Entity	F_1	Precision	Recall
amountCurrency	0.5079	0.7207	0.3922
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.4846	0.7354	0.3614
amountRndDiff	0.0000	0.0000	0.0000
amountTot	0.6399	0.8096	0.5290
amountVat	0.6541	0.7849	0.5606
dueDate	0.6313	0.7190	0.5626
invDate	0.7079	0.7730	0.6528
invNo	0.4664	0.7059	0.3483
ocrNo	0.4929	0.6420	0.4000
orderNo	0.0000	0.0000	0.0000
referenceName	0.0000	0.0000	0.0000
type	0.7775	0.8176	0.7410
undefined	0.9884	0.9825	0.9944
Macro average	0.4602	0.5493	0.3959

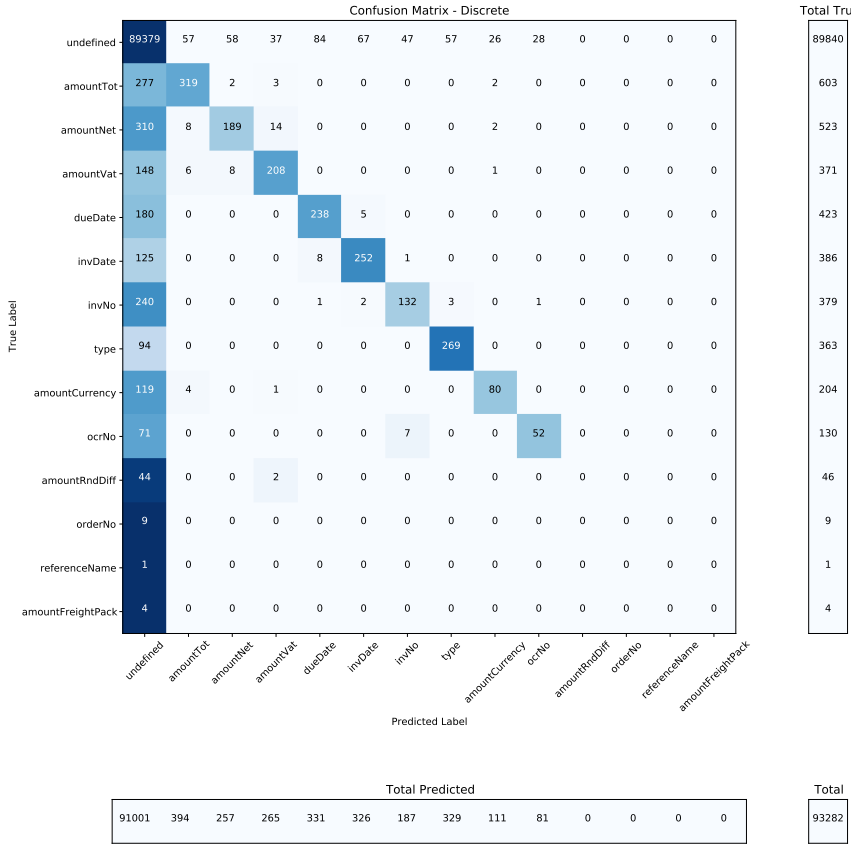
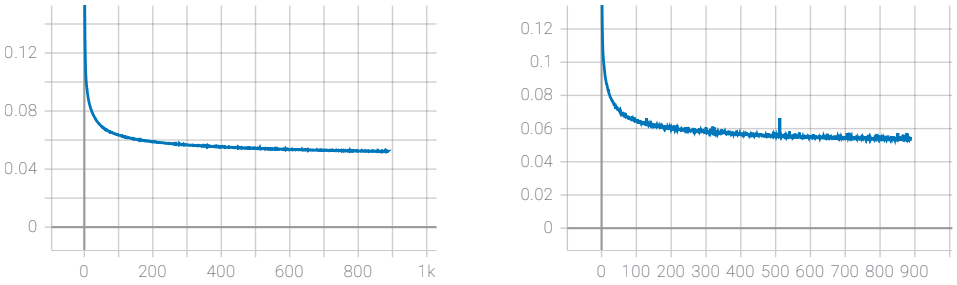


Figure 47: Discrete confusion matrix for the GCN-4 model on the unseen templates subset.

Figure 48: Normalized confusion matrix for the GCN-4 model on the unseen templates subset.



(a) Train loss

(b) Validation loss

Figure 49: The train (a) and validation (b) loss curves for the GCN-4 model. The y-axis shows the calculated loss while the x-axis shows at which epoch the loss has been calculated.

Results for GCN-4 + DropEdge, $p = 0.10$

Table 24: The corresponding performance metrics for each entity for the GCN-4 + DropEdge model with $p = 0.10$, evaluated on the test set.

Entity	F_1	Precision	Recall
amountCurrency	0.6283	0.7847	0.5239
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.5526	0.6922	0.4599
amountRndDiff	0.0264	0.6154	0.0135
amountTot	0.7218	0.7886	0.6654
amountVat	0.6312	0.7427	0.5488
dueDate	0.7146	0.7598	0.6744
invDate	0.7421	0.7871	0.7020
invNo	0.5791	0.7535	0.4702
ocrNo	0.5353	0.6556	0.4524
orderNo	0.1845	0.6981	0.1063
referenceName	0.0000	0.0000	0.0000
type	0.8284	0.8532	0.8051
undefined	0.9920	0.9886	0.9954
Macro average	0.5381	0.6514	0.4584
Validation loss			0.0510

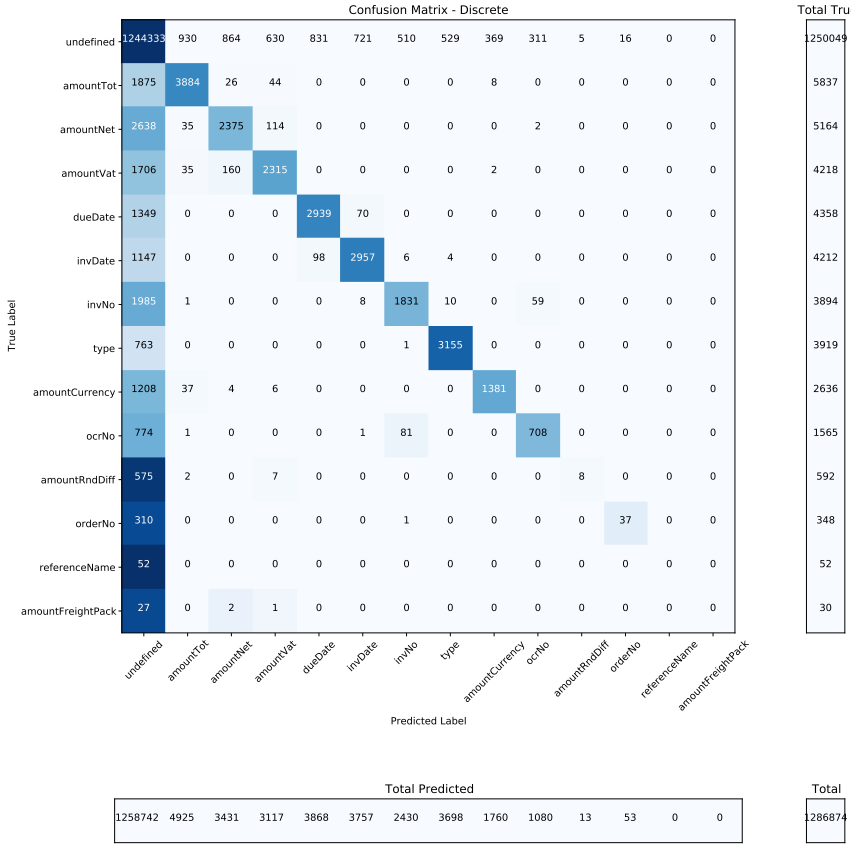


Figure 50: Discrete confusion matrix for the GCN-4 + DropEdge model with $p = 0.10$ on the whole test set.

Figure 51: Normalized confusion matrix for the GCN-4 + DropEdge model with $p = 0.10$ on the whole test set.

Table 25: The corresponding per-class performance metrics for each entity for the GCN-4 + DropEdge model with $p = 0.10$, calculated on invoices with unseen templates.

Entity	F_1	Precision	Recall
amountCurrency	0.5919	0.8120	0.4657
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.5111	0.7148	0.3977
amountRndDiff	0.0000	0.0000	0.0000
amountTot	0.6860	0.8218	0.5887
amountVat	0.6591	0.8226	0.5499
dueDate	0.6735	0.7415	0.6170
invDate	0.7270	0.7726	0.6865
invNo	0.5078	0.7449	0.3852
ocrNo	0.5236	0.5922	0.4692
orderNo	0.0000	0.0000	0.0000
referenceName	0.0000	0.0000	0.0000
type	0.8034	0.8256	0.7824
undefined	0.9890	0.9835	0.9946
Macro average	0.4824	0.5594	0.4241

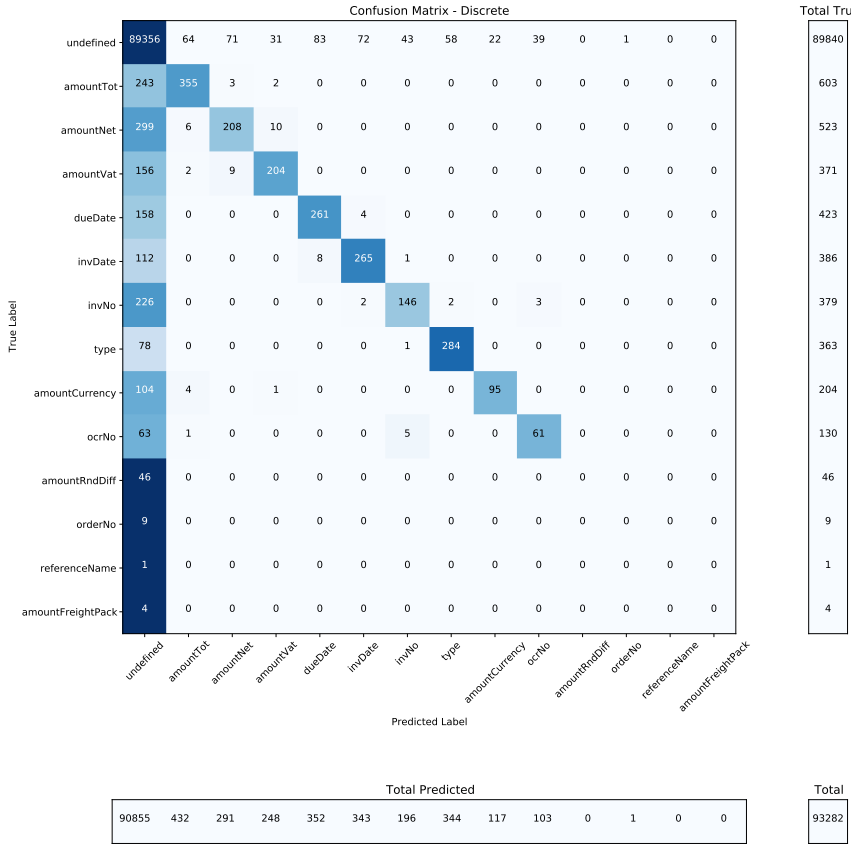
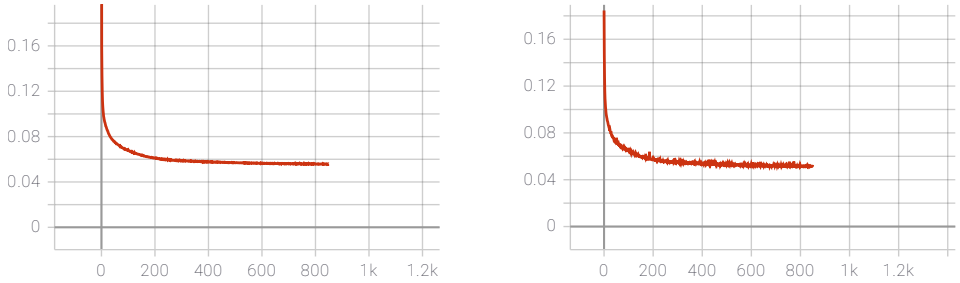


Figure 52: Discrete confusion matrix for the GCN-4 + DropEdge model with $p = 0.10$ on the unseen templates subset.

