Predictive Analytics by Inferring Structure from Electronic Health Records

Master's Thesis
by

Luisa Sophie Werner

Supervisors:
Nabil Layaïda and Pierre Genevès
Tyrex Team - Inria

Project Period: 01/10/2019 – 30/04/2020

Abstract

Data science in the domain of healthcare recently registered considerable results in predicting medical outcomes based on Electronic Health Records. Especially deep learning methods enhance prediction performance by finding meaningful representations of input data. Latest studies show that Electronic Health Records explicitly or implicitly contain underlying causal relations which can be modeled as directed acyclic graphs. Recently, Graph Convolutional Transformer was proposed to learn the implicit graph structure of Electronic Health Records based on a method called attention. Then, Graph Convolutional Transformer exploits this structure to extract representations and conduct prediction tasks on Electronic Health Records. In this work, Graph Convolutional Transformer is applied on a large data set from Premier Healthcare Database in order to perform mortality prediction for patients who are admitted to hospitals. This work shows that Graph Convolutional Transformer leads to a state-of-the-art performance on the mortality prediction task in comparison to the applied baseline models. Furthermore, some possible extensions of Graph Convolutional Transformer are illustrated that have the potential to further improve the predictive performance of Graph Convolutional Transformer.

Acknowledgement

I would like to express my sincere gratitude to Pierre Genevès and Nabil Layaïda of the Tyrex Team of INRIA who co-supervised and this project. Furthermore, I also thank all members of the Tyrex Team who assisted me and gave me helpful feedback on my work.

Contents

List of Acronyms

AI Artificial Intelligence

Adam Adaptive Moment Estimation

API Application Programming Interface

AUC-PR Area Under the Precision Recall Curve

AUC-ROC Area Under the Receiver Operating Characteristic Curve

CNN Convolutional Neural Network

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

CuDNN CUDA Deep Neural Network

EHR Electronic Health Records

eICU eICU Collaborative Research Database

FN False Negatives

FP False Positives

GAN Graph Attention Network

GCN Graph Convolutional Network

GCT Graph Convolutional Transformer

GNN Graph Neural Network

GPU Graphics Processing Unit

ICD International Code of Diseases

ICU Intensive Care Unit

KL divergence Kullback Leibler divergence

LTS Long Term Support

MiME Multilevel Medical Embedding of Electronic Health Records

MLP Multi-Layer Perceptron

MRCI Medication Regimen Complexity Index

PHD Premier Healthcare Database

RAM Random Access Memory

ReLU Rectified Linear Unit

RNN Recurrent Neural Network

ROC Receiver Operating Characteristic

SWATS Hybrid Optimizer SWitches From Adam To SGD

TFRecord TensorFlow Record

TN True Negatives

TNR True Negatives Rate

TP True Positives

TPR True Positives Rate

Part I

Introduction

Deep learning gained remarkable success in recent years. This trend results from two major reasons: On the one hand, advances in Graphics Processing Unit (GPU) computing enable efficient training of neural networks. On the other hand, the availability of big data accelerates the advance of big deep learning.

At the same time, digitization proceeds in the domain of healthcare. An increasing number of hospitals introduce Electronic Health Records (EHR) systems. EHR systems routinely store heterogeneous hospital data, such as lab results, diagnoses or prescribed drugs. Consequently, EHR systems provide large amounts of data for research which paves the way for deep learning in healthcare [AMHK+17, HYV].
Utilizing the data source of EHR offers tremendous opportunities for public health. Data science does not only facilitate improvements in efficiency and quality of healthcare but also opens the door for new medical insights. Numerous promising studies were presented that show the potential of quantitative and predictive analysis to prevent diseases, adverse effects and eventually death. For example, future diagnoses [LKEW15, CBK+16, CBS+16c, MCZ+17], patients at risk [FML15, PTPV17] are predicted or diseases detected [CSSS16b, CBS+15, ESB+16].
Whereas the mentioned methods treat EHR data as a flat-structured bag of features, an approach called Multilevel Medical Embedding of Electronic Health Records (MiME) [CXSS18] takes advantage of the graphical structure of EHR to conduct heart failure prediction. MiME is based on a data set which explicitly contains causal information on diagnoses and prescribed drugs of hospital visits. Given this information, data is modeled as a hierarchical graph. The graph structure is used to learn meaningful representations of medical codes and enhances heart failure prediction. MiME demonstrates that models reflecting the graph structure have the potential to outperform bag of features approaches.
However, MiME is only applicable to data sets that explicitly show the underlying graph structure. Unfortunately, in the most common EHR data sets the graph structure is not obvious. To still utilize the unknown graph structure, Choi et al. [CXL+19] propose Graph Convolutional Transformer (GCT). GCT aims to learn the graph structure in case it is hidden. GCT is based on the concepts of Graph Neural Networks (GNNs) [ZCZ+18, XHLJ18], Transformer [VSP+17] and Graph Attention Networks (GANs) [VCC+17].

The goal of this work is to perform mortality prediction on a large data set of Premier Healthcare Database (PHD) [HZRS15] that contains more than one million of hospital admissions with respect to the underlying graph structure in EHR. Consequently, GCT is investigated as a method to learn the hidden graph structure. In the following step, the obtained graph structure serves as foundation for the acquirement of meaningful representations of EHR. These representations are potentially expected to improve the quality of the mortality prediction. The prediction is executed at hospital admission based on medical codes known on the first day of the hospital stay.
To assess the performance of GCT, it is compared to the baselines Transformer [VSP+17] and logistic regression [FGLB18]. The results of this work show that GCT after primary experiments already results in a good prediction performance. GCT performs slightly lower than the large-scale regression that was used as baseline. Regarding that the performance of GCT is only resulting from initial experiments, the approach seems to have great potential

and can be extended in several directions.

Outline: This thesis is structured as follows: In the first part, a general overview of the state of the art is given. This includes an introduction to big data analytics in Section 1, to GPU computing in Section 2 and the applied deep learning libraries, especially TensorFlow and Keras in Section 3. Section 4 gives background information on neural networks and introduces particular neural network architectures, such as GNNs that are relevant for this work. Section 5 displays the particular motivation and challenges of machine learning in the domain of healthcare and illustrates several approaches related to this work.
The second part refers to the experiments conducted in this work. Section 8 explains the applied method of GCT and the baselines in details. The results of GCT are then presented and compared to the baseline models in Section 9. Section 10 mentions remaining research questions and further extensions of GCT. In the end, a conclusion is drawn in Section IV.

Part II

State of the Art

This part of the thesis gives an overview of the state of the art. At first, the main challenges and characteristics or big data are mentioned in Section 1. Then, GPU computing and machine learning frameworks that are used in this work are presented in Sections 2 and 3. Section 4 provides basic theoretical foundations on neural networks. Section 5 introduces relevant approaches of deep learning in healthcare.

1 Big Data Analytics

Since an increasing number of processes are digitized, the amount of broadly available data grows. Regarding this phenomenon, the term big data emerged. Big data can be defined as "an expression that means large volume of data [that] can either be structured or unstructured and whose processing is difficult using traditional databases" [KSS+20].

Besides, big data can be characterized by the classic 5Vs: **Velocity** refers to the high speed of data creation and data updates. **Variety** indicates the heterogeneity and complexity coming along with big data. The variety of data requires additional effort in preprocessing and data integration. **Volume** describes the large amount of data. **Veracity** specifies the difficulty of big data to validate its quality. With many sources of big data, quality is difficult to control. **Value** refers to the output of aggregating many data resources which can lead to new valuable insights.

The characteristics of big data impose several challenges. The integration of heterogeneous high-volume data can be difficult. Appropriate hardware is necessary to store and manage the data and reveal new insights from it. Algorithms applied to big data should have the capability to upscale.

To analyze big data, Kulkarni et al. [KSS+20] mention several steps that are usually conducted. Firstly, data has to be collected. In the following, data is extracted and features are generated from raw data. Then, relevant features are selected and kept for the analysis. Based on the features, a predictive model can be estimated. In the end, the model or its result should be visualized [KSS+20].

2 Graphics Processing Unit Computing

The invention of GPUs in 1999 by the company NVIDIA[1] revolutionized the gaming market, redefined modern computer graphics and advanced parallel computing. GPUs allow to solve large problems by processing multiple threads simultaneously. While the Central Processing Unit (CPU) of a computer is especially adapted to sequential processing, the GPU is optimized for executing multiple tasks in a parallel manner [NBGS08]. Thereby, GPU ignited the era of Artificial Intelligence (AI) and made a massive contribution to the success of deep learning.
Originally, GPUs were developed for applications in computer graphics. The rendering of high definition graphic scenes requires inherent parallelism to draw numerous pixels in parallel. The range of application of GPUs was then extended to non-graphical applications. Since the use of GPU dramatically speeds up computing operations, especially linear algebra, it thereby enhances deep learning which requires a massive amount of computation steps when training neural networks [NBGS08].

2.1 Compute Unified Device Architecture

In 2006, NVIDIA developed the Compute Unified Device Architecture (CUDA) [NBGS08]. On the one hand, CUDA is a GPU architecture with hundred of cores. On the other hand, CUDA describes a parallel programming model that makes the application of multi-core architecture easier and more efficient. CUDA is beneficial in processing thousands of threads efficiently, since threads are modified to communicate and collaborate with each other. Furthermore, CUDA enables the joint engagement of CPU and GPU to process large amounts of data. CUDA is compatible with common programming languages.
The difference of computing an output on a CPU to computing on a GPU is clarified in the following example considering the addition of two vectors: $\vec{\mathbf{a}} + \vec{\mathbf{b}} = \vec{\mathbf{c}}$.

$$\begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \dots \\ c_n \end{bmatrix} \tag{1}$$

Calculating the operation on a CPU is equivalent to the computation of the output with a `for` loop. The CPU processes the vectors element-wise and subsequently.

```
for i in range[1,n]:
    c[i] = a[i] + b[i]
```

By contrast, CUDA solves this task by defining a function, `VecAdd` in this example. The task is split into subtasks and given to several threads which work in parallel. A thread is identified by a thread ID.

```
VecAdd(a, b, c)
    {
    int i = threadIdx
    c[i] = a[i] + b[i]
    }
```

The call of the `VecAdd` function can be illustrated as follows:

[1] NVIDIA: https://www.nvidia.com

```
VecAdd << n >> (a,b,c)
```

In this way, CUDA enables parallel execution on the GPU using multiple threads. By parallelizing, a task executed on a GPU can be dramatically speed up compared to the execution on the CPU. CUDA supports different hierarchies of parallelism that can be adapted by the programmer depending on the demands of the respective task. Creation, parallel execution, management, scheduling, synchronisation and termination of threads is completed automatically [NBGS08].

2.2 CUDA Deep Neural Network Library

The CUDA Deep Neural Network (CuDNN) library [CWV+14] provides efficient routine operations for deep learning tasks. CuDNN implements highly optimized low-level operations in parallel, such as backpropagation, pooling or normalization. The library increases reliability and efficiency, since researchers no longer have to write parallel code manually. CuDNN is compatible with many common machine learning libraries, such as TensorFlow [AAB+15], for example. CUDA enables developers to focus on high-level tasks instead of solving issues related to parallelization [CWV+14].

3 Deep Learning Frameworks

This section gives a short introduction to the deep learning frameworks TensorFlow [AAB+15] and Keras [C+15] which are used for the implementation in this work. Besides, several other frameworks, such as Theano [The16] or Pytorch [PGM+19] exist.

3.1 TensorFlow

TensorFlow is one of the most famous machine learning libraries to build and train neural networks. As already indicated in its name, tensors are the central data unit. Tensors can be described as multidimensional abstractions of matrices. In TensorFlow, a neural network is modeled as a graph. Within the graph, nodes represent operations and edges represent the data flow. Figure 2 illustrates the summation of two input tensors a and b:

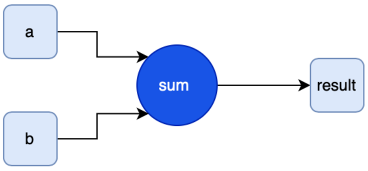


Figure 1: The operation $c = a + b$ represented as a TensorFlow graph.

In TensorFlow, mathematical operations are at first defined as stateful data flow graphs and initialized when a TensorFlow session is started. TensorFlow has many advantages. Code based on TensorFlow can run on a CPU or on GPUs in a computationally efficient way. TensorFlow also provides the flexibility to build customized neural networks and include advanced functionalities. Furthermore, a specialized debugger is available to examine the internal structure of TensorFlow graphs. A tool called Tensorboard comes along with TensorFlow and enables the visualization of complex graphs as well as monitoring the training process. TensorFlow can be complemented by high-level Application Programming Interfaces (APIs), such as Keras [C+15] which is built upon TensorFlow [AAB+15, Hol19, Gam18, XMS+17].

3.1.1 TensorFlow Records

TensorFlow proposes its specific binary storage format TensorFlow Records (TFRecords) which gives benefits in terms of training time and data storage. The class TensorFlow Sequence Example is optimized for storing sequence data. It allows to easily combine sequential and non-sequential information of a sample record. Sequential data can be stored in feature lists, while non-sequential data is saved as context [AAB+15, Gam18].