Figure 53: Normalized confusion matrix for the GCN-4 + DropEdge model with $p = 0.10$ on the unseen templates subset.



(a) Train loss (b) Validation loss

Figure 54: The train (a) and validation (b) loss curves for the GCN-4 + DropEdge model with $p = 0.10$. The y-axis shows the calculated loss while the x-axis shows at which epoch the loss has been calculated.

Results for GCN-4 + DropEdge, $p = 0.20$

Table 26: The corresponding performance metrics for each entity for the GCN-4 + DropEdge model with $p = 0.20$, evaluated on the test set.

Entity	F_1	Precision	Recall
amountCurrency	0.5918	0.7739	0.4791
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.4312	0.6848	0.3147
amountRndDiff	0.0034	1.0000	0.0017
amountTot	0.6868	0.7749	0.6168
amountVat	0.5667	0.7137	0.4699
dueDate	0.6610	0.7448	0.5941
invDate	0.6887	0.7545	0.6334
invNo	0.5129	0.6954	0.4063
ocrNo	0.4130	0.6432	0.3042
orderNo	0.0114	1.0000	0.0057
referenceName	0.0000	0.0000	0.0000
type	0.8114	0.8087	0.8142
undefined	0.9911	0.9866	0.9956
Macro average	0.5069	0.6843	0.4025
Validation loss			0.0580

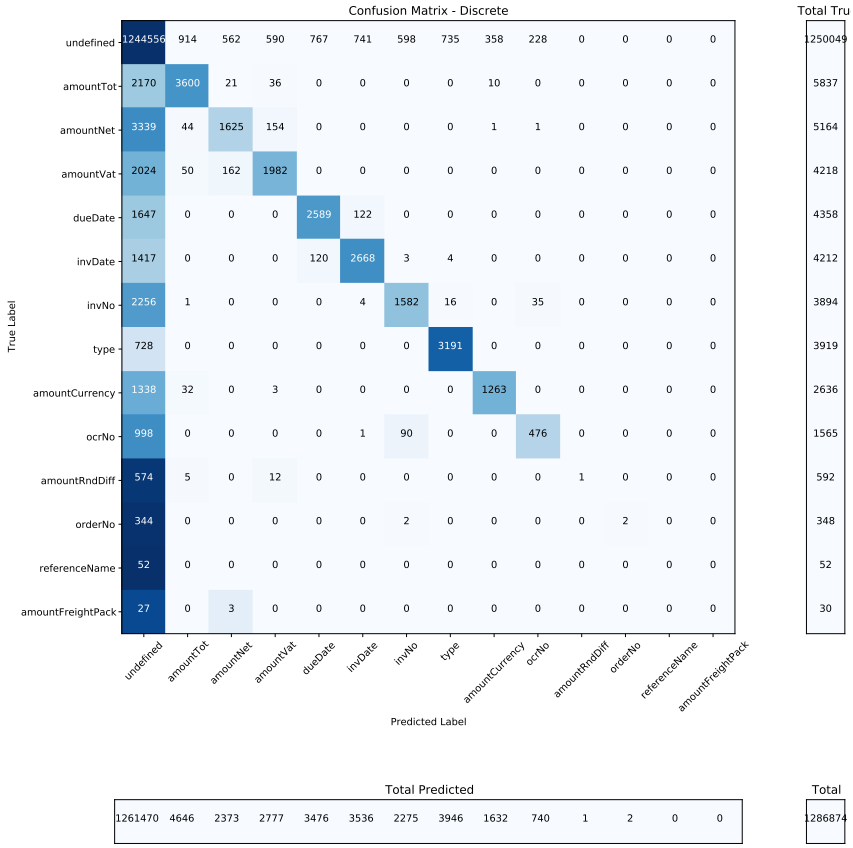


Figure 55: Discrete confusion matrix for the GCN-4 + DropEdge model with $p = 0.20$ on the whole test set.

Figure 56: Normalized confusion matrix for the GCN-4 + DropEdge model with $p = 0.20$ on the whole test set.

Table 27: The corresponding per-class performance metrics for each entity for the GCN-4 + DropEdge model with $p = 0.20$, calculated on invoices with unseen templates.

Entity	F_1	Precision	Recall
amountCurrency	0.5392	0.7478	0.4216
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.3854	0.7287	0.2620
amountRndDiff	0.0000	0.0000	0.0000
amountTot	0.6633	0.8380	0.5489
amountVat	0.5824	0.7609	0.4717
dueDate	0.6222	0.7147	0.5508
invDate	0.7131	0.7711	0.6632
invNo	0.4727	0.7128	0.3536
ocrNo	0.4423	0.5897	0.3538
orderNo	0.0000	0.0000	0.0000
referenceName	0.0000	0.0000	0.0000
type	0.8136	0.8348	0.7934
undefined	0.9883	0.9816	0.9952
Macro average	0.4536	0.5486	0.3867

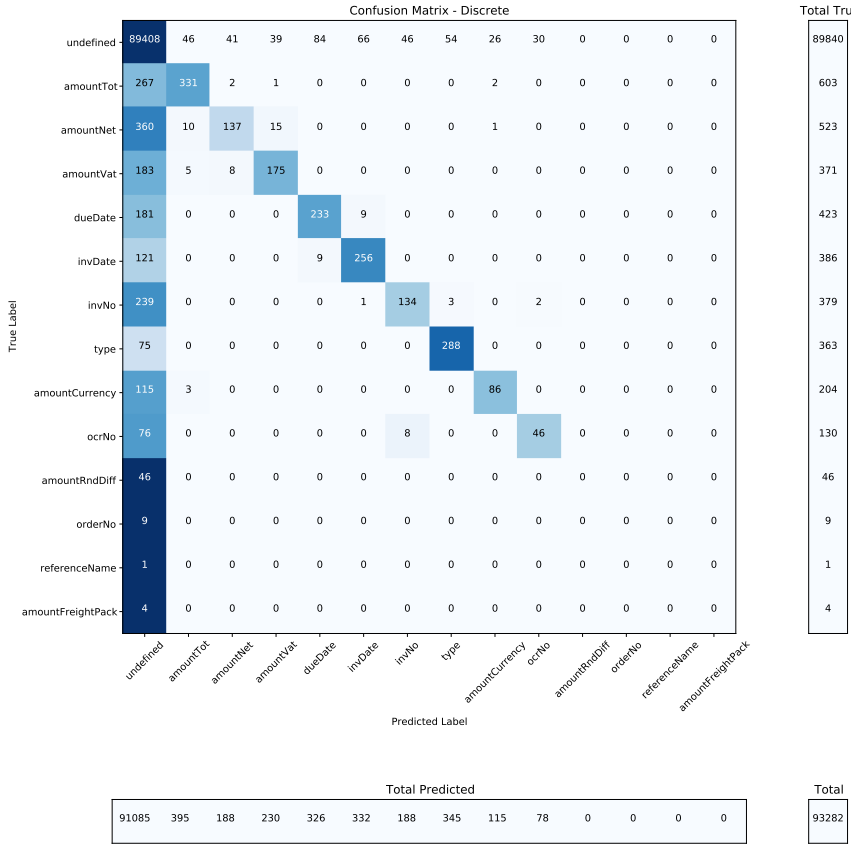
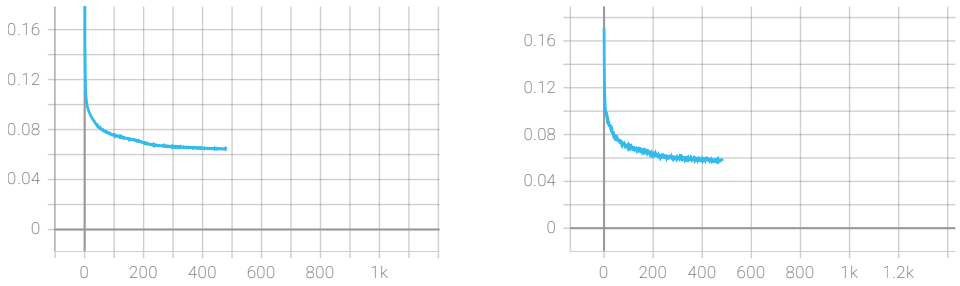


Figure 57: Discrete confusion matrix for the GCN-4 + DropEdge model with $p = 0.20$ on the unseen templates subset.

Figure 58: Normalized confusion matrix for the GCN-4 + DropEdge model with $p = 0.20$ on the unseen templates subset.



(a) Train loss

(b) Validation loss

Figure 59: The train (a) and validation (b) loss curves for the GCN-4 + DropEdge model with $p = 0.20$. The y-axis shows the calculated loss while the x-axis shows at which epoch the loss has been calculated.

Results for GCN-4 + DropEdge, $p = 0.50$

Table 28: The corresponding performance metrics for each entity for the GCN-4 + DropEdge model with $p = 0.50$, evaluated on the test set.

Entity	F_1	Precision	Recall
amountCurrency	0.1415	0.6306	0.0797
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.1411	0.5691	0.0806
amountRndDiff	0.0000	0.0000	0.0000
amountTot	0.5474	0.7077	0.4463
amountVat	0.2856	0.6878	0.1802
dueDate	0.4835	0.6625	0.3807
invDate	0.6113	0.6379	0.5869
invNo	0.1166	0.5510	0.0652
ocrNo	0.0983	0.6748	0.0530
orderNo	0.0000	0.0000	0.0000
referenceName	0.0000	0.0000	0.0000
type	0.6441	0.8038	0.5374
undefined	0.9884	0.9800	0.9968
Macro average	0.3259	0.4932	0.2433
Validation loss			0.0750

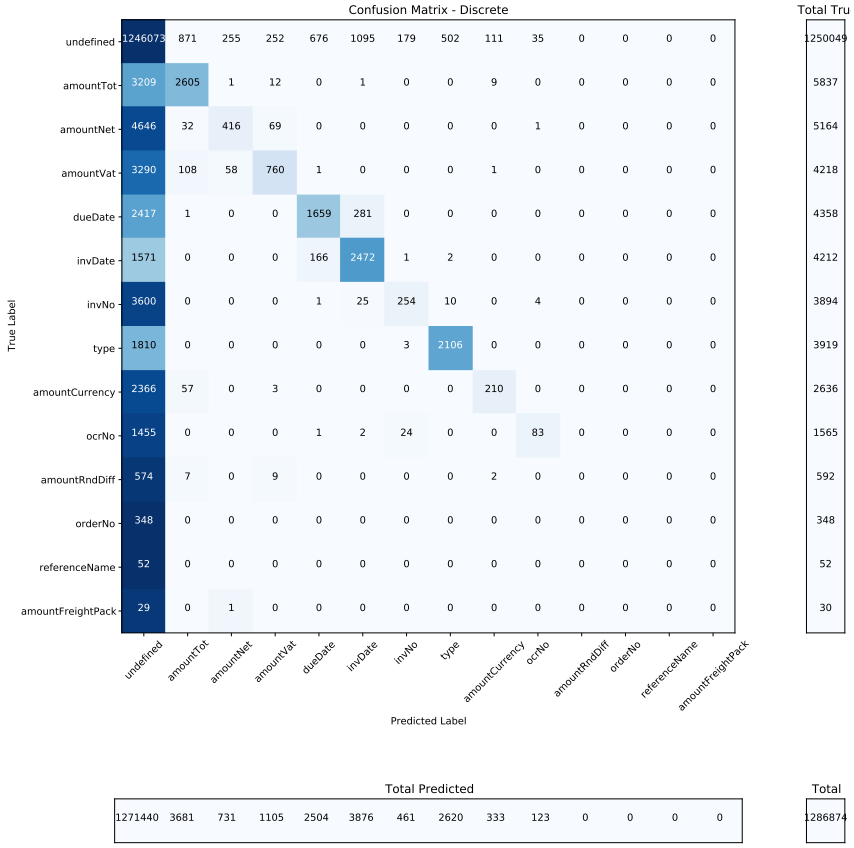


Figure 60: Discrete confusion matrix for the GCN-4 + DropEdge model with $p = 0.50$ on the whole test set.

Figure 61: Normalized confusion matrix for the GCN-4 + DropEdge model with $p = 0.50$ on the whole test set.

Table 29: The corresponding per-class performance metrics for each entity for the GCN-4 + DropEdge model with $p = 0.50$, calculated on invoices with unseen templates.

Entity	F_1	Precision	Recall
amountCurrency	0.1459	0.5862	0.0833
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.0977	0.5600	0.0535
amountRndDiff	0.0000	0.0000	0.0000
amountTot	0.5082	0.7420	0.3864
amountVat	0.2838	0.8000	0.1725
dueDate	0.4604	0.6481	0.3570
invDate	0.6630	0.7003	0.6295
invNo	0.1274	0.6000	0.0712
ocrNo	0.2252	0.8095	0.1308
orderNo	0.0000	0.0000	0.0000
referenceName	0.0000	0.0000	0.0000
type	0.6623	0.8299	0.5510
undefined	0.9852	0.9740	0.9966
Macro average	0.3328	0.5179	0.2451

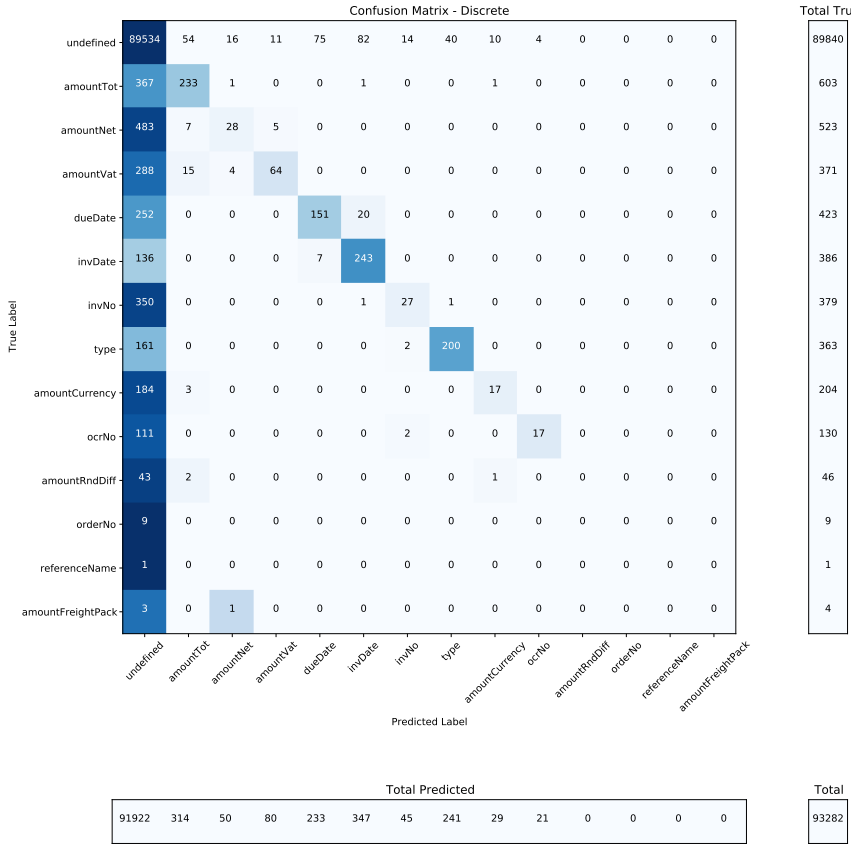


Figure 62: Discrete confusion matrix for the GCN-4 + DropEdge model with $p = 0.50$ on the unseen templates subset.

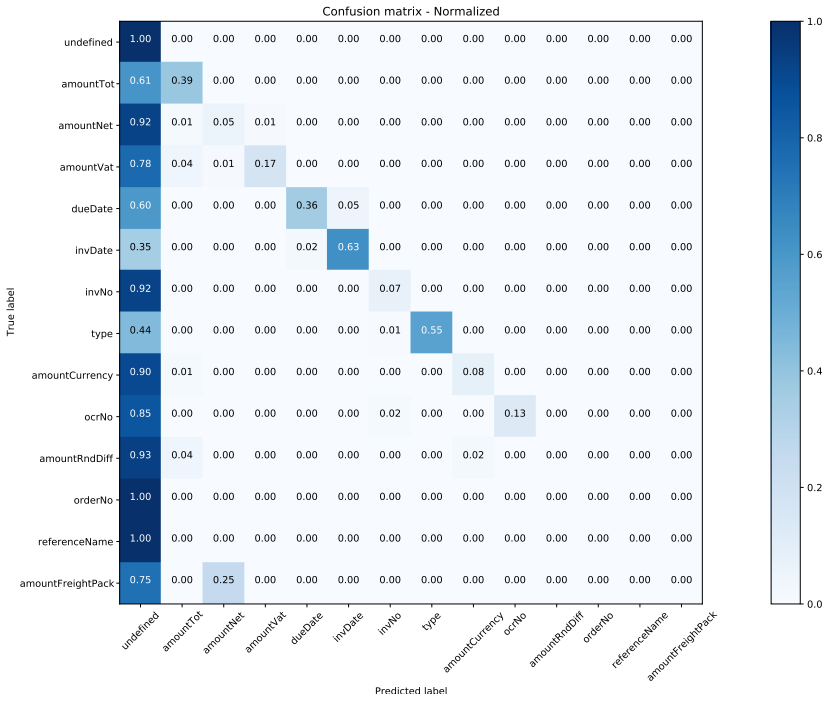
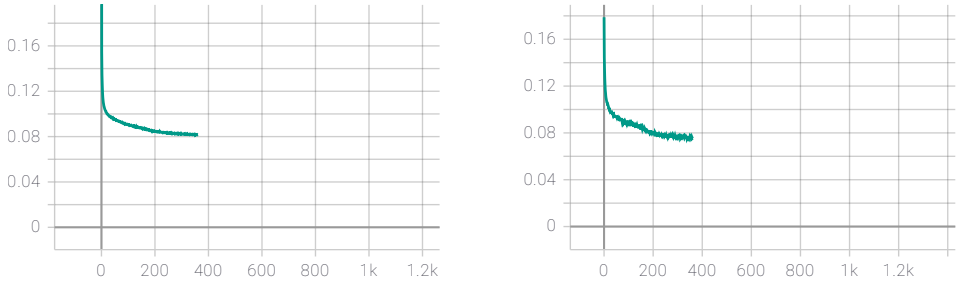


Figure 63: Normalized confusion matrix for the GCN-4 + DropEdge model with $p = 0.50$ on the unseen templates subset.



(a) Train loss (b) Validation loss

Figure 64: The train (a) and validation (b) loss curves for the GCN-4 + DropEdge model with $p = 0.50$. The y-axis shows the calculated loss while the x-axis shows at which epoch the loss has been calculated.

Results for GCN-4 + DropEdge, $p = 0.80$

Table 30: The corresponding performance metrics for each entity for the GCN-4 + DropEdge model with $p = 0.80$, evaluated on the test set.