3.1.2 TensorFlow Estimator

TensorFlow Estimator [XMS+17] is a high-level API built upon TensorFlow that simplifies the development of deep learning models. It provides preimplemented methods that enhance the creation of computational graphs, variable initialization, data input or creation of logging files, for instance. The main class of of TensorFlow Estimator is the Estimator class as demonstrated in Figure 2 that contains the following core functions:

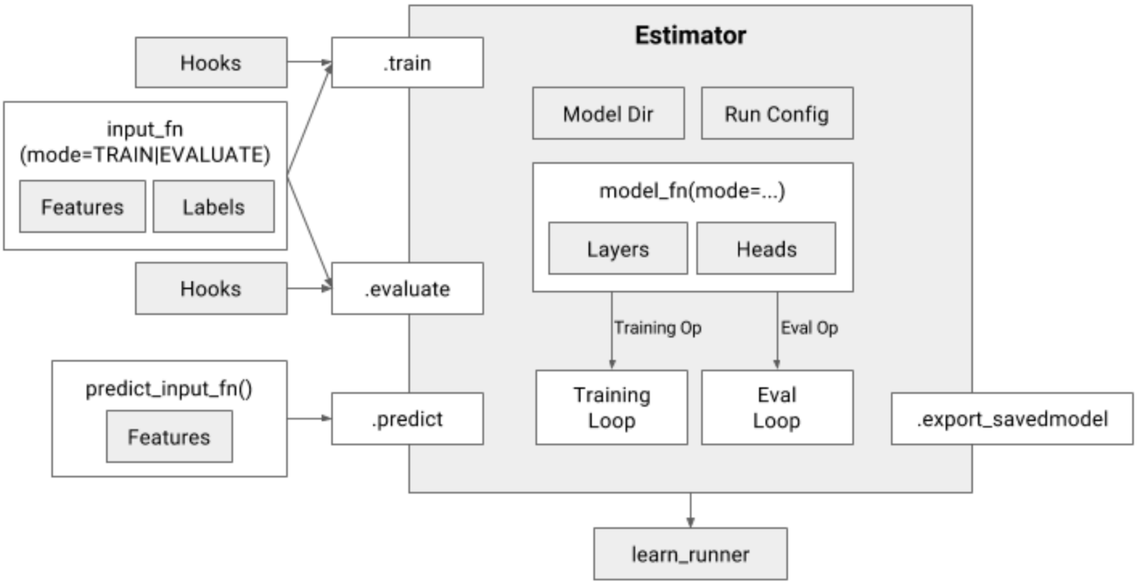


Figure 2: High-level overview of the class Estimator. [XMS+17]

- `Input_fn()` contains the input in form of features and labels as tensors.
- `Model_fn()` specifies the model itself including the details and the configurations for training, evaluation and testing.
- `Train()` proceeds the training of the model on the train set.
- `Evaluate()` calculates evaluation metrics.
- `Predict()` enables making predictions with the obtained model.

Hooks allow the implementation of individualized, advanced optimization techniques. Estimator itself is able to initialize the underlying TensorFlow session and is in control of the training process.

3.2 Keras

Keras is a high-level API built upon TensorFlow in a way that both libraries can be used jointly. An advantage of Keras is the stronger modularity compared to TensorFlow. Several neural network components can be stacked with high-level primitives while maintaining readability. The Model and Sequential API of Keras serve as a sufficient base to build up neural networks from scratch. To get the best out of both libraries, Keras and TensorFlow are recommended to be combined. Keras is more appropriate for implementing standard structures, while TensorFlow provides further options for adding custom elements to the network [C+15, Moo19].

4 Introduction to Neural Networks

This section gives an introduction to the methodology of neural networks. Neural networks are inspired by the processing of information in the human brain. The basic unit of a neural network is a neuron. It receives an input from another neuron or from external sources and computes an output.

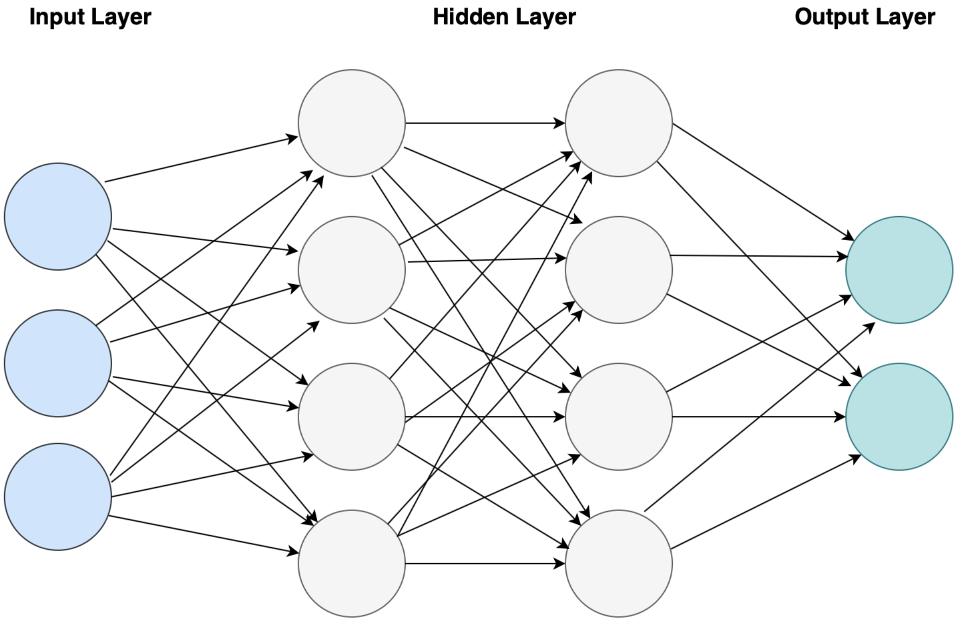


Figure 3: Overview of a simple neural network with an input, two hidden layers and one output layer.

Typically, each input is multiplied by a weight. All inputs multiplied by the weights and a bias are added and passed to a so-called activation function. Activation functions are used to introduce non-linearity in the learning process, since most real-world problems are non-linear. The most common activation functions are sigmoid, tanh or Rectified Linear Unit (ReLU):

$$\text{sigmoid: } \sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{2}$$

$$\text{tanh: } f(x) = \tanh(x) \tag{3}$$

$$\text{ReLU: } f(x) = \max(0, x) \tag{4}$$

More details on common activation functions can be found in [GBC]. In order to build a neural network, several neurons are linked with edges. The classic multi-layer feedforward neural network, or so-called Multi-Layer Perceptron (MLP), consists of one input layer, at least one hidden layer and one output layer as illustrated in Figure 3. The term feedforward describes that information is propagated from the input, through the hidden layers and finally to the output layer in a non-cyclic way. The output values of a layer are called activations. As mentioned in the previous section, neural networks are considered as deep learning algorithms. The term deep refers to the large number of hidden units that are used. These units extract representations or also called features from input data. The output layer converts the activations of the last hidden layer to a desired representation, such as probability values in classification tasks [GBC].

A wide range of variants of neural networks exist. To mention the most common categories, Convolutional Neural Networks (CNNs) arise from image processing and have the key ability of extracting representations on multiple abstraction levels with convolutional layers. Recurrent Neural Networks (RNNs) are optimized to process ordered sequences by storing past inputs in an internal state [GBC].

4.1 Deep Learning vs. Machine Learning

Machine learning can be defined as a subcategory of AI. Machine learning algorithms are based on data and do not call for explicitly and previously designed procedure steps. Typical machine learning tasks are classification, clustering or prediction [Reb19].

Deep learning is a subclass of machine learning. The most well-known deep learning models are neural networks. While machine learning algorithms rely on manual feature extraction, deep learning models extract valuable features themselves [Tau19]. The capacity of extracting meaningful features arises from several hidden layers in a neural network. These hidden layers learn data representations which are helpful to solve a specific task. The models are called deep, since several hidden layers allow a model to learn increasingly expressive representations of the input data from layer to layer. This process is also called representation learning, embedding learning or feature extraction [Tau19, BK18, Wan20]. Section 4.4 and 5.3 go deeper into the details of representation learning in general and applied to EHR. Machine learning and deep learning both only deliver valuable results if an adequate amount of data input as well as sufficient computation power are available. Recently, the availability of both resources massively increased and thus, machine learning and especially deep learning recorded a sharp increase in popularity [Tau19].

4.2 Advantages and Disadvantages of Neural Networks

Neural networks have several advantages and disadvantages. On the one hand, neural networks are able to model non-linear and complex relationships and approach arbitrary non-linear functions. With appropriate hyperparameters, they can generalize well to unseen test data and are fault-tolerant. A major strength of deep learning is the automatic feature extraction without human intervention.

On the other hand, the training of neural networks is computationally expensive and requires a large amount of labeled input data. The amount of required data grows with the number of trainable parameters. A general rule to determine how much data is enough data to solve a problem does not exist [Wan20]. Neural networks learn progressively and rely on the updates of the previous step. Thanks to GPUs, some calculation steps can be parallelized and accelerated, but in general, the training of neural networks remains a time-consuming task. Since neural networks learn by classifying samples and comparing their output to the ground truth, the whole learned information is stored in the trainable parameters. These are often difficult to explain and interpret [Wan20].

4.3 Training of Neural Networks

Before the actual training process starts, the underlying data set is randomly split into a train, a validation and a test set which are independent of each other. Neural networks are trained on the train set. While training proceeds, the performance is evaluated after each weight update on the train and the validation set. Afterwards, the final model is used to make predictions on the test set and assess the prediction performance of the neural network on unseen data. In general, neural networks belong to the class of supervised learning algorithms. This implies that ground truth labels are required for training neural networks. The algorithm of training neural networks is called backpropagation. At first, all learnable parameters are initialized. Multiple initialization methods exist such as initialization with uniformly distributed or random values. Then, an element is propagated through the network and the output is calculated with the initial weights. The output of the final layer is compared to the ground truth. A loss function is used to calculate the prediction error of the network. The loss function is deviated with respect to the weights in order to minimize the classification error. The weights are updated in the inverse direction of the gradient. Consequently, the network is expected to be capable of categorizing the element correctly in the next step.

Since the full train set often exceeds the capacity of the memory, dividing the data set into chunks of a given size is often necessary. These chunks of data are called mini-batches. To train a network on mini-batches, the proposed weight changes for each sample within a mini-batch are aggregated and applied when the whole mini-batch was passed through the network. An epoch describes a full forward and backward pass of the complete data set through the network. The process of propagating a mini-batch through the network and updating the weights is called an iteration or a step. If a data set has, for instance, 2,000 training samples in total and it is divided into mini-batches of size 500, it would take four iterations to complete one epoch [Wan20].

4.3.1 Gradient Descent

The most common method to train a neural network is gradient descent. Gradient descent optimizes the loss function by calculating gradients to update the values of the trainable parameters w in the network.

$$w_{t+1} = w_t - \eta \cdot \nabla_w \mathbf{E}(w_t) \tag{5}$$

The gradient of the loss function with respect to the weights w in step t is described as $\nabla \mathbf{E}(w_t)$. The learning rate η represents the step size towards the optimum and determines how much the network can learn in each step [Wan20]. More details on the learning rate can be found in Section 4.7.

Several variants of gradient descent exist. Classic gradient descent or so-called batch gradient descent computes the error for the whole train set and then proposes weight updates. Consequently, only one weight update is performed per epoch. Batch gradient descent is slow and large data sets might not fit in memory. In contrast, stochastic gradient descent updates the parameters of the network after each sample. This leads to faster training but more oscillation. To combine the advantages of both methods, mini-batch gradient descent is often used. As already described, the data set is split in chunks of a mini-batch size and the updates are calculated for each mini-batch [Wan20]. The mini-batch size is a hyperparameter. Typical mini-batch sizes range between 32 and 256. A mini-batch size of one corresponds to stochastic gradient descent, whereas a mini-batch size of the size of the train set corresponds to batch gradient descend. In mini-batch gradient descent, the batches are subsequently given to the network and the parameters are updated after each mini-batch. The train set is shuffled when the network has seen the full data set [Rud16].

Gradient descent still has several drawbacks. At first, the choice of the learning rate can be difficult. An extremely small learning rate slows down training while a large learning rate misses the optimum and leads to oscillation or divergence [Rud16]. Furthermore, gradient descent uses the same learning rate for all parameters although the parameters might require different step sizes. In classic gradient descent, the learning rate remains constant during the training process even though it would make sense to slow down or accelerate during training [Rud16].

4.3.2 Momentum

As previously mentioned, training with gradient descent often experiences problems in navigating through ravine areas and fails to accelerate in the right dimension. To address this issue, momentum [mom99] is proposed. Momentum adds a fraction of the update of the previous step, controlled by parameter γ, to the current update:

$$v_t = \gamma \cdot v_{t-1} + \eta \cdot \nabla_w \mathbf{E}(w_t) \tag{6}$$

$$w_{t+1} = w_t - v_t \tag{7}$$

The momentum term increases when gradients point in the same direction and reduces updates when gradients point in different directions. Momentum leads to faster convergence with less oscillation [Rud16, mom99]. The momentum term γ is a hyperparameter. In [mom99] a value close to 0.9 is recommended.

4.3.3 Adagrad and Adadelta

Adagrad [DHS11] is an optimizer that sets different learning rates for each parameter. Frequent parameters might require smaller updates than rare parameters. The update rule of Adagrad is described as follows:

$$w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{\mathbf{G}_{t,ii} + \epsilon}} \cdot \nabla_{w_t} \mathbf{E}(w_{t,i}) \tag{8}$$

Updates are done per parameter w_i and for each time step t. A matrix where each diagonal element (i, i) is the sum of the squares of the gradients with respect to weight w_i at time step t is represented by $\mathbf{G}_t \in R^{d \times d}$. The parameter ϵ is introduced to avoid division by zero. With Adagrad, the learning rate no longer requires manual tuning. Usually, the learning rate is set to an initial value of 0.01 and updated automatically during training. Since the learning rate is scaled down by gradients of past steps that are all added as positive values, the learning rate is at risk to shrink down to zero and prevent the network from gaining additional knowledge. Adadelta [Zei12] is an extension of Adagrad [DHS11] that diminishes this drawback by restricting the number of considered past gradients to a fixed window size [Rud16].