Entity	F_1	Precision	Recall
amountCurrency	0.0000	0.0000	0.0000
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.0000	0.0000	0.0000
amountRndDiff	0.0000	0.0000	0.0000
amountTot	0.0492	0.6866	0.0255
amountVat	0.0000	0.0000	0.0000
dueDate	0.0000	0.0000	0.0000
invDate	0.0205	0.5366	0.0104
invNo	0.0000	0.0000	0.0000
ocrNo	0.0000	0.0000	0.0000
orderNo	0.0000	0.0000	0.0000
referenceName	0.0000	0.0000	0.0000
type	0.0196	0.6724	0.0100
undefined	0.9855	0.9716	0.9999
Macro average	0.1095	0.2048	0.0747
Validation loss			0.1020

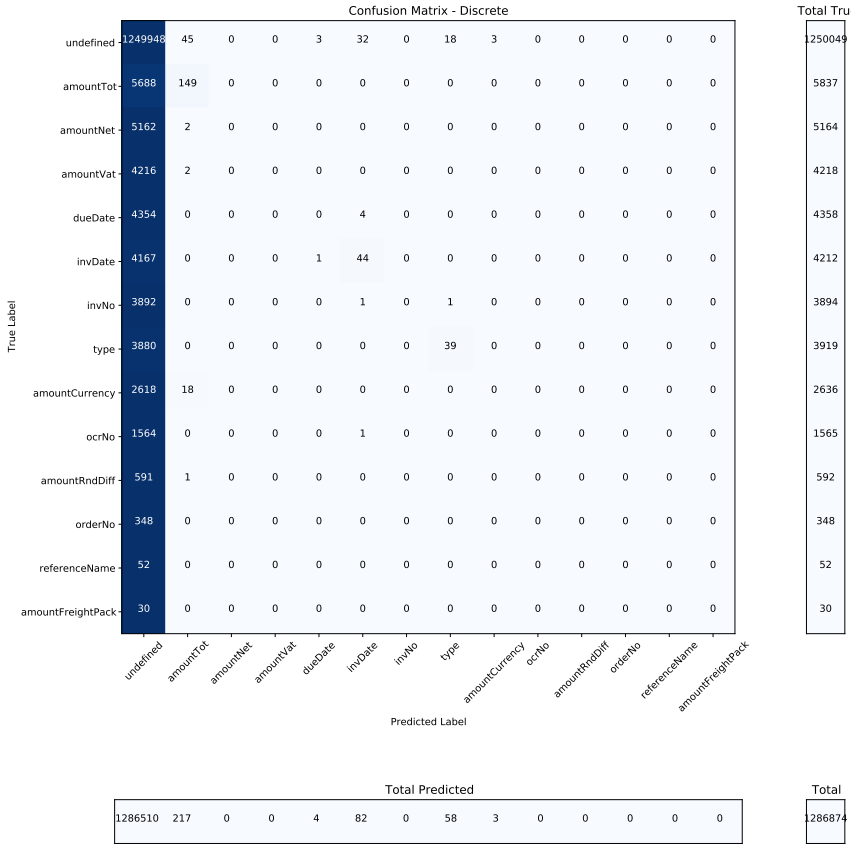


Figure 65: Discrete confusion matrix for the GCN-4 + DropEdge model with $p = 0.80$ on the whole test set.

Figure 66: Normalized confusion matrix for the GCN-4 + DropEdge model with $p = 0.80$ on the whole test set.

Table 31: The corresponding per-class performance metrics for each entity for the GCN-4 + DropEdge model with $p = 0.80$, calculated on invoices with unseen templates.

Entity	F_1	Precision	Recall
amountCurrency	0.0000	0.0000	0.0000
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.0000	0.0000	0.0000
amountRndDiff	0.0000	0.0000	0.0000
amountTot	0.0546	0.8500	0.0282
amountVat	0.0000	0.0000	0.0000
dueDate	0.0000	0.0000	0.0000
invDate	0.0051	0.2500	0.0026
invNo	0.0000	0.0000	0.0000
ocrNo	0.0000	0.0000	0.0000
orderNo	0.0000	0.0000	0.0000
referenceName	0.0000	0.0000	0.0000
type	0.0323	0.7500	0.0165
undefined	0.9813	0.9634	0.9999
Macro average	0.1090	0.2010	0.0748

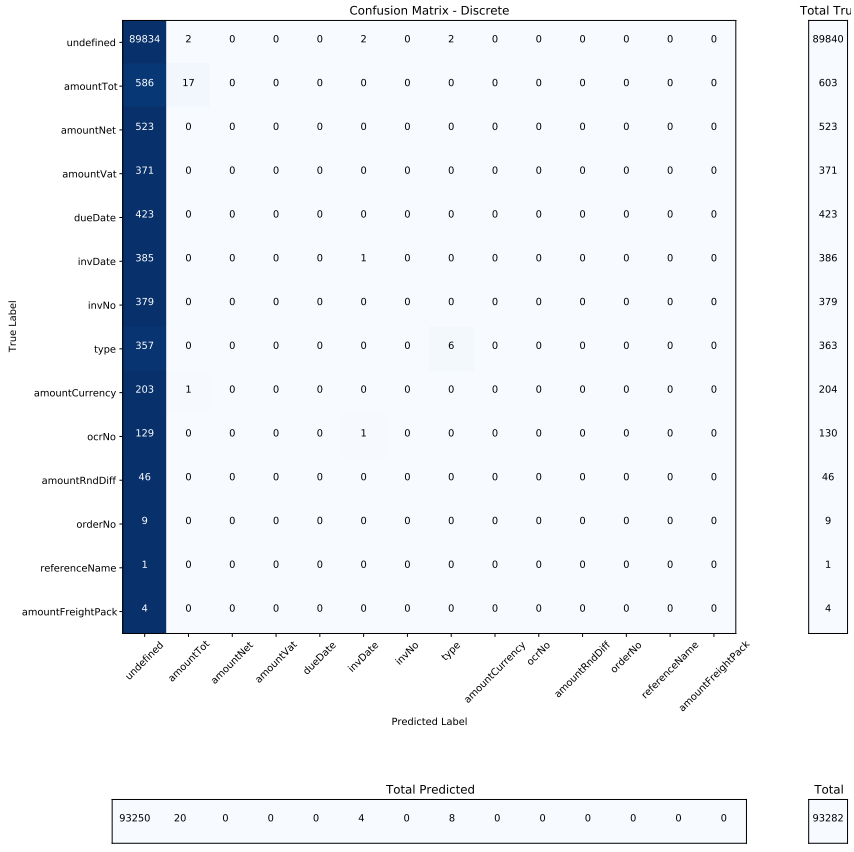
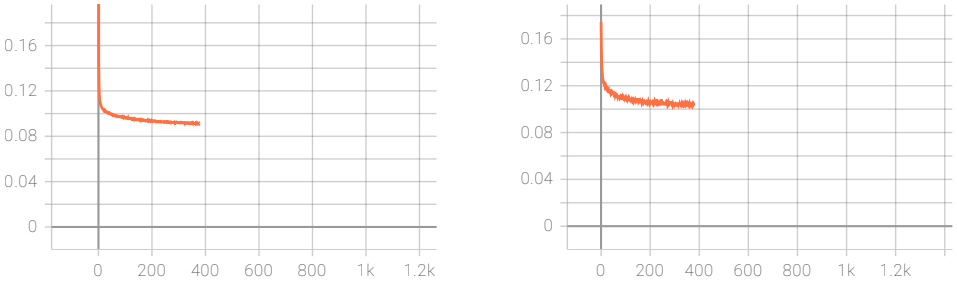


Figure 67: Discrete confusion matrix for the GCN-4 + DropEdge model with $p = 0.80$ on the unseen templates subset.

Figure 68: Normalized confusion matrix for the GCN-4 + DropEdge model with $p = 0.80$ on the unseen templates subset.



(a) Train loss

(b) Validation loss

Figure 69: The train (a) and validation (b) loss curves for the GCN-4 + DropEdge model with $p = 0.80$. The y-axis shows the calculated loss while the x-axis shows at which epoch the loss has been calculated.

ChebNet + DropEdge, $p = 0.10$

Table 32: The corresponding performance metrics for each entity for the ChebNet + DropEdge model with $p = 0.10$, evaluated on the test set.

Entity	F_1	Precision	Recall
amountCurrency	0.7212	0.8090	0.6506
amountFreightPack	0.0645	1.0000	0.0333
amountNet	0.7361	0.8082	0.6758
amountRndDiff	0.2979	0.7000	0.1892
amountTot	0.8126	0.8167	0.8085
amountVat	0.8025	0.8309	0.7760
dueDate	0.8069	0.8172	0.7969
invDate	0.8592	0.8564	0.8621
invNo	0.8186	0.8438	0.7948
ocrNo	0.6920	0.7151	0.6703
orderNo	0.5236	0.8313	0.3822
referenceName	0.2462	0.6154	0.1538
type	0.9089	0.9028	0.9150
undefined	0.9946	0.9938	0.9957
Macro average	0.7088	0.8243	0.6217
Validation loss			0.0330

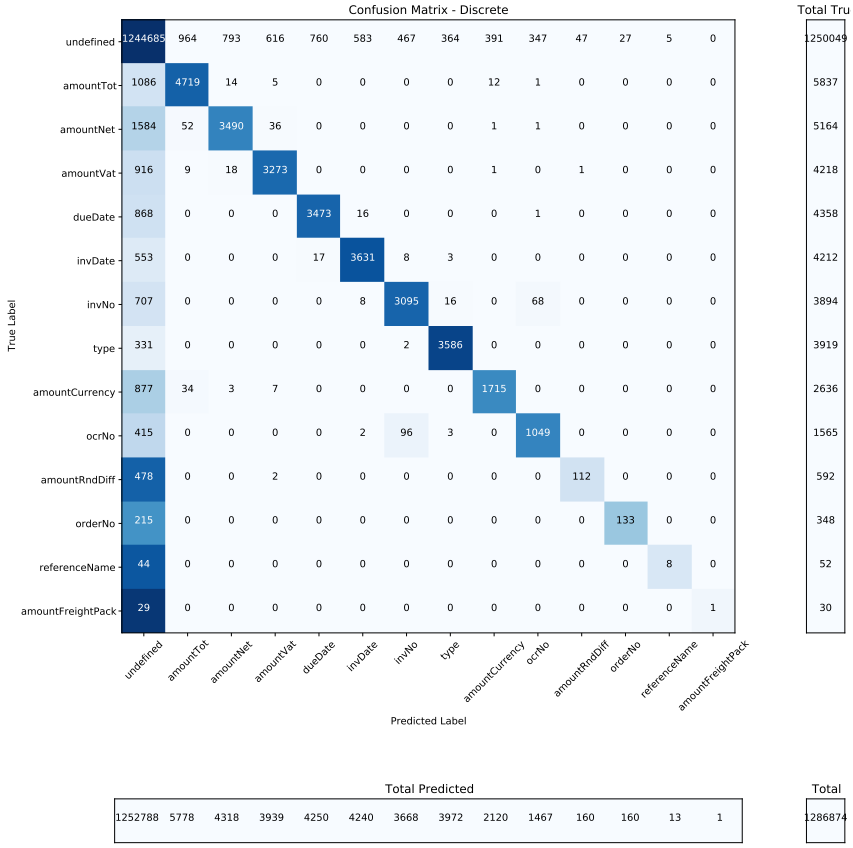


Figure 70: Discrete confusion matrix for the ChebNet + DropEdge model with $p = 0.10$ on the whole test set.