4.3.4 Adaptive Moment Estimation

The optimizer Adaptive Moment Estimation (Adam) [KB14] computes adaptive learning rates for each parameter w_i per time step t. Adam takes past gradients and past squared gradients into account by exponentially decaying the average of them. The exponential decay makes shorter lagging gradients more important for the calculation of the current gradient. To increase readability, the gradient in step t is denoted as

$$g_t = \nabla_{w_t} \mathbf{E}(w_t) \tag{9}$$

The estimators of the first and second moment are declared as m_t and v_t.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1)\, g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2 \end{aligned} \tag{10}$$

The weight decay is controlled by β_1 and β_2. Kingma et al. [KB14] recommend the default values of $\beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$. Since the estimates m_t and v_t are biased towards zero, Adam introduces bias corrected versions $\hat{m}_t$ and $\hat{v}_t$, which are defined as follows:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{11}$$

In summary, the Adam update rule can be described as denoted below:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{12}$$

Adam outperforms other optimization algorithms in speed of convergence and lower variance of updates. Especially for large data sets Adam is applicable because of its computational efficiency and its lower memory footprint compared to other methods [Wan20, Rud16]. Ruder [Rud16] recommends Adam as the most common optimizer for training neural networks. More details on Adam can be found in [KB14] and [Wan20].

4.4 Representation Learning

As previously described, the main strength of neural networks is the capability to extract representations or also called features of input data. This qualification enables neural networks to so-named end-to-end learning, since the representations are automatically learned during training without human guidance. A representation can be obtained by the projection of an input vector in a lower-dimensional space.
When neural networks are applied, it is usually required to feed discrete, categorical variables, such as words, codes or objects as input into the network. The intuitive way to find a numeric representation of categorical variables is to model them as one-hot-encoded vectors $\vec{x} \in \{0, 1\}$. In context of words, for example, a vocabulary might be $\{queen, king, man\}$. The words are respectively represented as follows:

$$\vec{x_1} \ = \ [1, 0, 0], \ \vec{x_2} \ = \ [0, 1, 0], \ \vec{x_3} \ = \ [0, 0, 1] \tag{13}$$

The one-hot-encoding has two main drawbacks. At first, the representation is sparse and leads to high dimensionality when the vocabulary is large. This can cause extensive computation times. Second, the representation does not reflect relationships between entities. Considering the mentioned vocabulary, it would be desirable to model the vector of *queen* closer to the vector of *woman* than to the vector of *man*. Such a representation can be obtained by projecting vectors into a low-dimensional space. The dot product $\odot$ of two vectors allows to draw conclusions about their similarity. Calculating the dot product of one-hot-encoded vectors returns zeroes for all combinations, since the one-hot-encoding does not contain relational information. Geometrically, the vectors are orthogonal to each other:

$$\vec{x_1} \odot \vec{x_2} = 0, \quad \vec{x_1} \odot \vec{x_3} = 0, \quad \vec{x_2} \odot \vec{x_3} = 0 \tag{14}$$

By projecting the vectors into a lower dimensional space, here two-dimensional, real-valued vectors are obtained.

$$\vec{x_1} = [0.53, 0.83], \quad \vec{x_2} = [0.6, 0.8], \quad \vec{x_3} = [-0.78, -0.62] \tag{15}$$

The output of the dot product of the updated vectors is an indicator of similarity. In this case, high positive values indicate a strong relation between the words, as the following example shows:

$$\vec{x_1} \odot \vec{x_2} = 0.99, \quad \vec{x_1} \odot \vec{x_3} = -0.94, \quad \vec{x_2} \odot \vec{x_3} = -0.97 \tag{16}$$

The projection is acquired by multiplying the values by a weight matrix whose values can be learned by a neural network [Koe18].

4.5 Overfitting and Underfitting

A central challenge in machine learning is to find a balance between over and underfitting. A model should not only performs on seen training data but also on unseen test data. A complex model tends to learn the training data perfectly but may introduce unnecessary variance when predicting on the test set. This phenomenon is called overfitting. A model will underfit, if it is too simple to capture meaningful relations in the data and consequently will not able to make reliable predictions. The dilemma is illustrated in Figure 4.

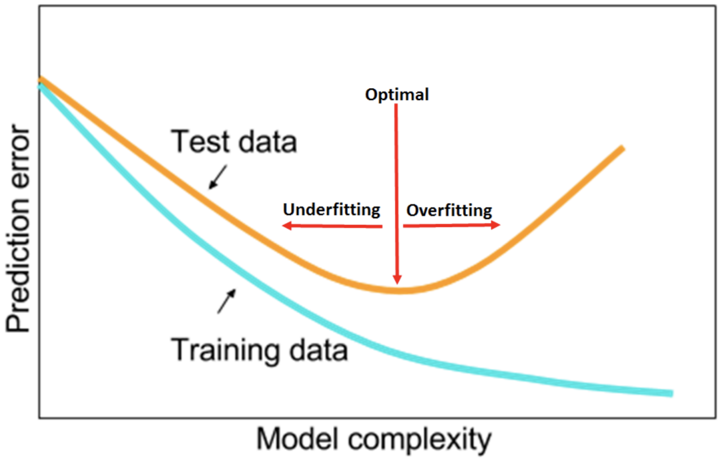


Figure 4: Trade-Off between Over- and Underfitting [Smi18]. While the prediction error on the training data (blue curve) monotonously decreases in theory, the validation error (orange curve) reaches a minimum and rises. The minimum of the validation loss corresponds to the model complexity which is most appropriate to handle the trade-off between over- and underfitting.

Considering neural networks, the number of hidden layers should correspond to the complexity of the model. In general, a model with too many layers leads to overfitting, whereas a model with not enough layers underfits. In order to solve the trade-off between over- and underfitting, a model that minimizes the loss on unseen data and not only on the train set should be chosen [Smi18]. The tuning of hyperparameters such as regularization coefficient or dropout rate as well as restricting the number of hidden layers helps to balance over- and underfitting [GBC, Wan20]. The following Section 4.6 introduces some common regularization methods.

4.6 Regularization Methods

In general, regularization describes the process of introducing additional information to a model in order to prevent overfitting. Several methods exist to regularize neural networks and enable the learned model to generalize to unseen data. While adding more complexity to a network reduces the risk of underfitting, several techniques are suggested to prevent overfitting. Regularization methods allow to prevent a neural network from learning unnecessarily complex models that work successfully on training data but fail to generalize to unseen test data [GBC]. In the following, some common regularization methods are presented [GBC].

4.6.1 Dropout

Dropout [SHK+14] can be understood as training several thinned neural networks and averaging over an ensemble of these thinned networks. For each training sample the units of a network are randomly dropped which serves as an effective regularization method. The probability of a unit to be dropped is defined as p. Consequently, for each training sample a new network is sampled and trained. The training can be processed as already described in Section 4.3. Parameters of deactivated units contribute a gradient of zero. At test time, all units are present but the learned weights are rescaled by multiplication by a factor $\frac{1}{1-p}$ [AAB+15]. This can be understood as approximately averaging over the thinned networks. As consequence, each neuron has to learn meaningful representations itself without relying on the presence of other units. Neurons become more robust which avoids overfitting and leads to a lower generalization error [SHK+14, AAB+15]. The dropout rate is a hyperparameter that can be tuned, see Section 4.7.

4.6.2 Weight Decay

Large weights in neural networks often indicate overfitting and lead to instable training. Weight decay is a widespread procedure to prevent a neural network from learning large weights by adding a regularization term to the loss function. The additional term penalizes large weights. Consequently, having small weights is integrated in the loss function and becomes part of the optimization. The influence of the additional term is controlled by a hyperparameter λ, see Section 4.7. Common regularization terms are L1 regularizer which considers the amount of the weights and L2 regularizer which considers the squared weights [Ben12].

4.6.3 Early Stopping

Early stopping refers to the choice of the number of training iterations. Bengio [Ben12] argues that early stopping is an efficient and simple technique to prevent neural networks from overfitting. By examining the loss functions on the train and the validation set during training, overfitting can be discovered, see Figure 4.5. Typically, the loss on the train set continues to decrease, while the loss on the validation set reaches a minimum and starts to increase. A low validation loss indicates a low generalization error. The idea behind early stopping is to interrupt the training progress when the validation error reaches its minimum. Prechelt [Pre96] illustrates several stopping criteria referring to the loss function on the validation set [Ben12, Pre96].

4.6.4 Batch Normalization and Layer Normalization

Batch normalization and layer normalization provide several positive effects on neural networks, including regularization. As already mentioned, deep neural networks are composed of many layers. The input of a layer corresponds to the output activations of the previous layer. Since the activations of the previous layer are updated during training, their distribution is not constant. Consequently, a layer continuously has to adapt to the distribution of the previous layer. This phenomenon is called internal covariate shift [IS15] and leads to slow convergence. Batch normalization is a technique responding to this issue. The idea behind batch normalization is to rescale the activations for each mini-batch to a standard deviation of one and a mean of zero. The standard deviation and the mean of all activations are computed for a specific neuron over the whole mini-batch. Batch normalization enables

faster and more stable training and thus, serves as a regularizer. With batch normalization applied, less or no dropout is required. Furthermore, batch normalization leads to invariance to differently scaled input values. On the downside, batch normalization is dependent on the mini-batch size and can only be applied after one complete mini-batch is processed. Consequently, batch normalization is not applicable to online-learning, stochastic gradient descent (with mini-batch size one) and RNNs.
Another approach [BKH16] proposes layer normalization. Variance and mean for rescaling are computed for a single input sample based on all inputs to the neurons of an entire layer. This makes layer normalization independent of the mini-batch size, since normalization can be applied after processing one single sample. Layer normalization leads more stable and faster training [BKH16]. More information on batch and layer normalization can be found in [BKH16], [IS15] and [Wan20].

4.7 Hyperparameter Tuning

As already mentioned, neural networks do not only consist of trainable parameters that are tuned as part of the training process but also of hyperparameters. These are set prior to the the training and are manually or automatically selected. The process of determining these parameters is called hyperparameter tuning. In deep learning, hyperparameter tuning is often considered as an "art" [Smi18], since no universal procedure exists that guarantees to lead to a powerful network. Moreover, the number of parameter combinations grows exponentially with the number of parameters and makes hyperparameter tuning a time-consuming and computationally expensive task. Possible approaches are Bayesian optimization [PGCP+99] or random search [BB12]. Random search can be parallelized on clusters. In general, several parameters can be tuned.

- **Mini-batch size:** As already mentioned in the previous Section 4.3, the mini-batch size determines how many samples are propagated through the network before a weight update is performed. Usually, a mini-batch size between one and a few hundred is chosen. A larger mini-batch size speeds up training, since more multiply-add operations can be parallelized within one iteration when calculating the gradients per sample. In total, less aggregated updates can be performed during the same computation time. The mini-batch size interacts with the learning rate. Larger mini-batches enable a larger learning rate, since the mini-batch contains more samples to learn from per iteration. A smaller mini-batch size can lead to faster but eventually more instable learning. According to Bengio [Ben12], the mini-batch size has more impact on the training time than on the test performance [Smi18, Ben12].

- **Learning rate:** The learning rate or step size is multiplied by the gradient and determines the impact of a sample on the network update. In Equation 5 in Section 4.3, the learning rate is described by the parameter μ. Bengio [Ben12] recommends a learning rate in the interval of $[10^{-6}, 1]$ and calls the learning rate the most important hyperparameter to tune. If the learning rate is too high, the network will learn fast, but on the downside it will diverge and miss the global minimum. If the learning rate is too low, the network will learn slowly and be stuck in local minima. In order to tune the learning rate, Bengio [Ben12] advises to start with a large learning rate and decrease it during the training process.

- **Dropout Rate:** Dropout is a useful technique to prevent overfitting. The details on dropout are already mentioned in Section 4.6.1. If the dropout rate equals to zero, no dropout is applied, whereas higher values lead to more dropout. High dropout rates lead to underfitting and low dropout rates are not effective enough to prevent overfitting [SHK+14, AAB+15].

- **Regularization coefficient:** Weight decay is also proposed to prevent overfitting, see Section 4.6.2. It can be implemented by adding a regularization component to the loss function that penalizes large weights. The strength of penalization is controlled by the hyperparameter λ. The larger λ is, the more larger weight values will be penalized and overfitting will be prevented. If λ is too large, the model might not be able to capture relations in the data.

- **Number of epochs:** The number of epochs determines how often the train set is propagated through the network, see Section 4.6.3. A high number of epochs can lead to overfitting whereas a small number of epochs can lead to underfitting. In his recommendations, Bengio [Ben12] proposes to regulate the epoch size by early stopping, see Section 4.6.3.

- **Number of hidden units:** The number of hidden units in a neural network is a further tunable parameter that impacts the trade-off between over- and underfitting. A too simple model underfits and be unable to cover all relations in the data, while a model of unnecessary complexity overfits on the training data.

4.8 Attention in Neural Networks

The main idea of attention in neural networks is to enable a model to "focus on the most relevant parts of the input to make decisions" [VCC+17]. Attention was firstly introduced by Mnih et al. [MHGK14] in order to determine the most relevant parts of images. Meanwhile, Bahdanau et al. [BCB14] applied attention to conduct machine translation tasks. Recently, numerous approaches emerged that use attention to enhance the predictive capability of models.