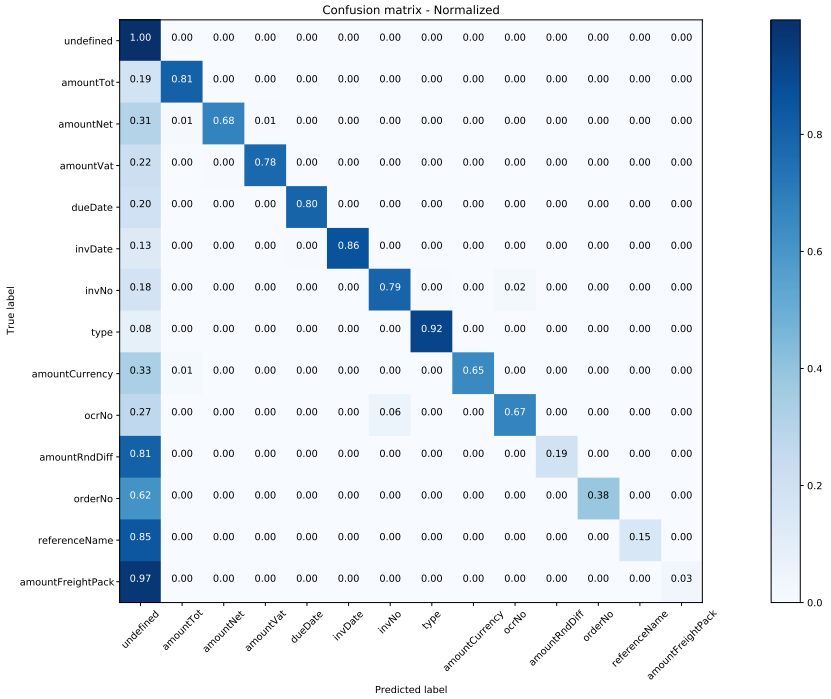


Figure 71: Normalized confusion matrix for the ChebNet + DropEdge model with $p = 0.10$ on the whole test set.

Table 33: The corresponding per-class performance metrics for each entity for the ChebNet + DropEdge model with $p = 0.10$, calculated on invoices with unseen templates.

Entity	F_1	Precision	Recall
amountCurrency	0.6140	0.7609	0.5147
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.6896	0.8206	0.5946
amountRndDiff	0.0392	0.2000	0.0217
amountTot	0.7892	0.8352	0.7479
amountVat	0.7989	0.8485	0.7547
dueDate	0.7494	0.8065	0.6998
invDate	0.8258	0.8226	0.8290
invNo	0.7427	0.7853	0.7045
ocrNo	0.6332	0.6357	0.6308
orderNo	0.1818	0.5000	0.1111
referenceName	0.0000	0.0000	0.0000
type	0.8631	0.8754	0.8512
undefined	0.9919	0.9893	0.9944
Macro average	0.5789	0.6343	0.5325

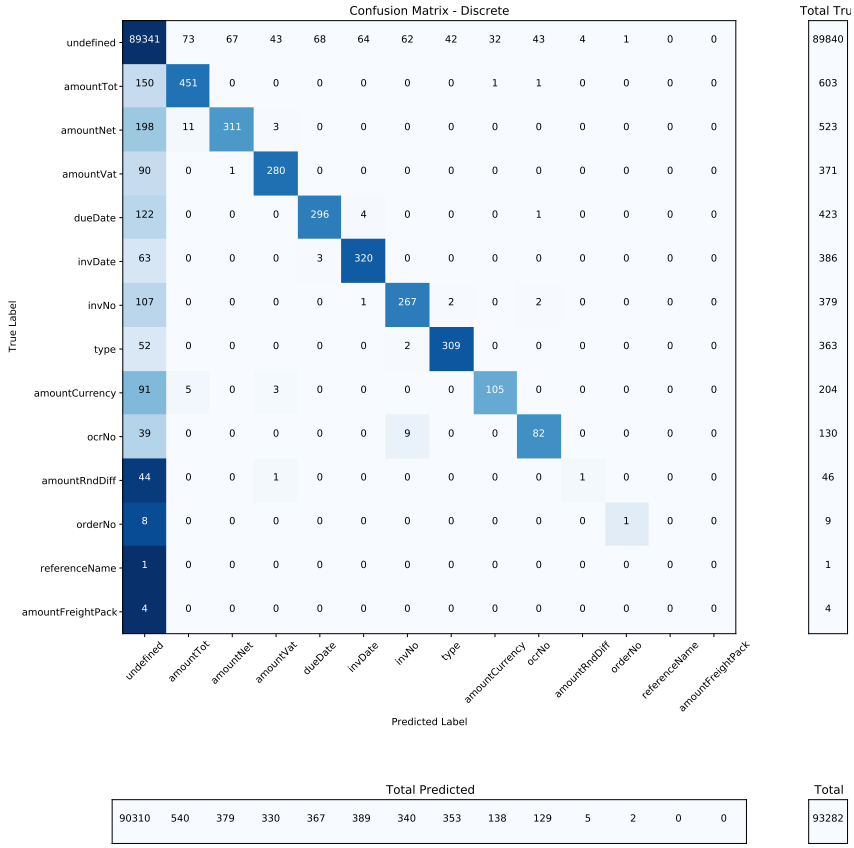


Figure 72: Discrete confusion matrix for the ChebNet + DropEdge model with $p = 0.10$ on the unseen templates subset.

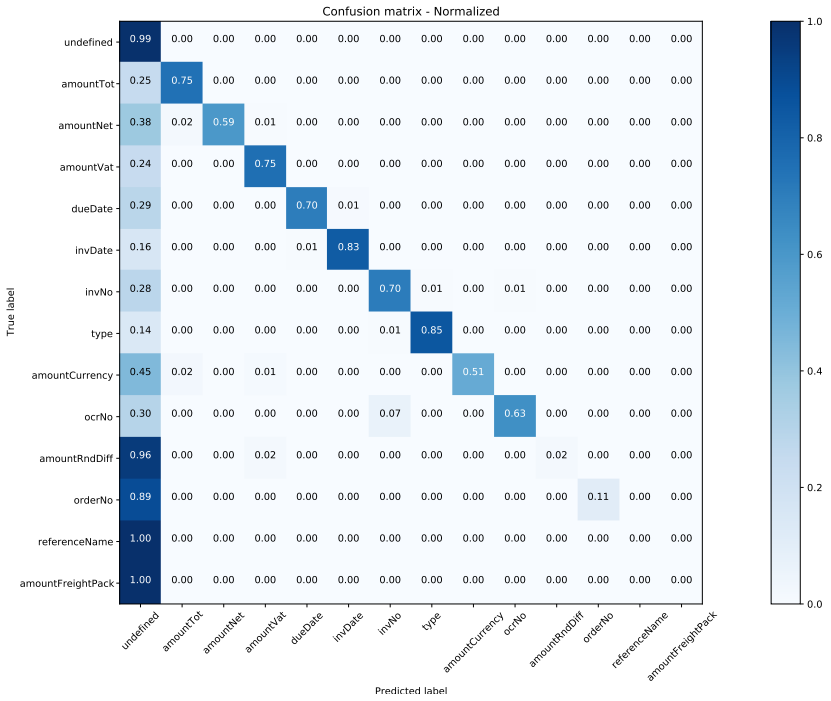
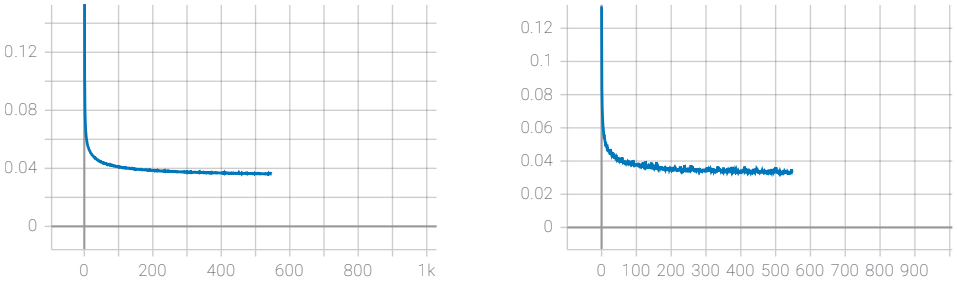


Figure 73: Normalized confusion matrix for the ChebNet + DropEdge model with $p = 0.10$ on the unseen templates subset.



(a) Train loss (b) Validation loss

Figure 74: The train (a) and validation (b) loss curves for the ChebNet + DropEdge model with $p = 0.10$. The y-axis shows the calculated loss while the x-axis shows at which epoch the loss has been calculated.

ChebNet + DropEdge, $p = 0.20$

Table 34: The corresponding performance metrics for each entity for the ChebNet + DropEdge model with $p = 0.20$, evaluated on the test set.

Entity	F_1	Precision	Recall
amountCurrency	0.7396	0.7918	0.6939
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.7261	0.7919	0.6704
amountRndDiff	0.2181	0.7238	0.1284
amountTot	0.8047	0.8234	0.7869
amountVat	0.7930	0.8397	0.7513
dueDate	0.7955	0.8168	0.7754
invDate	0.8396	0.8354	0.8438
invNo	0.8040	0.8337	0.7763
ocrNo	0.6896	0.7227	0.6594
orderNo	0.4307	0.8347	0.2902
referenceName	0.2000	0.7500	0.1154
type	0.9030	0.8886	0.9178
undefined	0.9944	0.9932	0.9956
Macro average	0.6710	0.7604	0.6003
Validation loss			0.0350

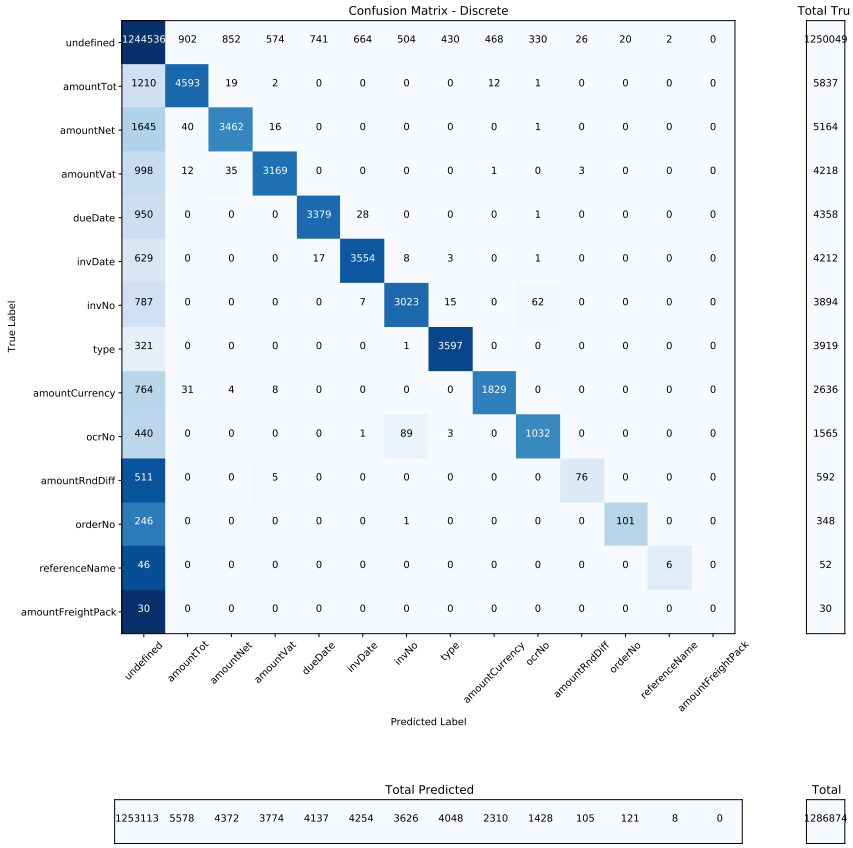


Figure 75: Discrete confusion matrix for the ChebNet + DropEdge model with $p = 0.20$ on the whole test set.