4.8.1 Transformer: Attention in Sequence-to-Sequence Models

In the following, a short introduction to the structure of attention is given. The comprehension of attention in general is beneficial for understanding how attention in graphs works as it is applied in this work.
The origin of the attention mechanism lies in sequence-to-sequence models such as machine translation. Classic RNN models often fail to capture long-term dependencies because they tend to forget the beginning of a sentence. Furthermore, sequential models are difficult to parallelize which leads to extensive computation time. The idea behind the attention mechanism is to represent a sentence as a context vector. This vector contains global information about the importance of each word for the translation. In their approach Transformer [VSP+17], Vaswani et al. show that a model based only on attention without any recurrence or convolution can outperform previously proposed RNN models in machine translation tasks.
An exemplary task might be to translate the English sentence *I like trees* to the German equivalent *Ich mag Bäume*. Both sentences have to represent vectors to be interpreted by an algorithm. The most intuitive way is to describe the input sentences as an one-hot-encoded vector given a vocabulary of all possible words. Given the vocabulary {*trees, I, like, sky*}, for example, the English sentence can be represented as a composition of its words as vectors:

$$\text{I: } (0,1,0,0), \text{ like: } (0,0,1,0), \text{ trees: } (1,0,0,0) \tag{17}$$

As described in Section 4.4 the multiplication of the vectors by a learned weight matrix $\mathbf{W}$ returns representations. The size of $\mathbf{W}$ defines the output size of the embedded word vectors. In the next step, the obtained representations are fed into an encoder, which basically is a stack of several identically structured encoder-layers. Each of them consists of an attention component and a feedforward component. The attention component computes the importance of other words within a sentence itself in order to encode one word of the sentence respectively. Consequently, the words are assigned a computed weight that indicates the degree of relevance for the encoded word. Thus, the input vector of a word is split into three different vectors: Query vector, key vector and value vector for an input word x_i at position i. Queries q_i, keys k_i and input words x_i are defined so that following equations hold:

$$x_i\mathbf{W}^Q = q_i \ , \quad x_i\mathbf{W}^K = k_i \ , \quad x_i\mathbf{W}^v = v_i \tag{18}$$

In other words, keys, values and queries are projections of each word and helpful abstractions for the attention calculation. $\mathbf{W}^Q$, $\mathbf{W}^K$ and $\mathbf{W}^V$ are weight matrices which can be obtained in the training process. In matrix notation, $\mathbf{X}$ contains the representation of one word per row and $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ represent the stacked vectors of queries, keys and values. Next, a score for a respective word is calculated against all other words in the sentence. The score measures the meaningfulness of another word in the sentence for the target word. The score of a target word x_i with another word x_j within a sentence can be described by the following equation:

$$z_{i,j} = score_{i,j} = q_i * k_j \tag{19}$$

This calculation is done for all words in the sentence. To make sure that the scores range between zero and one, they are passed to a softmax function. The scores are then scaled down by division by the square root of the dimension $\sqrt{d_k}$ of k to prevent large values. The softmax function has slow gradients on the upper and lower range so that large values could lead to vanishing gradients. Afterwards, the adapted scores are multiplied by the value vectors. As a result, the values are weighted with their respective attention weight. In summary, the process can be described as follows:

$$\text{Attention } (\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{Z} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{20}$$

To stabilize learning, multiple attention heads can be applied. This idea is named multi-head attention. It means that not only one query, key and value matrix are used but multiple versions of these matrices. The number of applied attention heads is a definable hyperparameter. The different matrices can be learned in parallel and independently of each other. Since the trainable matrices are initialized with random values, the model has the opportunity to learn varying representations which leads to more stable results. The outputs of the attention heads are concatenated and multiplied by another learnable matrix $\mathbf{W}^O$.

Besides the encoder, the original approach Transformer consists of a decoder. After an input passes all the encoder layers, the output keys and values of the last encoder layer are fed into the decoder. Again, a decoder is built by several stacked decoder layers. All of them have the same structure which includes an attention layer, a masked encoder-decoder attention layer and a feedforward layer. While keys and values generated by the encoder serve as inputs for the decoder layers, the queries are obtained from the previous decoder layer. Masking is employed in order to prevent the decoder from concentrating on unknown positions after the currently translated word that are only available in the train set. The output of the decoder is fed into a linear layer and subsequently in a softmax layer resulting in probabilities. During the training process, these probabilities can be compared to the true word denoted as one-hot-encoded vector. The loss is backpropagated through the network to update the parameters. When predicting, the highest value in the vector indicates which one-hot-encoded vector should be proposed as the predicted output word [VSP+17, Ala18, Thi18, Dra19].

4.9 Graph Neural Networks

The application of CNNs [GBC] to tasks such as image classification, image segmentation or machine translation led to a breakthrough in deep learning. Nevertheless, data that is fed into a CNN has to follow a specific order and CNNs fail to capture relationships in non-euclidean domains, such as graph structures [VCC+17]. However, graphs apply to numerous applications, such as social networks, knowledge graphs or natural science [ZCZ+18, WPC+19].
In case of graph-structured data, a specific order is not given, since a graph has no unique starting point. The examination of all possible orders of nodes would lead to a large number of redundant computations. Furthermore, the edges of a graph contain valuable information about the relationship between nodes. GNNs introduce graph convolution that translates the concept of convolution in CNNs to graphs. GNNs can on the one hand be applied to end-to-end learning tasks on graph-structured data. On the other hand, GNNs are suitable for learning low-dimensional vector representations of the nodes in a graph by aggregating over the neighborhood of nodes. In the following, the low-dimensional representations of nodes in a graph can be used to solve machine learning tasks, such as outcome prediction or classification [WPC+19, VCC+17].

4.9.1 Introduction to Graphs

A graph G can be defined as $G = (U, E)$ with a set of nodes $U = \{u_1,\ u_2,\ ...,\ u_n\}$ and a set of edges E. The number of nodes is described as $n = |u|$. The adjacency matrix $\mathbf{A}$ notates the relations between the nodes and has the size $n \times n$. It contains binary values if the graph is unweighted or the respective edge weights a_{ij} between two nodes u_i and u_j, if the graph is weighted. For an undirected graph, $\mathbf{A}$ is symmetric. Furthermore, a graph $G = (U, E, X)$ can be attributed with a set of features $X = \left(\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_n\right)$ of size $n \times d$, where each node u_i obtains a d-dimensional attribute. The feature vector for node u_i is defined as $\vec{h_i}$. The neighborhood $\mathcal{N}_i$ of a target node u_i is a set of nodes that includes the u_i itself and all first-ordering neighbors [LRK+18, VCC+17].

4.9.2 Graph Convolution

Fundamentally, the convolution is the main operator to obtain representations from input data that support a predictive model to improve its performance. In image processing, a convolution is defined as the computation of the dot product between a filter matrix and the overlapping portion of an input image. In images, for example, typical filters are edge detectors. The application of different filters enables the network to extract features on multiple abstraction levels from input data. [GBC, Reb19, Tau19, WPC+19].
As already mentioned, convolutions as used in CNNs are only applicable to euclidean data, such as images. Since graphs do not correspond to a fixed grid-like structure, convolutions cannot be applied to graphs in the same way as to images [VCC+17]. Although graphs can be considered as a more general representation. An image, for instance, can be interpreted as a graph with fully connected pixels as nodes. The neighborhood of a target pixel would be the eight surrounding pixels.
Velickovi et al. [VCC+17] derive a generalized version of the convolution operator that is applicable to graphs. Regarding the notations in Section 4.9.1, the output of a graph convolutional layer updates the features of a graph to new features $X' = \left(\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_n\right)$. Similarly to convolutions on images, convolutions on graphs are an aggregation over the neighborhood

of a node. Velicovic et al. [VCC$^+$17] propose the following function to compute the output features of a convolutional layer:

$$\overrightarrow{h_i}' = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \vec{g_i}\right) \tag{21}$$

The activation function is specified by σ and α_{ij} represents a factor describing the importance of the features of node u_j to node v_i. The factor α_{ij} can be determined by edges of the graph or by learnable weights in the attention mechanism, see Section 4.8. The transformation function $\vec{g_i}$ with a learnable weight matrix $\mathbf{W}$ can be described as $\vec{g_i} = \mathbf{W}h_i$ [VCC$^+$17].

4.9.3 Graph Attention Networks

GANs apply the attention proposed in Transformer [VSP$^+$17] to GNNs. The main principle of GANs is adding attention to graph convolutions. Attention layers enable nodes to focus on the nodes in their neighborhood [VCC$^+$17] when finding representations. Several objectives can be taken into account when attention is applied to graphs. First, graph structures are often noisy and complex. Attention helps to focus on important parts of a graph and ignore noisy parts. Second, attention provides a relevance score for each node in a graph. Third, this relevance score provides an interpretable measure to understand the importance of an element for a respective task [LRK$^+$18].
Given a graph as defined in Section 4.9.1, u_i is a target node with the neighborhood $\mathcal{N}_i = \{u_0, u_1, ..., u_{|\mathcal{N}_i|}\}$. Attention on graphs can be defined as a function f that assigns every node in in the neighborhood of a target node an attention score that lies in the interval of [0,1]:

$$f : \{u_i\} \times \mathcal{N}_i \rightarrow [0, 1] \tag{22}$$

The representation of a target node can be learned by aggregating over the neighborhood of the target node with respect to obtained attention values for each neighbor [LRK$^+$18].

5 Machine Learning in Healthcare

This section presents the motivation behind deep learning in healthcare as well as the particular challenges in this domain. Furthermore, some approaches of deep learning in healthcare that refer to the model in this work are introduced.

5.1 Motivation and Opportunities

Big data offers many opportunities to improve healthcare. Firstly, since digitization proceeds, an increasing amount of data is accessible for machine learning. Secondly, healthcare is a major concern for humanity. Improvements in this area have the potential to ameliorate the quality of living which is a goal of high priority. Machine learning algorithms do not only offer opportunities to gain new medical insights but can also lead to higher quality in healthcare. Based on machine learning, standard clinical tasks could be automated and allows to detect and to focus on severe or rare diseases. Furthermore, machine learning in healthcare could lead to high-precision and individualized medicine. Since machine learning allows to make predictions, preventive care becomes conceivable. This would imply the detection of diseases before measurable symptoms can be manifested. Thanks to data-driven approaches, health trend analysis and drug efficiency studies can be conducted [KSS+20].

5.2 Challenges

Besides the previously described challenges resulting from big data, see Section 1, the use of machine learning applied to big data in healthcare faces particular challenges.

First, EHR data is often segmented. Usually, a patient visits several different hospital in his or her life. In consequence, complete patient histories are difficult to track [STBR18]. As long as data is not perfectly integrated amongst hospitals, the lack of data must be carefully taken into account [GNS+18, BK18].

Second, EHR data contains confidential information. In order to meet privacy requirements, data can only be published in an anonymized version. Strong confidentiality requirements for EHR inhibit authors to produce comparable results, since only a few publicly available data sets exist [GNS+18, STBR18].

Third, EHR data is usually heterogeneous, since it is composed of many different data sources, such as diagnosis codes, drug prescriptions, notes in form of free text, laboratory results or images. Diverse data sources lead to various data types such as numeric, datetime, categorical or string [STBR18].

Furthermore, some variables have an underlying time order. Typically, a patient record consists of a time series of several visits, while drug prescription codes of a visit are not necessarily ordered [CBS+16a]. Thus, data preprocessing and data organization play an important role when working with big data in healthcare [RWD+17].

Moreover, decision making in the domain of healthcare entails an additional effort in producing interpretable and explainable models and results [RWD+17, STBR18, KSS+20].

5.3 Representation Learning in Healthcare

As already described in Sections 4.1 and 4.4, the main strength of deep learning methods lies in the extraction of meaningful representations. The projection of one-hot encoded vectors in a lower-dimensional space to obtain real-valued vectors is not only applicable to words but also to EHR data, such as diagnoses codes, drug codes or lab codes. Representation learning

for medical codes poses several challenges. Firstly, interpretability is a main issue in healthcare. Representations should therefore be understandable and intuitive. Secondly, EHR data sets often capture millions of visits. To process this large amount of data, the representation algorithm must be scalable [CBS+16a].
Same as for words, it is desirable to capture code relations within the representation. Regarding medical codes, for instance, the vector representations of the diagnoses pneumonia and bronchitis should be closer to each other than the representations of the diagnoses pneumonia and obesity, since the firstly mentioned diagnoses are both lung diseases. As indicated in Section 4.4, the one-hot-encoding fails to capture such dependencies. An appropriate representation replaces manual feature engineering, is scalable, general, interpretable and does not require medical expert knowledge [CBS+16a, CSSS16a].

The following approaches implement representation learning on EHR. In their approach [CSSS16a], Choi et al. investigate skip-gram [MCCD13] to learn representations of medical codes. Initially, skip-gram was applied to words [MCCD13], but the concept can be transferred to medical codes. Skip-gram is based on the co-occurrence of codes of visits and does not require additional labels. The average log-probability of two codes appearing together in a record is maximized to learn a representation. In the following, the code representations are aggregated to a patient representation. Based on this patient representation, a heart failure risk score is predicted. The method shows superior performance compared to standard machine learning models, such as logistic regression, support vector machine or k-nearest neighbors [MCCD13, CSSS16a].
Another approach called Med2Vec [CBS+16a] also intends to learn meaningful representations of EHR. The intention of Med2Vec [CBS+16a] is to predict the characteristics of possible future visits and the progress of a patient's health state. Choi et al. [CBS+16a] propose a two-fold representation learning structure. Med2Vec therefore considers the sequential property of EHR data that arises from subsequent visits and the co-occurrence of medical codes of each visit. At first, representations of medical codes are obtained and then a visit representations is derived by including demographic information about the patient. The code representations are learned by skip-gram [MCCD13]. To investigate the interpretability of the learned representations, medical experts reviewed and approved the obtained representations. In consequence, Med2Vec leads to superior predictive performance and meaningful interpretable representations.

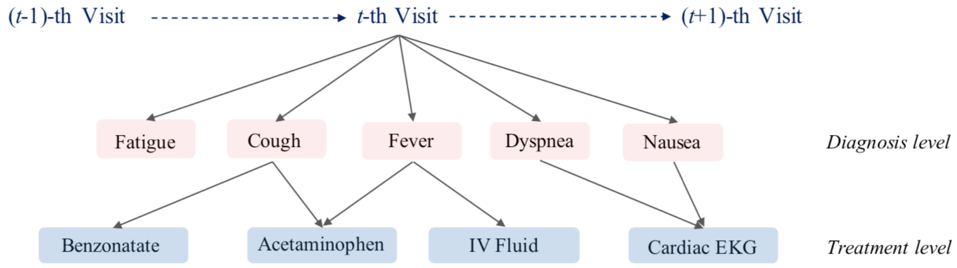


Figure 5: Multilevel structure in MiME [CXSS18]. A patient record consists of a time series of hospital visits. Each hospital visit is composed of a set of diagnosis codes and a set of treatment codes. The graph structure is used to learn a hierarchical representation of visits.

In previous approaches such as Med2Vec, [CSSS16a], [FWH+16], [TNPV15] [CBS+16b], [CBS+16c], [CSSS16b] [CBS+15], [CBK+16] [MCZ+17] and [jac17] EHR is treated as a flat-structured bag of features. In contrast, an approach called MiME [CXSS18] exploits the inherent multilevel structure of EHR to perform prediction tasks, such as heart failure prediction and sequential disease prediction. MiME is applied to a private data set that explicitly contains causal relations between diagnosis and drug codes and allows to model the data as illustrated in Figure 5. Choi et al. [CXSS18] assume that the multilevel structure of EHR reflects a doctor's decision process and contains valuable information that could enhance prediction quality. The approach MiME models a visit as a hierarchical graph with several levels: The visit level, the diagnosis level and the treatment level. The multilevel graph structure is turned into a multilevel representation. This representation is created by training the weight matrices of a neural network. Code representations are subsequently pooled together in a bottom-up manner in order to derive a final visit representation. On the one hand, binary outcome variables are used as labels. On the other hand, Choi et al. [CXSS18] jointly conduct auxiliary prediction tasks that refer to the multilevel structure of EHR. The auxiliary tasks are helpful in order to find general-purpose representations which are not limited to a specific prediction task. In their work, Choi et al. [CXSS18] show that the multilevel representation improves learning efficiency even with a limited data volume. In the conducted study, MiME outperforms all baselines including other deep learning approaches on EHR, such as Med2Vec [CBS+16a] or GRAM [CBS+16c].

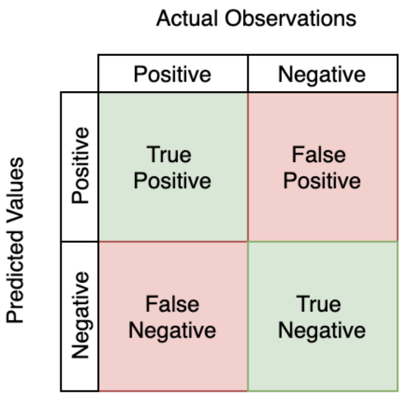
6 Performance Metrics

In order to evaluate the performance of a model and compare different models to each other, an appropriate performance metric is necessary. As already described, prior to the training process, the data set is randomly split into three independent divisions: the train set, the validation set and the test set. Models are learned on the train set and their performance is evaluated on the validation set during training. Finally, the model is evaluated on the test set. To assess generalization performance, the performance metrics on the test set are considered when models are compared to each other.

6.1 Threshold-Based Performance Metrics

In order to receive binary output classes, prior scores or so-called logits produced by a model have to be compared to a threshold. The classification of the samples depends on the choice of the threshold. Threshold-based performance metrics can only be calculated on binary outputs and not on the logits.

The most common performance metrics in machine learning are based on the confusion matrix which displays the quantities of correctly and incorrectly predicted samples of each class for binary classification problems. Samples are categorized as False Positives (FP), False Negatives (FN), True Positives (TP) and True Negatives (TN) predicted values, as illustrated in Figure 6. FP is also called false alarm and FN missed values [SR15].



	Positive	Negative
Positive	True Positive	False Positive
Negative	False Negative	True Negative

Figure 6: Confusion matrix of a binary classification problem. The predicted values are compared to the true observations and can be put in the four categories: TP, FP, FN and TN.

Several performance metrics can be derived from the confusion matrix in order to measure the performance of a classifier. Accuracy, sensitivity, specificity, precision and recall are some widespread metrics:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FN} + \text{FP} + \text{TN}} \tag{23}$$

$$\text{Sensitivity} = \text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{24}$$

$$\text{Specificity} = \text{TNR} = \frac{\text{TN}}{\text{TN} + \text{FP}} \tag{25}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{26}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{27}$$

Sensitivity can also be described as True Positives Rate (TPR) and specificity as True Negatives Rate (TNR).

6.2 Threshold-Free Performance Metrics

Instead of evaluating a model based on the predicted binary output classes, the logits can be examined. Performance metrics related to logits are threshold-free, since they judge the performance of a model regardless of the chosen threshold.
An example for a threshold-free metric based on the confusion matrix is the Receiver Operating Characteristic (ROC) curve. The plot of the ROC displays sensitivity, see Equation 24, on the y-axis and 1 - specificity, see Equation 25, on the x-axis for all possible thresholds. The sensitivity is inversely related to the specificity and the ROC curves displays the trade-off between both metrics. From the ROC, the Area Under the Receiver Operating Characteristic Curve (AUC-ROC) can be obtained. The ROC curve of a perfect classifier goes from the origin straight up the y-axis, crosses the point $(0, 1)$ and leads to the point $(1, 1)$. In this case, the AUC-ROC equals to one. A random classifier is displayed as a straight line connecting the points $(0, 0)$ and $(1, 1)$. The AUC-ROC of this model would be 0.5. Most classifiers range between both extremes and return an AUC-ROC within the interval $[0.5, 1]$ [HT13].
Another threshold-free performance metric is the Area Under the Precision Recall Curve (AUC-PR). In comparison to the ROC curve, the precision-recall curve only considers correct predictions by plotting the precision, see Equation 26, on the y-axis and the recall, see Equation 27, on the x-axis.

6.3 Performance Metrics for Imbalanced Classes

Especially, when machine learning is applied to a problem with imbalanced outcome classes, the choice of a suitable performance metric is crucial. Classes are imbalanced if one outcome variable is significantly overrepresented in the underlying data set. Examining the accuracy can be misleading in that case, since a model might return a high accuracy by always predicting the overrepresented class. Furthermore, the AUC-ROC is not appropriate for highly imbalanced class distributions. While the value of the AUC-ROC for a random model equals to 0.5, the baseline value of the AUC-PR equals to $\frac{\text{P}}{\text{P} + \text{N}}$ with P and N as the numbers of positive and negative elements and consequently depends on the class distribution. The AUC-PR is recommended when dealing with a highly imbalanced data set [SR15]. More information on performance metrics can be found in [Wan20] and [GBC].

Part III

Contributions

7 Premier Healthcare Database

The data examined in this work is collected in the PHD which is the largest clinical databases in the United States [HZRS15]. PHD is a dynamic database which is updated quarterly. The data submitted by hospitals and healthcare institutions to PHD runs through several quality and validation checks. Thus, the data can be considered as a robust research tool. Since a diverse selection of hospitals is included, the data is assumed to be representative for the USA. The database consists of standard hospital discharge files, demographic and disease state information as well as information on billed services, such as medication, laboratory, diagnostics and therapeutic services [GCL+18].

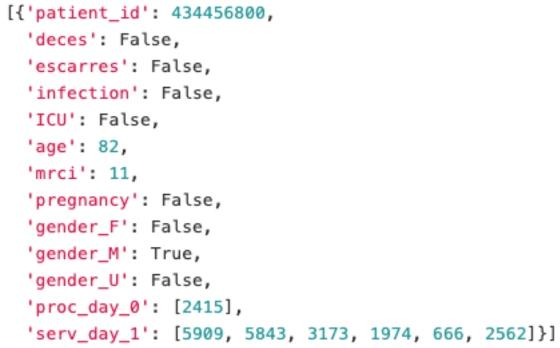
7.1 Preprocessing

In this work, a data set from PHD collected in the year 2006 is used. It contains records from 417 hospitals in the USA from around one million of patients over a period of twelve months. Patients are tracked with a nine-digit patient identifier which is unique per hospital. In the present data set, the identifier is unique over the whole data set. At first, patient data is joined with billing data that contains information on drug prescriptions and then joined with International Code of Diseases (ICD) data. ICD codes identify therapeutic and diagnostic procedures [FGLB18].

In the following, some filters are applied. At first, outpatients and consultations are removed, since patients with these admission types do not stay in the hospital for more than one day. Patients with admission types newborn and pregnancy are removed considering that these encounters are not related to diseases. The records of patients with admission type surgery are excluded, since medical prescriptions in surgery vary considerably and are extremely more complex than non-surgery admissions. In the present data set, all patients above the age of 89 are automatically grouped the same age category of above 89. In this case, the exact age is unknown. For that reason, all patients in this age category are removed. Besides, encounters without any prescribed drugs or stays shorter than three days are filtered. The reason for this filter is that prescribed drugs are essential for the conduction of the prediction task and the stay should be long enough to observe the outcome. To improve readability and performance, drug codes and procedure codes are indexed to integer values. Then, duplicated codes are removed. Afterwards, code lists are shuffled randomly, since the order of the code list per stay is irrelevant. After the application of the mentioned filters, the remaining set contains 1,271,733 records.

Since the goal of this work is to predict mortality based only on ICD codes and drug codes known on the day of hospital admission, the codes tracked on other days but admission day are deleted. To apply GCT, every record must contain a list of administered drugs and ICD codes with at least one code each. Encounters without ICD codes are removed. In the present data set, all patients have at least one prescribed drug on admission day. The list of administered medication is double checked in order to keep only prescriptions that are drugs.

Previously, general prescriptions such as those of medical accessories were included. After applying these additional filters, the number of patients is reduced from 1,271,733 to 885,241. In the next step, type conversions are conducted, such as converting strings to categorical values. After all, an example encounter can be identified as displayed in Figure 7:



```
[{'patient_id': 434456800,
  'deces': False,
  'escarres': False,
  'infection': False,
  'ICU': False,
  'age': 82,
  'mrci': 11,
  'pregnancy': False,
  'gender_F': False,
  'gender_M': True,
  'gender_U': False,
  'proc_day_0': [2415],
  'serv_day_1': [5909, 5843, 3173, 1974, 666, 2562]}]
```

Figure 7: An example record in the underlying data set.

According to Figure 7, an encounter is uniquely identified by the `patient_id`. `Deces` (engl. Death), `escarres` (engl. pressure ulcers), `infection` (refers to hospital-acquired infections) and `ICU` (refers to admission to Intensive Care Unit (ICU)) are binary outcome variables that declare whether a patient died, developed pressure ulcers, caught an hospital-acquired infection or was admitted to ICU during the hospital stay. `Age` and `mrci` contain integer values. The Medication Regimen Complexity Index (MRCI) represents the complexity of the administered medication by aggregating information about drugs in a single score. The minimum value of the MRCI is zero and no maximum value exists. A larger MRCI demonstrates a higher risk of in-hospital mortality. `Pregnancy`, `gender_F`, `gender_M` and `gender_U` are binary and declare if a patient is pregnant and the patient's gender. `proc_day_0` contains the list of ICD codes, while `serv_day_1` contains the list of administered drugs.
Geneves et al. [GCL+18] examined whether the distribution of outcome variables depends on the hospital ID and biases the prediction. On average, the mean number of submissions per hospital equals to 3,568 in the present data set. Outliers only emerge from hospitals which submitted significantly fewer encounters than the other hospitals to the PHD [GCL+18]. Further, Choi et al. [CXL+19] recommend to limit the list of ICD and drug codes to a maximum of 50 codes each in order to avoid the matrices that have to be processed by the network of becoming to large. By consequence, encounters with lists exceeding 50 codes are removed. After performing the described steps, a data set of 885,191 patients remains.

7.2 Imbalanced Data Set

In the underlying data set, the quantitative representations of dead and alive patients are highly imbalanced. Out of 885,241 patients 28,236 patients die, which leads to a prevalence of 0.033.
Machine learning on imbalanced classes can be challenging [BPM04]. Models are usually trained by maximizing the accuracy, see Section 6. In case of imbalanced classes, a model can achieve a high accuracy by always predicting the overrepresented class and never predicting the underrepresented class. Even though the accuracy might be high in this case, the model is useless and the accuracy as performance metric is misleading.
Several strategies exist in order to handle imbalanced data sets and prevent a model from solely predicting the overrepresented class. One solution is to rebalance the data set by undersampling the major class or oversampling the minor class, for example, by duplicating records.

	Total Size of Set	**Number of Dead Patients**	**Percentage of Dead Patients**
Train Set	37,888	18,925	49.9%
Validation Set	9,294	4,613	49.6%
Test Set	9,290	4,698	50.6%

Table 1: The frequency distribution of dead patients after undersampling the overrepresented class. The train, the validation and the test set are all nearly balanced to a prevalence of 50%.

As well as in [GCL+18], the overrepresented class of alive patients is randomly undersampled, while the class of dead patients is fully kept to receive a balanced data set. This step reduces the vocabulary of used drug codes from 18,002 to 7,334. Out of 19,134 total codes defined in the ICD standard, 5,094 are used in the filtered data set. One major reason why undersampling is an appropriate strategy to handle the imbalance in this case, is the reduction of the code vocabulary. An element of the later described method is the calculation of conditional probabilities based on the co-occurrence of medical codes. If the entire imbalanced data set was kept, the matrix containing the conditional probabilities would not fit in the memory. Consequently, undersampling the class of alive patients solves two problems at once [BTR15, Roc19, Bro19a, Cha10, HG09, Sei18].

8 Method

The objective of this work is to predict patient mortality during hospital stays based on ICD codes and drug codes available on the first day of hospital admission. The binary outcome variable death declares if a patient died during the hospital stay. It equals to one in case of death, otherwise zero. The goal is to make predictions based on representations of medical codes which reflect the inherent graph structure of EHR as introduced in MiME. Since the underlying data set in this work does not directly contain causal relations that describe a graph, MiME [CXSS18] is not applicable. In consequence, GCT [CXL+19] is used to firstly learn the hidden graph structure of EHR, then derive representations that reflect the graph structure and finally use these representations to conduct mortality prediction. To assess the performance of GCT on the present data set, it is compared to two baselines models logistic regression and Transformer that are fitted or trained on the same present data set. This section describes the methodological foundations of GCT and the baselines.