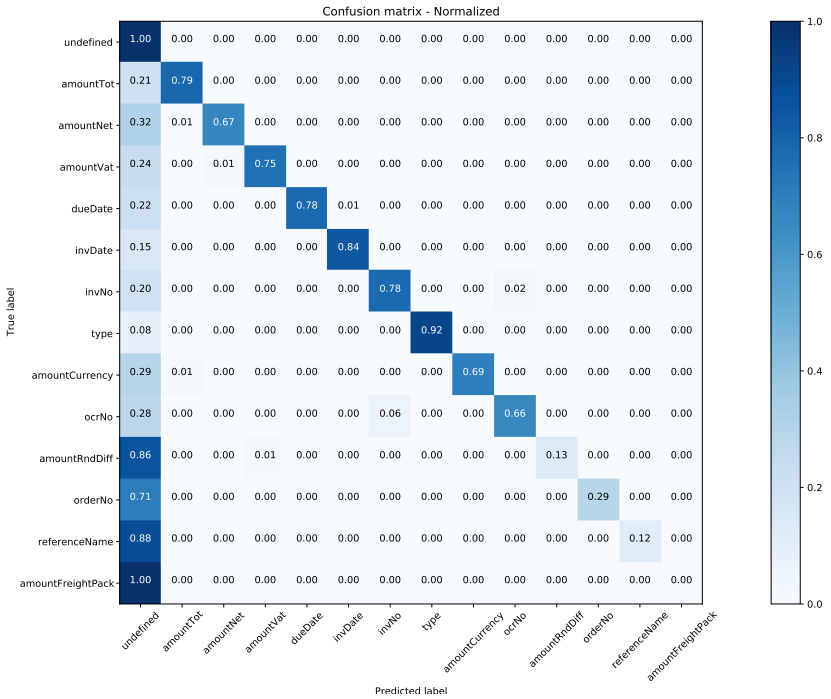


Figure 76: Normalized confusion matrix for the ChebNet + DropEdge model with $p = 0.20$ on the whole test set.

Table 35: The corresponding per-class performance metrics for each entity for the ChebNet + DropEdge model with $p = 0.20$, calculated on invoices with unseen templates.

Entity	F_1	Precision	Recall
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.6858	0.8099	0.5946
amountRndDiff	0.0408	0.3333	0.0217
amountTot	0.7635	0.8376	0.7015
amountVat	0.7959	0.8667	0.7358
dueDate	0.7165	0.7711	0.6690
invDate	0.7979	0.8031	0.7927
invNo	0.7374	0.8082	0.6781
ocrNo	0.6748	0.7155	0.6385
orderNo	0.0000	0.0000	0.0000
referenceName	0.0000	0.0000	0.0000
type	0.8719	0.8625	0.8815
undefined	0.9915	0.9887	0.9944
Macro average	0.5633	0.6123	0.5215

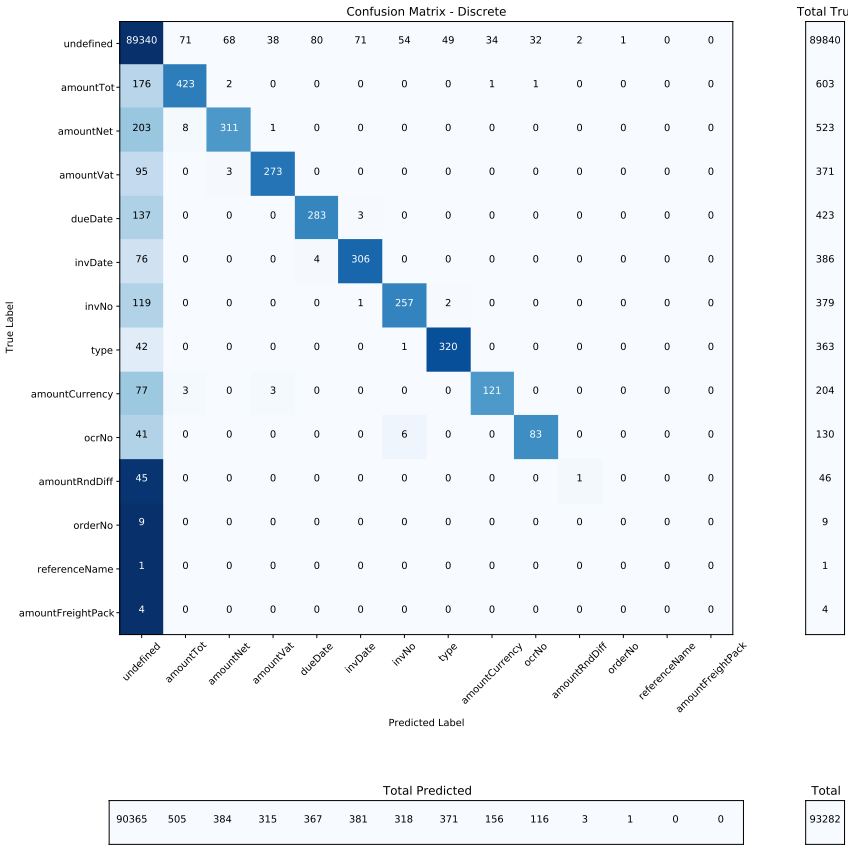
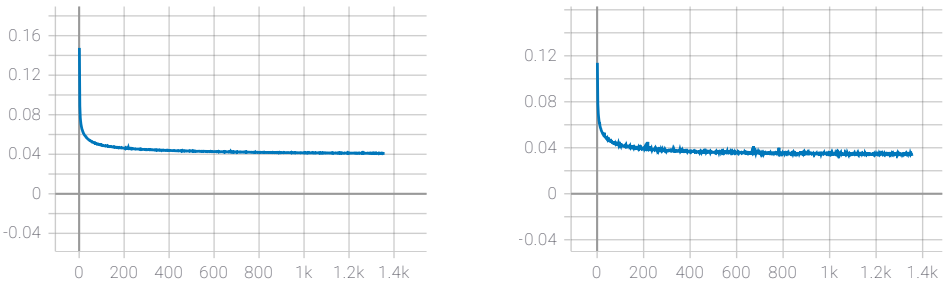


Figure 77: Discrete confusion matrix for the ChebNet + DropEdge model with $p = 0.20$ on the unseen templates subset.

Figure 78: Normalized confusion matrix for the ChebNet + DropEdge model with $p = 0.20$ on the unseen templates subset.



(a) Train loss

(b) Validation loss

Figure 79: The train (a) and validation (b) loss curves for the ChebNet + DropEdge model with $p = 0.20$. The y-axis shows the calculated loss while the x-axis shows at which epoch the loss has been calculated.

ChebNet + DropEdge, $p = 0.50$

Table 36: The corresponding performance metrics for each entity for the ChebNet + DropEdge model with $p = 0.50$, evaluated on the test set.

Entity	F_1	Precision	Recall
amountCurrency	0.6674	0.7850	0.5904
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.6443	0.7671	0.5554
amountRndDiff	0.0832	0.7879	0.0439
amountTot	0.7766	0.8129	0.7434
amountVat	0.7455	0.8066	0.6930
dueDate	0.7634	0.7909	0.7377
invDate	0.8004	0.7908	0.8103
invNo	0.7353	0.7759	0.6988
ocrNo	0.6275	0.6738	0.5872
orderNo	0.4381	0.7448	0.3103
referenceName	0.1818	0.4286	0.1154
type	0.8770	0.8403	0.9171
undefined	0.9932	0.9914	0.9950
Macro average	0.6253	0.7140	0.5563
Validation loss			0.0400

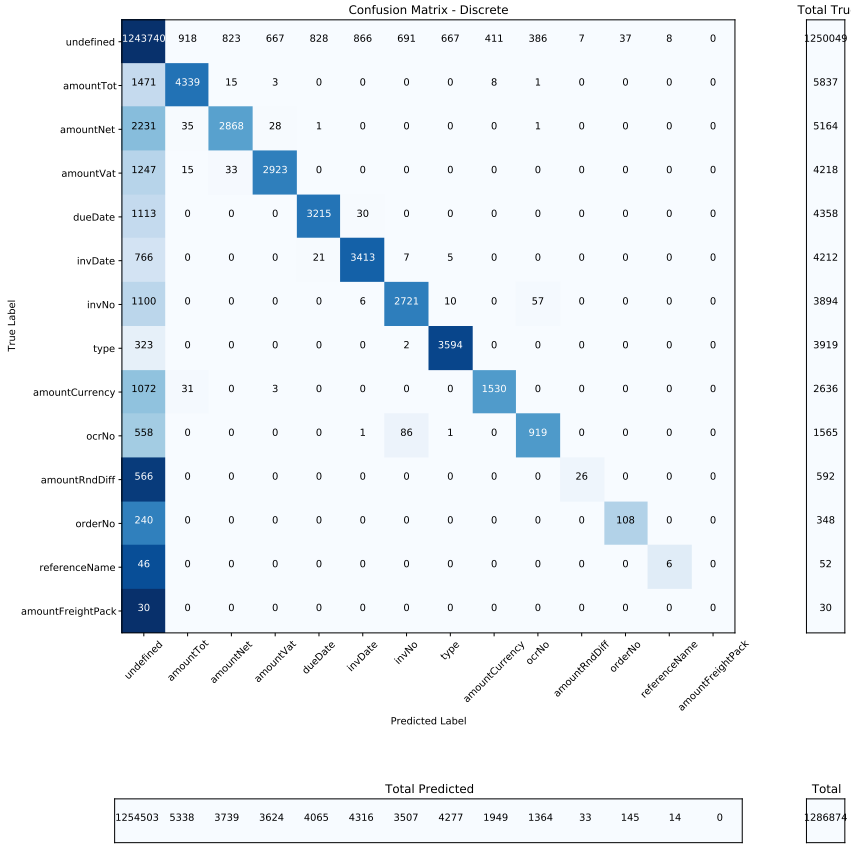


Figure 80: Discrete confusion matrix for the ChebNet + DropEdge model with $p = 0.50$ on the whole test set.

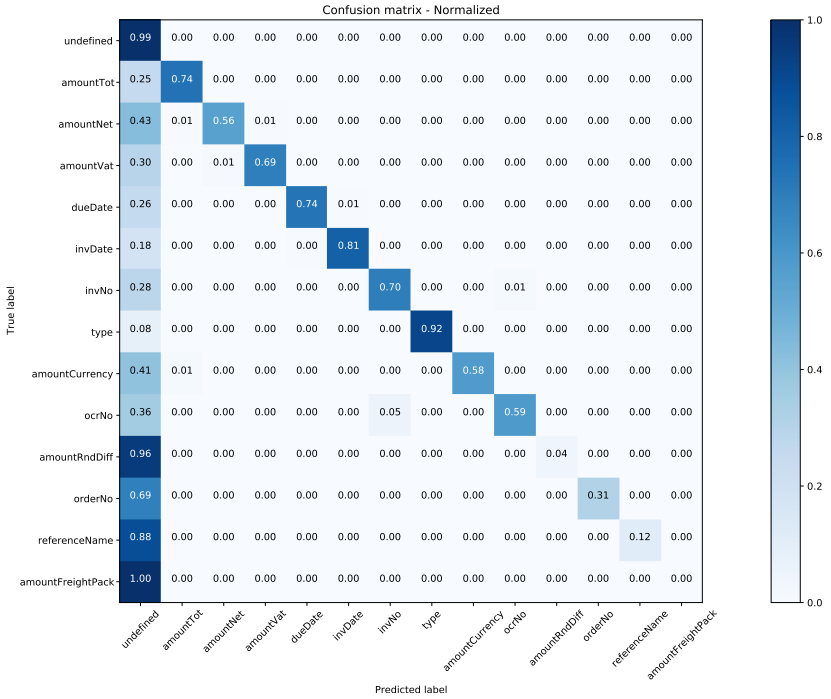


Figure 81: Normalized confusion matrix for the ChebNet + DropEdge model with $p = 0.50$ on the whole test set.

Table 37: The corresponding per-class performance metrics for each entity for the ChebNet + DropEdge model with $p = 0.50$, calculated on invoices with unseen templates.

Entity	F_1	Precision	Recall
amountCurrency	0.6048	0.7769	0.4951
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.6141	0.8185	0.4914
amountRndDiff	0.0000	0.0000	0.0000
amountTot	0.7373	0.8488	0.6517
amountVat	0.7615	0.8454	0.6927
dueDate	0.6978	0.7730	0.6359
invDate	0.7713	0.7923	0.7513
invNo	0.6777	0.7847	0.5963
ocrNo	0.5680	0.5917	0.5462
orderNo	0.0000	0.0000	0.0000
referenceName	0.0000	0.0000	0.0000
type	0.8765	0.8636	0.8898
undefined	0.9906	0.9866	0.9947
Macro average	0.5252	0.5773	0.4818

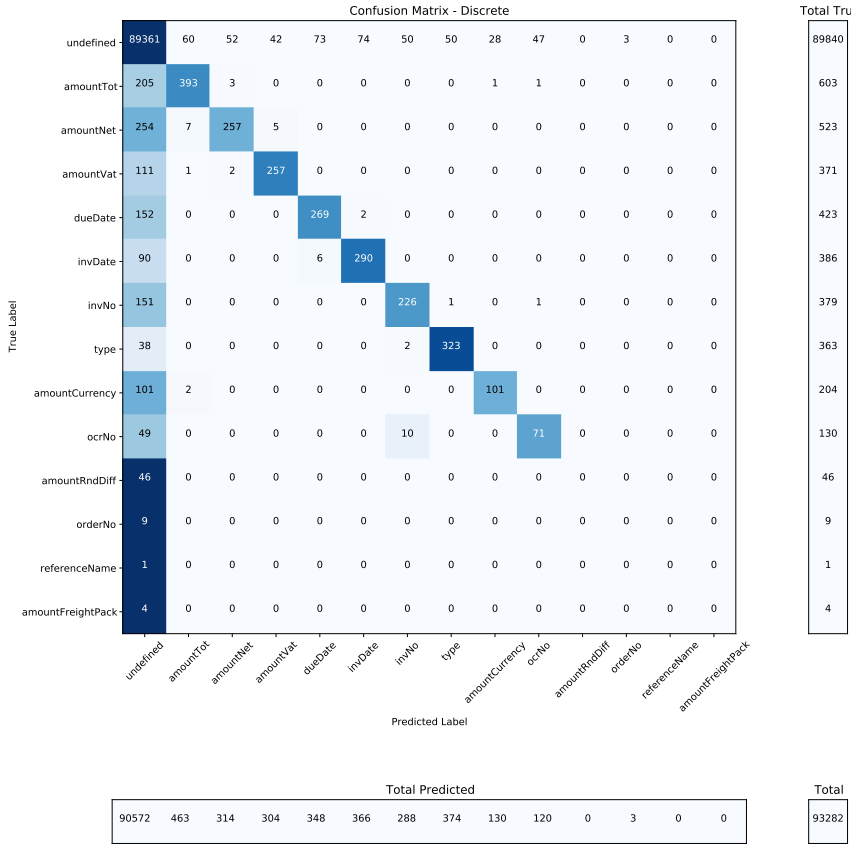
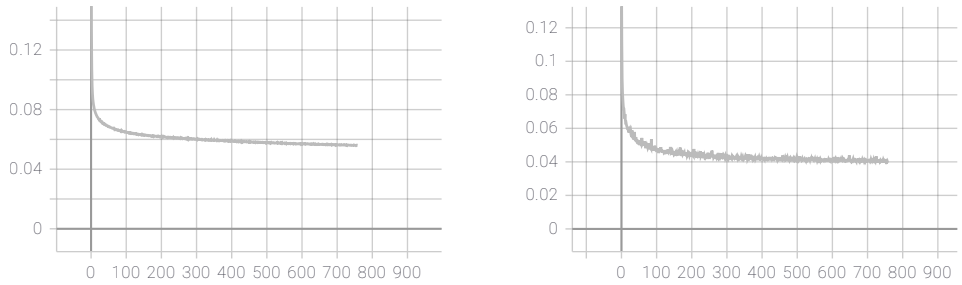


Figure 82: Discrete confusion matrix for the ChebNet + DropEdge model with $p = 0.50$ on the unseen templates subset.

Figure 83: Normalized confusion matrix for the ChebNet + DropEdge model with $p = 0.50$ on the unseen templates subset.



(a) Train loss (b) Validation loss

Figure 84: The train (a) and validation (b) loss curves for the ChebNet + DropEdge model with $p = 0.50$. The y-axis shows the calculated loss while the x-axis shows at which epoch the loss has been calculated.

ChebNet + DropEdge, $p = 0.80$

Table 38: The corresponding performance metrics for each entity for the ChebNet + DropEdge model with $p = 0.80$, evaluated on the test set.

Entity	F_1	Precision	Recall
amountCurrency	0.5603	0.7677	0.4412
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.4308	0.6595	0.3199
amountRndDiff	0.0361	0.6111	0.0186
amountTot	0.6781	0.7805	0.5995
amountVat	0.5730	0.7345	0.4697
dueDate	0.6050	0.7740	0.4966
invDate	0.6920	0.7314	0.6567
invNo	0.5763	0.6924	0.4936
ocrNo	0.3343	0.6561	0.2243
orderNo	0.0057	1.0000	0.0029
referenceName	0.0000	0.0000	0.0000
type	0.8312	0.7982	0.8671
undefined	0.9907	0.9862	0.9953
Macro average	0.4963	0.6565	0.3989
Validation loss			0.0550

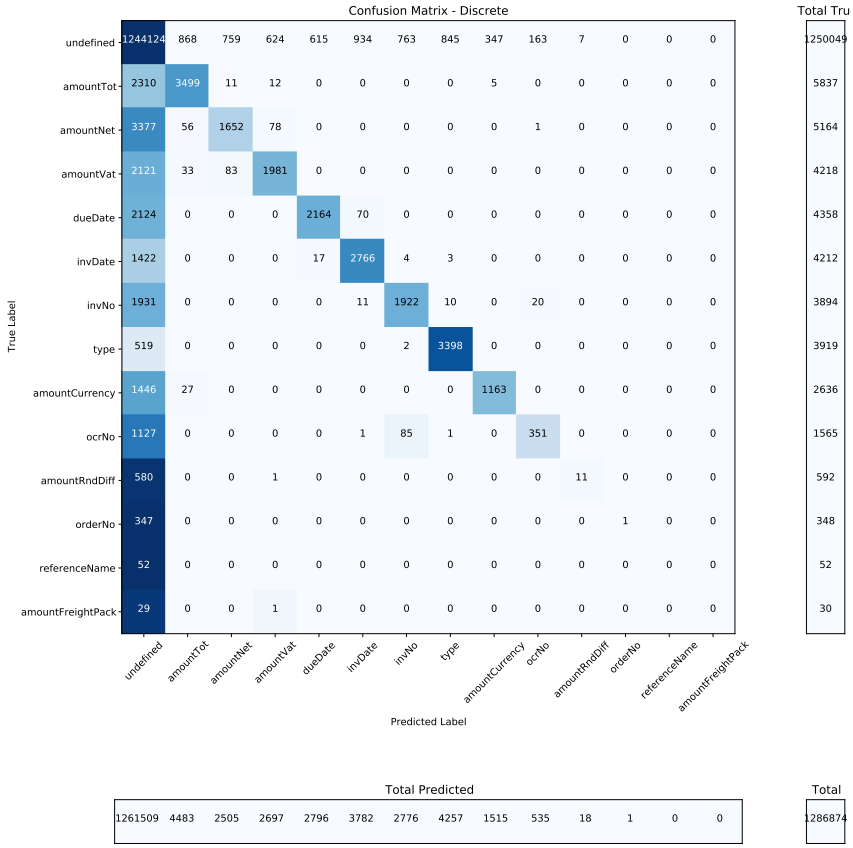


Figure 85: Discrete confusion matrix for the ChebNet + DropEdge model with $p = 0.80$ on the whole test set.

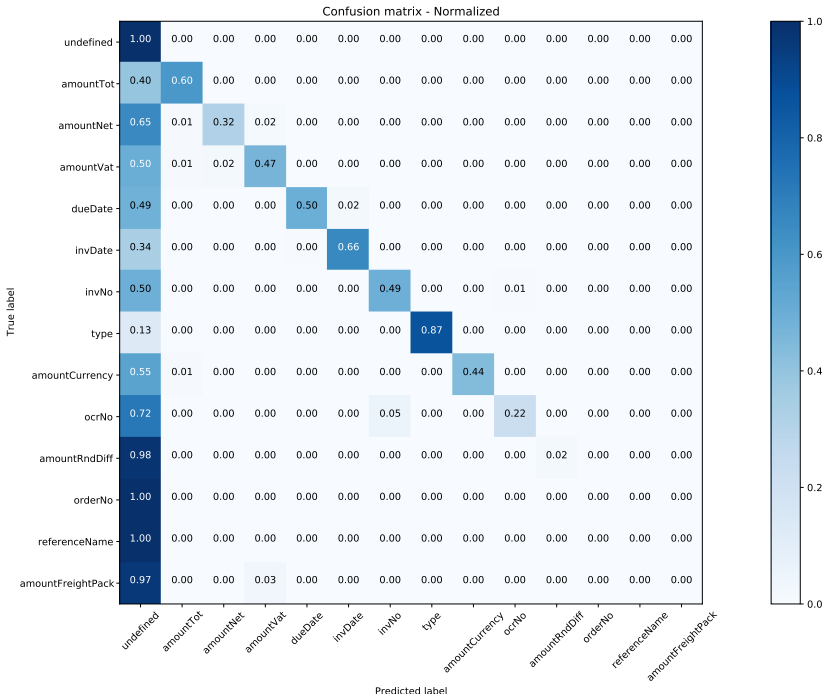


Figure 86: Normalized confusion matrix for the ChebNet + DropEdge model with $p = 0.80$ on the whole test set.