8.1 Logistic Regression

Fejza et al. [FGLB18] showed that logistic regression is appropriate to obtain interpretable and accurate predictions on a data set from PHD based on the ICD and drug codes at hospital admission. Logistic regression is often applied to classification tasks in a medical or biostatistical context, since it generates interpretable results. Logistic regression computes class probabilities that lie in an interval of $[0, 1]$ and sum up to one [Has17]. The function in Equation 28 is minimized to fit a logistic regression to the training data:

$$f(w) = \lambda R(w) + \frac{1}{n} \sum_{i=1}^{n} L\left(w; x_i, y_i\right) \tag{28}$$

The number of patients is indicated as n and w is the vector of weights that is estimated. Records of the train set are declared as $x_i \in R^d$. Each patient vector contains d binary values that indicate whether an ICD or drug code occurs in an encounter (1) or not (0). The predicted outcome variable death is described as $y_i \in \{0, 1\}$ which equals to 1 if a patient died and 0 if not. The term $\lambda R(\boldsymbol{w})$ regularizes the model in order to avoid overfitting on the training data.

The logistic regression takes the binary outcome variable death and a joined list of indexed ICD and drug codes as input. The model is implemented in Spark using the logistic regression function in the classification module of the pyspark.ml package [ZCF+10]. The model is fit to the training data with the parameters recommended by [GCL+18]. More details on the method can be found in [FGLB18] and [Has17].

8.2 Transformer and Graph Convolutional Transformer

Choi et al. [CXL+19] propose to learn the structure of EHR data with an approach related to Transformer [VSP+17] including the attention mechanism, as described in Section 4.8. According to [CXL+19], Transformer is in this case applied as a graph embedding algorithm that starts with a fully connected graph and learns important connections between nodes. In the following, the term Transformer baseline refers to the application of Transformer as a graph embedding algorithm and not to the original Transformer approach of Vaswani et al. [VSP+17], unless it is explicitly mentioned. Choi et al. [CXL+19] argue that a GNN can be interpreted as a special case of Transformer that has a fixed adjacency matrix instead of the adjacency matrix learned with attention. Simultaneously, Transformer can be seen as a useful tool for learning an adjacency matrix using attention, if the true adjacency matrix is unknown [CXL+19]. Once the graph structure is learned, GNNs or MiME can be applied to find a representation that utilizes the learned graph structure and captures it in a real-valued vector which is the visit representation in this case. Finally, the visit representation can be used to conduct arbitrary prediction tasks. Choi et al. [CXL+19] use the attention mechanism of Transformer to learn the adjacency matrix, or so-called attention matrix of graphs. GCT and the Transformer baseline have in common that both models learn the adjacency matrix of the hidden graph structure based on the attention mechanism. When the graph structure is obtained, the following steps of learning representations based on convolutions over the structure and making predictions are identical. While the Transformer baseline learns the adjacency matrix without any guidance, restrictions or targeted initialization, GCT can be seen as a extension of Transformer with restrictions and guidance on EHR-specific properties.

In comparison to original Transformer which is applied to machine translation tasks [VSP+17], medical codes are sets instead of sequences. Hence, the position of a code in the list of codes is not relevant. By consequence, positional encoding is omitted. Furthermore, Transformer and GCT do not contain a decoder since the encoded representations are directly used to solve a prediction task instead of being decoded to an output sequence as in machine translation.

The main idea of GCT and the Transformer baseline is to model ICD and drug codes together in the same latent space as an acyclic directed graph. Each ICD or drug code corresponds to a node in the graph. The input data set, see Section 7, contains encounters of patients that are admitted to the hospital. The data set is notated as a set of visits $\mathcal{V} = \{v_0, v_1, ..., v_n\}$ with n number of visits. Each visit is treated as a single encounter. The potential time dimension of subsequent visits of the same patient is neglected, since visit histories are not transparent. Each encounter contains a set of ICD codes $\{d_1,\ d_2,\ ....,\ d_{|l|}\}$ and a set of drug prescriptions $\{m_1,\ m_2,\ ....,\ m_{|m|}\}$. The number of ICD and drug codes of a visit are indicated as $|l|$ and $|m|$. The vector representation of a node is denoted as c_i. Considering this input format, the graph of a visit v_i consists of different layers:

- Visit node that contains final representation v_i
- ICD nodes that contain ICD codes of a visit: $\{d_1,\ d_2,\ ....\ ,\ d_{|l|}\}$
- Drug nodes that contain drug codes of a visit $\{m_1,\ m_2,\ ....\ ,\ m_{|m|}\}$

Regarding all nodes of a graph, their representations are denoted as matrices. In order to obtain the graph structure of medical codes, GCT and the Transformer baseline start with a fully connected graph linking all nodes of an encounter. In the following, meaningful connections are learned using attention, see Section 4.8. An encoder with single-head attention is described in the following:

$$\mathbf{C}^{(j)} = \mathrm{MLP}^{(j)} \left(\mathrm{softmax} \left(\frac{\mathbf{Q}^{(j)} \mathbf{K}^{(j)\top}}{\sqrt{d}} \right) \mathbf{V}^{(j)} \right) \tag{29}$$

The node representation of the j-th encoder layer is denoted as $\mathbf{C}^{(j)}$. The MLP can in interpreted as a graph convolution of the j-th encoder layer, see Section 4.9.2 that aggregates over the neighboring nodes based on the obtained adjacency matrix. The queries, keys and values as defined in Section 4.8 are calculated with trainable weight matrices $\mathbf{W}_K^{(j)}$, $\mathbf{W}_Q^{(j)}$ and $\mathbf{W}_V^{(j)}$ and d as their column size as follows:

$$\mathbf{Q}^{(j)} = \mathbf{C}^{(j-1)} \mathbf{W}_Q^{(j)} \ , \quad \mathbf{K}^{(j)} = \mathbf{C}^{(j-1)} \mathbf{W}_K^{(j)} \ , \quad \mathbf{V}^{(j)} = \mathbf{C}^{(j-1)} \mathbf{W}_V^{(j)} \tag{30}$$

To better understand Equation 29, it is helpful to compare it to the formula of representation learning when the graph structure is obvious, as Xu et al. [XHLJ18] propose:

$$\mathbf{C}^{(j)} = \text{MLP}^{(j)} \left(\tilde{\mathbf{D}}^{-1} \tilde{\mathbf{A}} \mathbf{C}^{(j-1)} \mathbf{W}^{(j)} \right) \tag{31}$$

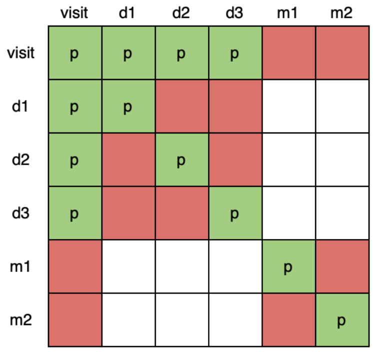
The matrix $\tilde{\mathbf{A}}$ describes the adjacency matrix including self-connections, described by the identity matrix $\mathbf{I}$:

$$\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I} \tag{32}$$

The addition of $\mathbf{I}$ to the adjacency matrix makes sure that the target node itself is not neglected when the neighborhood is aggregated. The division by the diagonal node degree matrix $\tilde{\mathbf{D}}$ can be interpreted as row-wise normalization. $\mathbf{C}^{(j)}$ corresponds to representation of the encoder layer of step j and $\mathbf{W}^{(j)}$ are the parameters of the respective encoder layer [XHLJ18, CXL+19]. Comparing Equations 31 and 29, a correspondence between the normalized adjacency matrix $\tilde{\mathbf{D}}^{-1}\tilde{\mathbf{A}}$ and the attention matrix softmax $\left(\frac{\mathbf{Q}^{(j)}\mathbf{K}^{(j)T}}{\sqrt{d}}\right)$ as well as between the node embeddings $\mathbf{C}^{(j-1)}\mathbf{W}^{(j)}$ and the value vectors $\mathbf{C}^{(j-1)}\mathbf{W}_V^{(j)}$ can be detected [CXL+19].

8.2.1 Graph Convolutional Transformer

GCT provides further regularization for the attention mechanism of Transformer presented in the previous section and includes EHR-specific characteristics.
The attention mechanism in Transformer regards all possible connections between nodes to find meaningful links. GCT introduces some restrictions to the links in the graph. In the initial encoder, some connections are not allowed, while others are guaranteed to exist. Connections between the root node visit and ICD codes, as well as the diagonal of the adjacency matrix are guaranteed to exist. These guaranteed connections between the root node visit and all ICD nodes make sure that the root node includes the whole underlying graph in the representation. The diagonal in the adjacency matrix corresponds to self-connections of the nodes. Initially forcing them to exist has the effect that a node includes its own representation of the previous encoder layer when aggregating over its neighborhood. Connections between codes on the same layer as well as connections skipping one layer in the hierarchical graph are restricted in order to keep the learned graph structure comparable to the hierarchical graph of MiME [CXSS18] and limit the search space.



	visit	d1	d2	d3	m1	m2
visit	p	p	p	p	🟥	🟥
d1	p	p	🟥	🟥		
d2	p	🟥	p	🟥		
d3	p	🟥	🟥	p		
m1	🟥				p	🟥
m2	🟥				🟥	p

Figure 8: The adjacency matrix of GCT. The self-connections on the diagonal and the connections between the visit node and the nodes of the ICD codes are in the initial encoder layer guaranteed to exist. These cells in the matrix are initialized with a prior scalar. In contrast, connections between the visit node and nodes of drug codes, as well as connections between nodes of the same layer are masked. The white cells are initialized with conditional probabilities.

In order to prevent the attention mechanism from focusing on the restricted connections, a mask $\mathbf{M}$ is added. The mask contains $-\infty$ for forbidden connections and zeroes for eligible connections. The values of the guaranteed connections in the adjacency matrix are initialized with a prior scalar that lies in the interval of $[0, 1]$ and determines the initial strength of the self-connections. For instance, a value of one puts a high focus on the self-connections and inhibits connections to other nodes in the initial encoder layer. In this work, the prior scalar is set to 0.5, as recommended by [CXL+19]. The restrictions on the adjacency matrix are illustrated in Figure 8. While Transformer starts with a random values sampled from a uniform distribution as initialization of the adjacency matrix, Choi et al. [CXL+19] argue that conditional probabilities of codes serve as a more appropriate initialization of the adjacency matrix. Intuitively, starting with conditional probabilities is reasonable, as the following example illustrates. A hospital stay as presented in Figure 9 that contains the following ICD and drug codes is considered:

- ICD codes = {d1, d2, d3}
- Drug codes = {m1, m2}

In the beginning, it is unknown whether m1 is prescribed for d1 or d2. However, if P(m1|d1) is larger than P(m1|d2), it is intuitively more likely to have a link between m1 and d1 instead of a link between m1 and d2 in the graph structure. Consequently, a guiding matrix $\mathbf{P} \in [0.0, 1.0]^{|c| \times |c|}$ containing conditional probabilities P(drug code | ICD code) is used. Before starting the training process, the conditional probabilities of co-occurring codes are calculated on the train set. The adjacency matrix is initialized with the precalculated conditional probabilities to guide the learning process. Expressed in formulas, matrix $\mathbf{P}$ serves as the adjacency matrix in the initial encoder layer $j = 1$.

$$\mathbf{C}^{(j)} = \text{MLP}^{(j)} \left(\mathbf{P}\mathbf{C}^{(j-1)}\mathbf{W}_V^{(j)} \right) \text{ if } j = 1 \tag{33}$$

In the following encoder layers $j > 1$, $\mathbf{P}$ is replaced by the matrix $\hat{\mathbf{A}}^{(j-1)}$ which contains the estimated attention weights of the previous encoder layer $j - 1$.

$$\mathbf{C}^{(j)} = \text{MLP}^{(j)} \left(\hat{\mathbf{A}}^{(j)}\mathbf{C}^{(j-1)}\mathbf{W}_V^{(j)} \right) \text{ if } j > 1 \tag{34}$$

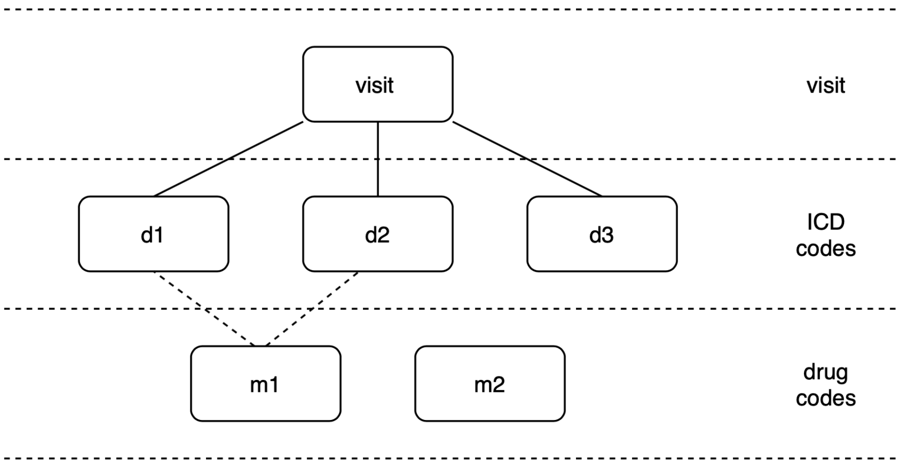


Figure 9: The inherent underlying graph structure of EHR. The graph consists of a root node that captures the final visit representation. ICD codes and drug codes are modeled as layers below. Initially, GCT forces links between the visit node and all ICD codes to exist. Connections within one layer (for example, m1 ↔ m2), and connections skipping one layer (for example, visit ↔ m1) are masked [CXL+19].