Table 39: The corresponding per-class performance metrics for each entity for the ChebNet + DropEdge model with $p = 0.80$, calculated on invoices with unseen templates.

Entity	F_1	Precision	Recall
amountCurrency	0.4916	0.7849	0.3578
amountFreightPack	0.0000	0.0000	0.0000
amountNet	0.4206	0.7243	0.2964
amountRndDiff	0.0000	0.0000	0.0000
amountTot	0.6385	0.8128	0.5257
amountVat	0.6014	0.8294	0.4717
dueDate	0.5281	0.7373	0.4113
invDate	0.7106	0.7551	0.6710
invNo	0.4871	0.6961	0.3747
ocrNo	0.4103	0.6154	0.3077
orderNo	0.0000	0.0000	0.0000
referenceName	0.0000	0.0000	0.0000
type	0.8610	0.8518	0.8705
undefined	0.9882	0.9810	0.9954
Macro average	0.4496	0.5563	0.3773

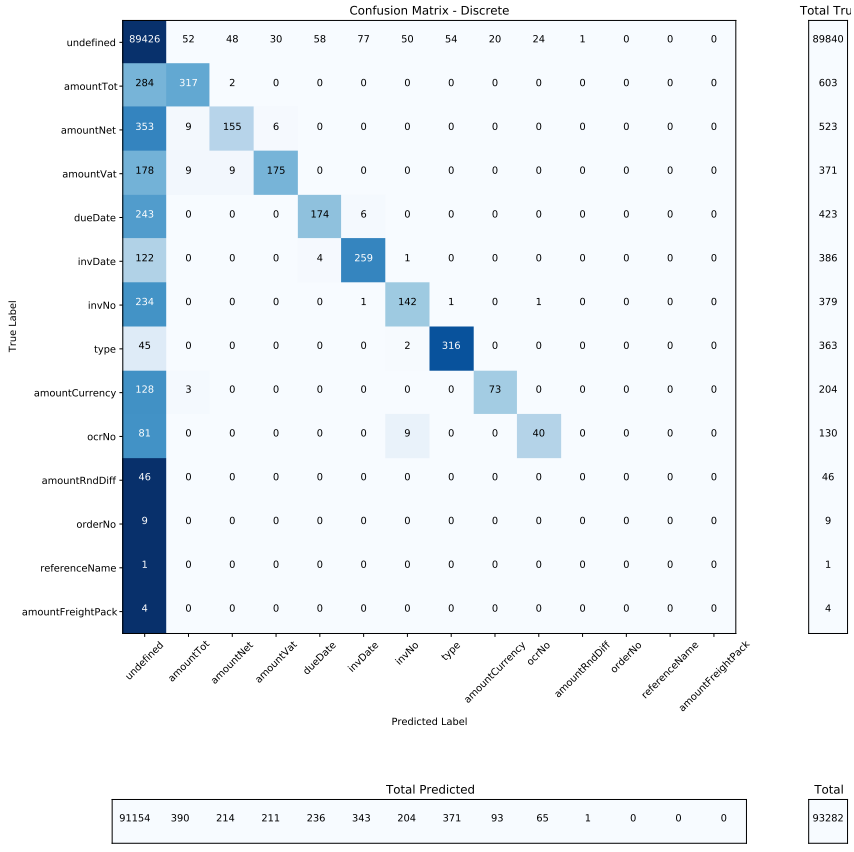


Figure 87: Discrete confusion matrix for the ChebNet + DropEdge model with $p = 0.80$ on the unseen templates subset.

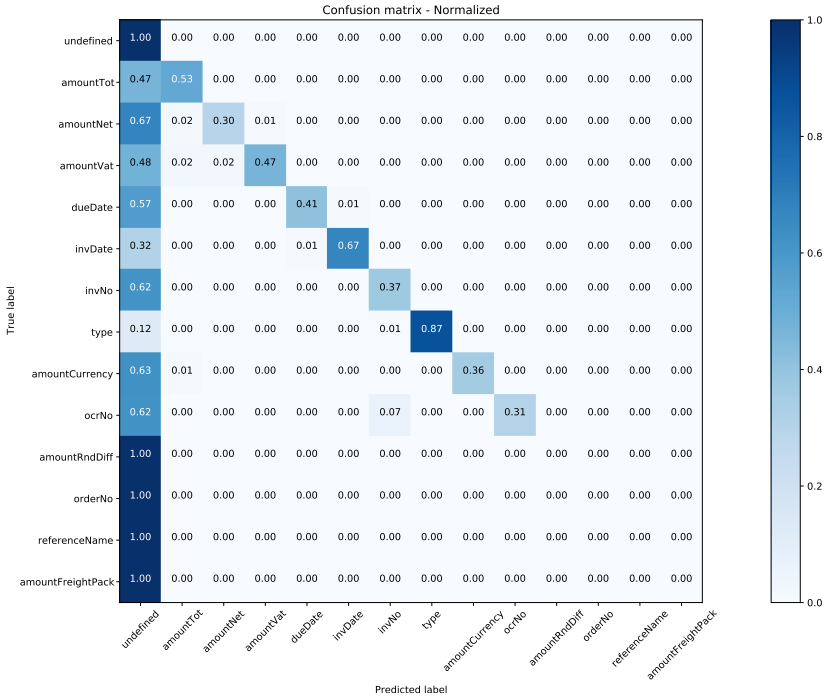
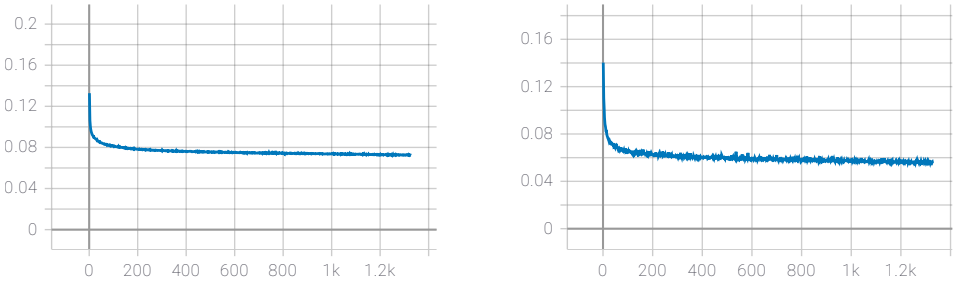


Figure 88: Normalized confusion matrix for the ChebNet + DropEdge model with $p = 0.80$ on the unseen templates subset.



(a) Train loss

(b) Validation loss

Figure 89: The train (a) and validation (b) loss curves for the ChebNet + DropEdge model with $p = 0.80$. The y-axis shows the calculated loss while the x-axis shows at which epoch the loss has been calculated.

Appendix B

Logit analysis

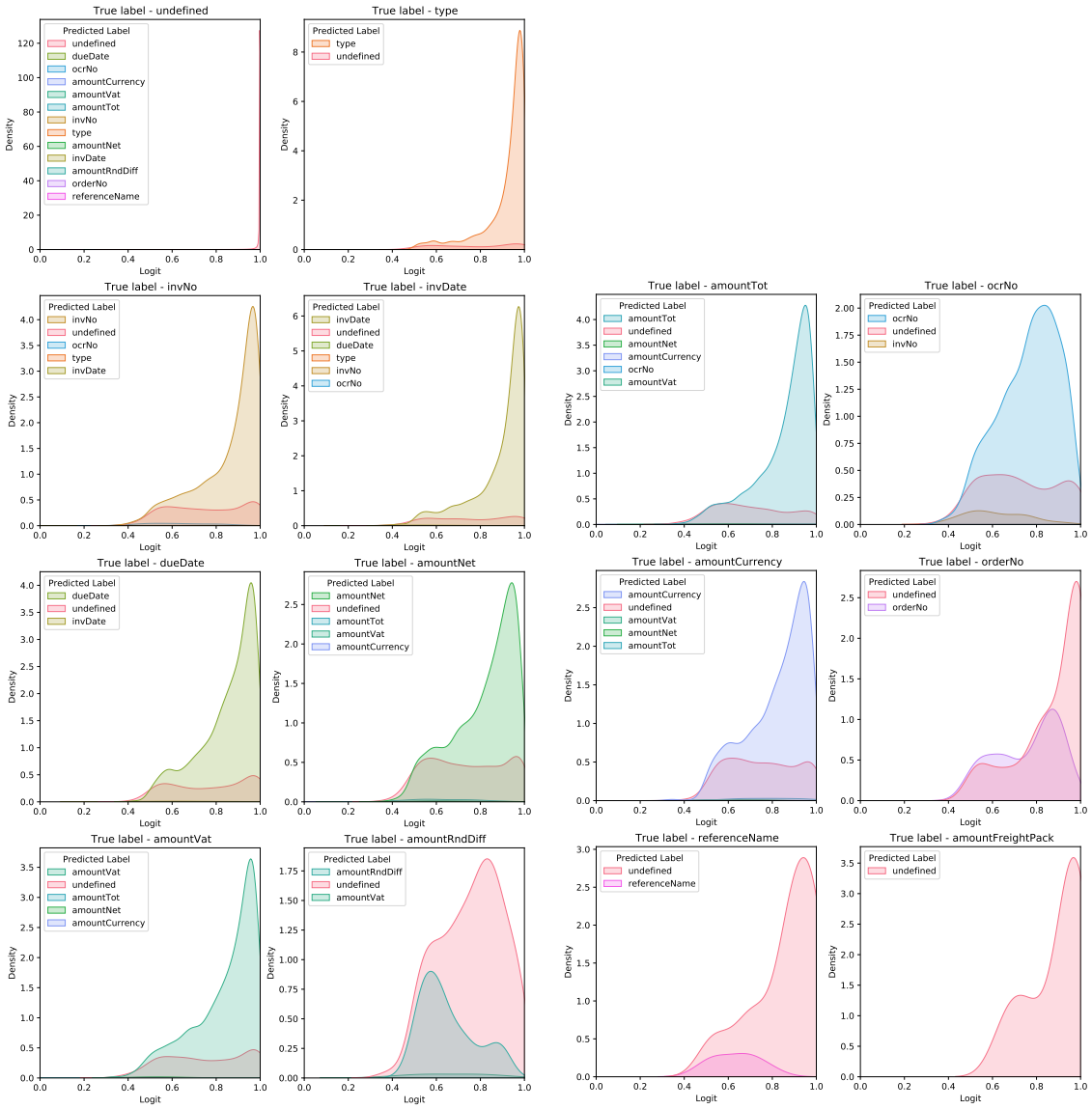


Figure 90: Kernel Density Estimation plots of predicted label logits, given a true label using the ChebNet model on the test set.