As mentioned above, the adjacency matrix is masked by the matrix $\mathbf{M}$. The matrices $\mathbf{P}$, $\mathbf{M}$ and $\hat{\mathbf{A}}^{(j)}$ are all of size $|c| \times |c|$. The adjacency matrix $\hat{\mathbf{A}}^{(j)}$ of encoder layer j is calculated as described in Equation 35. The softmax function turns logits into probabilities that sum to one.

$$\hat{\mathbf{A}}^{(j)} := \text{softmax}\left(\frac{\mathbf{Q}^{(j)}\mathbf{K}^{(j)\top}}{\sqrt{d}} + \mathbf{M}\right) \tag{35}$$

In order to optimize GCT, cross-entropy is used as a loss function. Cross-entropy is a common loss function for classification tasks as it calculates the error between the predicted and the ground truth outcome variable during training. It is defined as:

$$H(y) = -\sum_i y'_i \log(y_i) \tag{36}$$

The ground truth outcome is indicated as y'_i and y_i describes the outcome predicted by the classifier. The values are summed over all samples in the mini-batch [Wan20]. If all samples are correctly classified, the cross-entropy function will return zero.
To avoid that the learned adjacency matrices differ largely from encoder layer $j-1$ to encoder layer j, regularization is applied by adding the Kullback Leibler divergence (KL divergence) [SLM+15, Pri10] between the adjacency matrices in encoder layer j and $j-1$ to the loss function. In general, the KL divergence D_{KL} between two probability distributions $p(x)$ and $q(x)$ is defined as follows:

$$D_{KL}(p\|q) = \sum_x p(x) \log \frac{p(x)}{q(x)} \tag{37}$$

More details on the KL divergence can be found in [Pri10]. Considering GCT, the estimated adjacency matrix $\hat{\mathbf{A}}^{(j)}$ is compared to $\hat{\mathbf{A}}^{(j-1)}$ of the previous encoder layer. In the first encoder layer, $\mathbf{P}$ is the reference.

$$L_{reg}^{(j)} = D_{KL}\left(\mathbf{P}\|\hat{\mathbf{A}}^{(j)}\right) \text{ when } j = 1 \tag{38}$$

$$L_{reg}^{(j)} = D_{KL}\left(\hat{\mathbf{A}}^{(j-1)} \| \hat{\mathbf{A}}^{(j)}\right) \text{ when } j > 1 \tag{39}$$

The initial loss function L_{pred} equals to the cross-entropy $H(y)$ in Equation 36. The regularization terms of each iteration are summed up. Subsequently, the sum is multiplied by the coefficient λ:

$$L = L_{pred} + \lambda \sum_j L_{reg}^{(j)} \tag{40}$$

This factor λ is a tunable hyperparameter that controls the regularization of GCT. In their work [CXL+19], Choi et al. recommend values in a range from 0.01 to 100 for λ. Further details on Transformer and GCT can be found in [VSP+17] and [CXL+19].

After describing the functionality of GCT in details, the data flow of GCT can be illustrated as depicted in Figure 10.

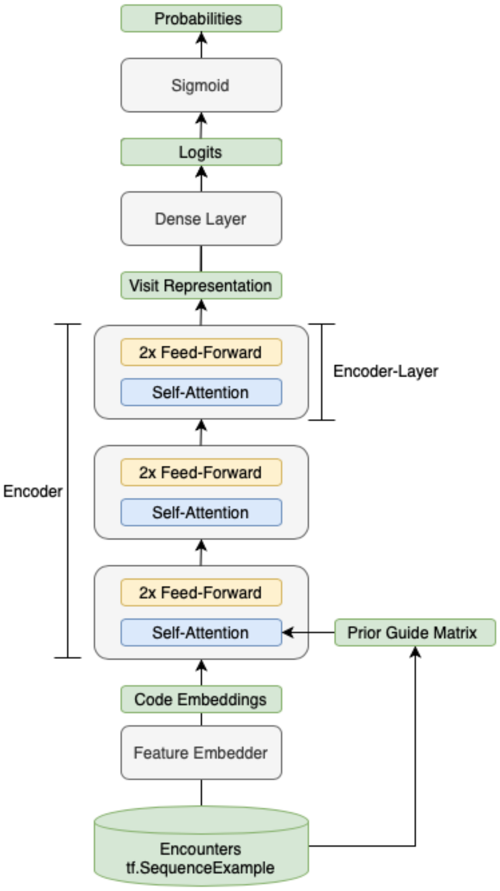


Figure 10: The architecture of GCT. Initially, medical codes are embedded before being fed into the encoder which contains three encoder layers. The attention component within each encoder learns the graph structure. The two feedforward components represent graph convolutions and return representations based on the learned graph structure. In the final steps, a final visit representation is processed to probabilities.

As data format, TFRecords are used which are appropriate for storing sequence data and context data, see Section 3. ICD codes and drug codes of patients can be described as order-invariant lists. A Sequence Example record contains context features and feature lists as attributes. Feature lists contain the sets of codes, while context features contain single values instead of sets and refer to the entire encounter. The context is used to store generic information about the encounter, such as ID, age or gender [AAB$^+$15, Gam18]. In the beginning, the data is processed by a feature embedder, see Section 4.4, which transforms one-hot-encoded vectors into real-valued vectors. The output of the feature embedder are code representations in the shape of

$$(\text{ mini-batch size},\ 2*\ \text{maximum number of code sequence } +1,\ \text{embedding size }) \tag{41}$$

as illustrated in Figure 11. The input to each encoder layer is of identical shape. The first dimension is described by the mini-batch size. The second dimension covers ICD and drugs codes with the definable maximum sequence length of 50 codes for each code type plus the visit representation vector. Each original code in the matrix is mapped to its representation of embedding size. Sequences shorter than 50 are padded with zeroes.

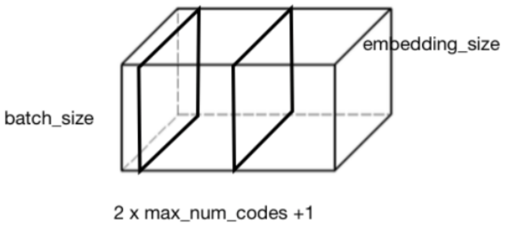


Figure 11: Illustration of the input to a encoder layer with the shape of (mini-batch size , $2*$ maximum number of code sequence $+1$, embedding size). The parameters mini-batch size, maximum number of code sequence and embedding size are set independently of training.

Choi et al. [CXL$^+$19] argue that three encoders are sufficient to learn the graph structure. This approach also uses three encoders, as illustrated in Figure 10. Each layer comprises an attention component and two feedforward components. Layer normalization [BKH16], see Section 4.6.4, is applied to the feedforward component as well as on the attention component of each encoder. The attention component estimates the adjacency matrix for the hidden graph structure and the feedforward components aggregate the nodes reflecting the learned adjacency matrix. The aggregated representation of an encoder layer is then passed to the next encoder layer. Each encoder layer, except for the uppermost, reproduces an output with a shape as indicated in Figure 11. Except for the last encoder layer, all layers use ReLU as activation function. The feedforward component of the last encoder employs linear activation $f(x) = x$ instead of ReLU. The representation created by the uppermost encoder is processed by a dense layer [C$^+$15] that produces logits. The sigmoid function, see Equation 2, turns these logits into class probabilities and enables the model to predict the mortality for a hospital stay.

9 Experiments

In this work, mortality prediction is conducted based on ICD and drug codes known on the first day of a hospital stay on a large data set from PHD.
Before testing, the data set is filtered, preprocessed and rebalanced as described in Section 7. Subsequently, the data set is randomly split in three independent sets, the train, the validation and the test set, in the ratio 8:1:1. Then, the baselines logistic regression and Transformer as well as GCT are trained and tested on the same division of the data. All models obtain identical train, validation and test sets as input to guarantee comparability. The AUC-PR and AUC-ROC of the models on the test set are compared. Since the data set is balanced, both measures are applicable.
This Section describes the training details, summarizes the results and observations of the conducted experiments. At first, the results of the baselines logistic regression and Transformer are presented. In the following, the results of GCT illustrated and compared to the baseline models.

9.1 Training Details

GCT and Transformer are implemented in Python 3 with use of the libraries TensorFlow [AAB+15] version 1.15 and Keras [C+15], see Section 3. The training is modeled in TensorFlow Estimator [XMS+17]. The model is trained and evaluated on a server with the following characteristics:

- 2x CPUs of 20 cores each (Intel(R) Xeon(R) Silver 4114 CPU at 2.20GHz)
- 128 GB RAM
- 1x GPU Nvidia Tesla P4 8 GB with driver 418.56 - CUDA 10.1 and CuDNN
- Ubuntu 16.04 LTS as host system

The logistic regression is implemented in Spark based the logistic regression method proposed by pyspark.ml [ZCF+10].

9.2 Prediction Performance of Logistic Regression

As baseline, the logistic regression described in Section 8.1 is fit to the train set. The model is applied to the test set in order to predict mortality. A summary of the results is given in Table 2. The baseline leads to an AUC-ROC of 80.0%, AUC-PR of 78.7% and to an accuracy of 72.1% on the test set.

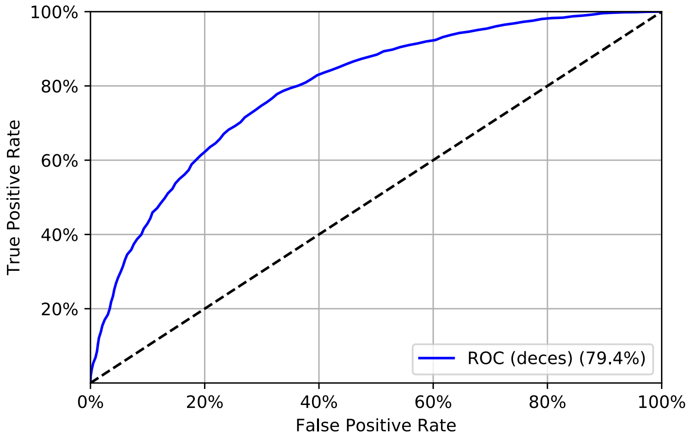


Figure 12: The ROC curve of the logistic regression. The AUC-ROC on the test set equals to 80.0 %.

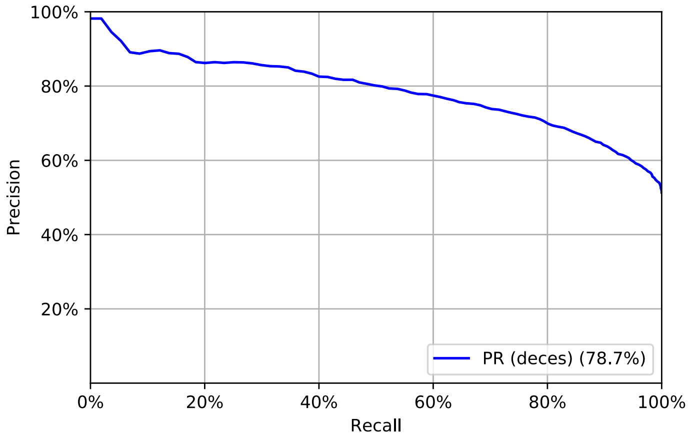


Figure 13: The PR curve of the logistic regression. The AUC-PR on the test set equals to 78.6 %.

# True Positives	2,007
# True Negatives	1,978
# False Positives	713
# False Negatives	831
Sensitivity	70.7 %
Specificity	73.5 %
Accuracy	72.1 %
AUC-PR	78.7 %
AUC-ROC	80.0 %

Table 2: Logistic regression: Overview of the results.

Figures 12 and 13 illustrate the ROC curve and the PR curve. The plotted diagonal in Figure 12 draws the ROC curve of a model that has the quality of a random classifier, as comparison.

9.3 Prediction Performance of Transformer

Transformer is evaluated as a second baseline in order to test whether the EHR-specific adaptions of GCT, see Section 8.2.1 contribute to the model quality. The Transformer approach disregards the prior calculated conditional probabilities on the train set that serve as an intuitive initialization of the adjacency matrix for GCT. Instead, Transformer starts with an initialization following a uniform distribution. The Transformer baseline is trained with the following hyperparameter setting:

- Mini-batch size: 256
- Dropout rate: 0.08
- Embedding size: 128
- Initial learning rate: 0.0033
- Number of Transformer stacks: 3
- Regularization coefficient λ: 0.1
- Number of steps: 5,000
- Optimizer: Gradient Descent

The Figures 14 and 15 display progress of the AUC-PR and AUC-ROC on the validation set during training. After each parameter update in the network, the performance metrics are calculated and plotted. The plots are created by Tensorboard [AAB+15]. Table 3 gives an overview of the AUC-ROC and AUC-PR values on the validation and the test set.

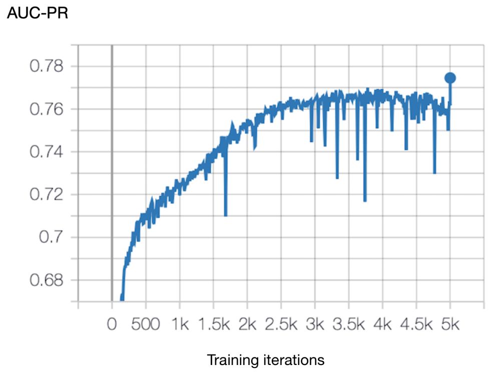


Figure 14: Transformer: The AUC-PR on the validation set during training.

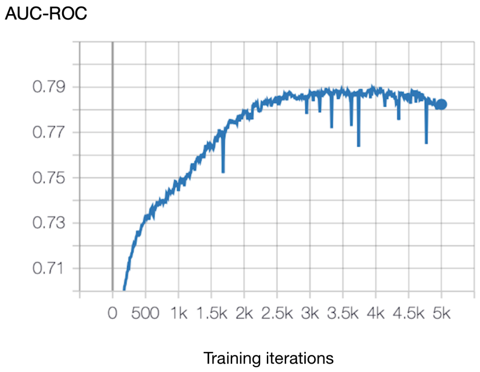


Figure 15: Transformer: The AUC-ROC on the validation set during training.

AUC-PR on the Validation Set	76.2 %
AUC-ROC on the Validation Set	78.3 %
AUC-PR on the Test Set	77.5 %
AUC-ROC on the Test Set	78.2 %

Table 3: Transformer: Overview of the results.

9.4 Prediction Performance of Graph Convolutional Transformer

The architecture of GCT is modeled as described in Section 8.2.1 and in [CXL+19]. Firstly, the conditional probabilities of co-occurring codes are calculated based on the train set. As described in Section 8.2.1, the conditional probabilities serve as the initial adjacency matrix in the first encoder layer and intend to add prior knowledge to the model. GCT is trained with the following hyperparameters:

- Mini-batch size: 256
- Dropout rate: 0.08
- Embedding size: 128
- Initial learning rate: 0.0033
- Number of transformer stacks: 3
- Regularization coefficient λ: 0.1
- Number of steps: 4,000
- Optimizer: Gradient Descent

After each mini-batch update, the model is evaluated on the validation set and the performance metrics AUC-PR and AUC-ROC are computed, see Figures 16 and 17. The losses on the train set (orange) and on the validation set (blue) are plotted in Figure 18 with smoothing coefficient 0.908 [AAB+15]. The decreasing trend of both loss curves indicates that the model is able to learn from the input data during the training process. A significant gap between the curves cannot be observed which shows that the model does not considerably overfit. As well as for Transformer, the plots are created by Tensorboard [AAB+15]. Table 4 gives an overview of the obtained AUC-PR and AUC-ROC values on the validation and the test set.

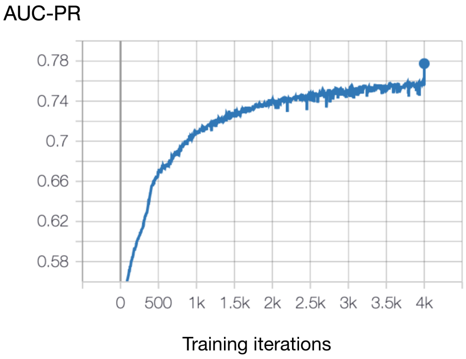


Figure 16: GCT: TheAUC-PR on the validation set during training.

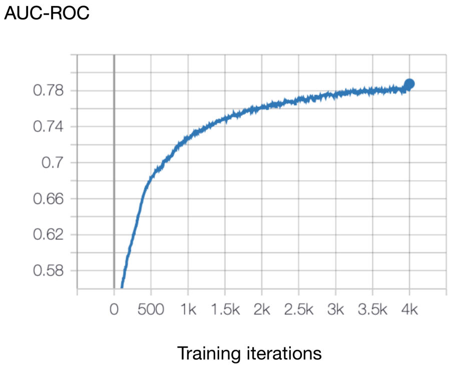


Figure 17: GCT: The AUC-ROC on the validation set during training.

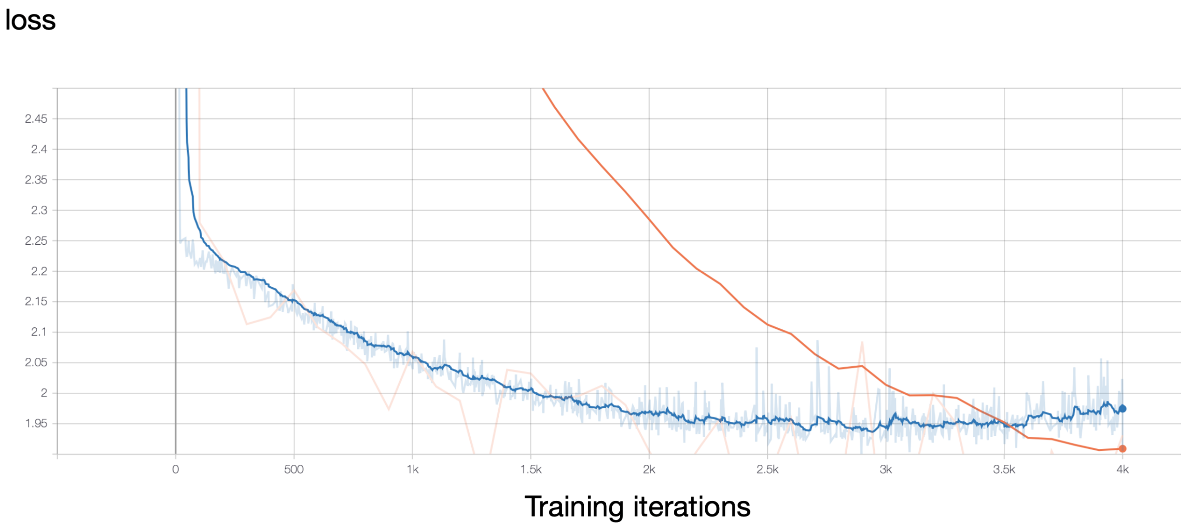


Figure 18: GCT: The loss on the validation set (blue) and loss on train set (orange) during training with smoothing a coefficient of 0.907.

AUC-PR on the Validation Set	75.6 %
AUC-ROC on the Validation Set	78.2 %
AUC-PR on the Test Set	77.4 %
AUC-ROC on the Test Set	78.8 %

Table 4: GCT: Overview of the results of GCT on the validation and the test set.

9.4.1 Selection of Hyperparameters

When Choi et al. [CXL+19] developed their model of GCT, an internal Google framework is utilized to perform hyperparameter tuning. In this work, no exhaustive hyperparameter optimization is conducted. Parameters are varied manually to test if the values recommended by [CXL+19] are reasonable for the present data set and to understand how the network reacts on changes in hyperparameters. As mentioned in Section 4.7, some parameters interact with each other and the exploration of all combinations is not feasible in a manual way. An adequate hyperparameter tuning for this experiment remains future work, see Section 10.5. Nevertheless, the following parameters are tested manually with 10,000 iterations. While one parameter is varied, all other hyperparameters remain constant. The bold parameters are the recommended parameters by [CXL+19]:

- Mini-batch size: {1, 15, **32**, 64, 128, 256, 512}
- Dropout rate: {0, 0.007, 0.05, **0.08**, 0.1, 0.9}
- Embedding size: {32, 64, **128**}
- Initial learning rate: {0.000001, 0.00001, **0.00022**, 0.001, 0.01, 1}
- Number of Transformer stacks: {2, **3**, 4, 6, 12}
- Regularization coefficient λ: {0.01, **0.1**, 1, 10, 100}
- Number of steps: 10,000

- Optimizer: Adam

The detailed results of these runs are listed in the Appendix V. In general, the interaction between mini-batch size and epochs can be observed. A higher mini-batch size leads to slower convergence but more stable training. Since more samples are propagated through the network before updating its weights, more epochs are processed with a higher mini-batch size and a constant number of iterations. The mini-batch size is limited by the size of the memory. For instance, a mini-batch size of 512 can not be processed in this work. The dropout rate as implemented in TensorFlow [AAB+15] determines the probability of elements being randomly set to zero. As expected, an excessive dropout rate of 0.9 leads to an underfitted model that returned an AUC-ROC of 0.5, while low dropout rates increase overfitting. The embedding size determines the vector size of the learned representations for the nodes of the graph. Smaller embedding sizes are tested in order to explore if the model can capture the same knowledge within a smaller vector. As expected, high learning rates lead to divergence and inhibit the model from learning, while extremely low learning rates dramatically slow down training. The number of Transformer stacks determines how many encoder layers are stacked, as described in Section 8. The preliminary tests indicate that more encoder layers do not improve the prediction quality. The influence of conditional probabilities as initialization values for the adjacency matrix of the graph is controlled by the regularization coefficient λ. A low value prevents the learned adjacency matrix from deviating from the initial probability matrix. According to the preliminary tests, the regularization suggest not to have major influence on the prediction performance.

9.4.2 Choice of Number of Training Steps

Initially, the hyperparameters as Choi et al. [CXL+19] used for the mortality prediction on the eICU Collaborative Research Database (eICU) data set are applied in this work to the present data set from PHD:

- Mini-batch size: 32
- Dropout rate: 0.08
- Embedding size: 128
- Initial learning rate: 0.00022
- Number of Transformer stacks: 3
- Regularization coefficient λ: 0.1
- Number of steps: $1,000,000$
- Optimizer: Adam

This results in strong overfitting as shown in Figure 21. The loss on the train set continues to decrease, whereas the loss on the validation set rises. The Figures 19 and 20 also indicate that the AUC-PR and AUC-ROC decrease the more training iterations are performed. Consequently, the model shows poor performance, as illustrated in Table 5.
In conclusion, 1,000,000 iterations are not appropriate and result in strong overfitting on the underlying data set. In order to counteract this issue, the training is stopped after fewer iterations. As already described in Section 4.6.3, this policy is named early stopping and can be regarded as a simple and effective form of tuning the number of steps [Ben12]. In order to determine the number of iterations before stopping, the model with the hyperparameter

AUC-PR on the Validation Set	61.6 %
AUC-ROC on the Validation Set	67.7 %
AUC-PR on the Test Set	63.5 %
AUC-ROC on the Test Set	67.5 %

Table 5: GCT with hyperparameters of the approach [CXL+19]: Overview of the results.

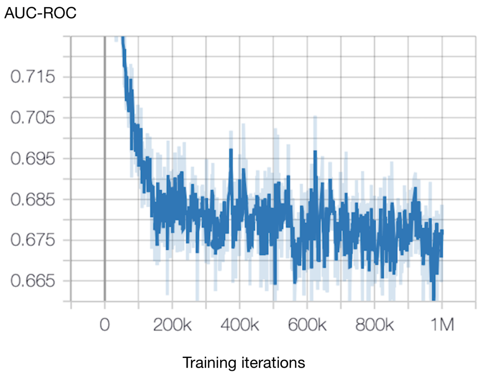


Figure 19: GCT with hyperparameters of the approach [CXL+19]: the AUC-ROC on the validation set during training.

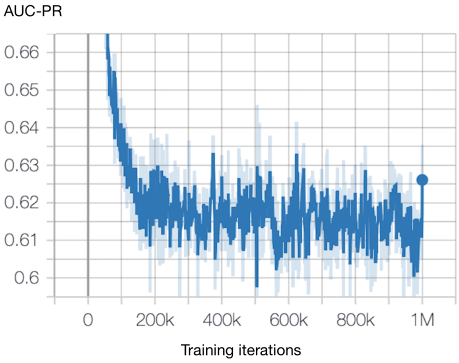


Figure 20: GCT with hyperparameters of the approach [CXL+19]: The AUC-PR on the validation set during training.

combination mentioned in Section 9.4 is trained while observing the smoothed loss curves in Tensorboard. The smoothing is done by exponentially averaging over the loss values and leads to a more stable loss curve. The training is stopped when the smoothed curve of the validation loss showed a significant rise. In theory, at this point, the model will generalize worse and overfit, if training continues. The stopping criterion is met after 4,000 iterations.

9.4.3 Choice of Optimizer

Initially, Adam [KB14] is applied as optimizer, since it is also used in [CXL+19] and is generally recommended by [Rud16] for training complex neural networks fast and efficiently. A performance plateau after 8,000 epochs and a gradually sinking AUC-PR and AUC-ROC curve during training can be observed. With an increasing number of steps, the loss on the validation set drops. This observation points towards overfitting. Wilson et al. [WRS+17] notice that adaptive learning rate optimizers such as Adam [KB14], Adagrad [DHS11] or Adadelta [Zei12] show fast initial progress in training but fail in some applications to generalize on the test set. Furthermore, Keskar et al. [KS17] argue that the automatically adapted learning rate of Adam can be too small for actual convergence and lead to a performance gap between gradient descent and Adam. Gradient descent or momentum [mom99] emerge sometimes as more suitable optimizers in order to avoid overfitting [WRS+17]. With this thought in mind, GCT is trained with gradient descent and Adam as optimizer while keeping the remaining hyperparameters constant for both runs in order to compare the performances of both optimizers.

- Mini-batch size: 256

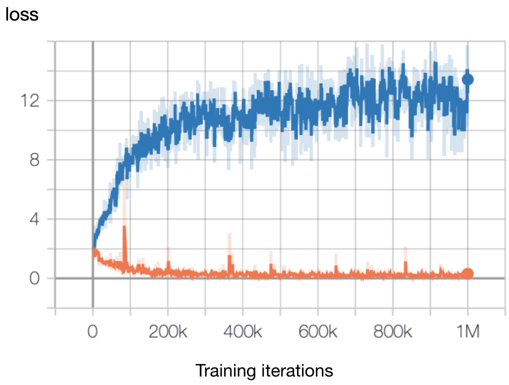


Figure 21: GCT with hyperparameters of the approach [CXL+19]: Loss on the validation set (blue) and loss on the train set (orange) during training.

- Dropout rate: 0.08
- Embedding size: 128
- Initial learning rate: 0.01
- Number of Transformer stacks: 3
- Regularization coefficient λ: 0.1
- Number of steps: 3000

Table 6 shows that gradient descent leads to better results comparing the AUC-ROC and the AUC-PR on both validation and test set. These findings go along with the observations made by Wilson et al. [WRS+17].

Metric	**Adam Optimizer**	**Gradient Descent Optimizer**
AUC-PR on the Validation Set	71.2 %	75.4 %
AUC-ROC on the Validation Set	75.6 %	77.4 %
AUC-PR on the Test Set	73.4 %	77.1 %
AUC-ROC on the Test Set	76.2 %	78.0 %

Table 6: GCT: Comparison of the prediction performance obtained with Adam and gradient descent. Gradient descent leads to superior performance.

9.5 Comparison of Graph Convolutional Transformer to Transformer

Since all models are fitted to the same identical train set and evaluated on the same test set, the obtained AUC-ROC values are comparable.
The comparison of GCT and Transformer enables to investigate whether the EHR-specific restrictions of GCT, see Section 8.2.1, help to learn the true graph structure.

Regarding the AUC-ROC on the test set, no superior performance of GCT in comparison to Transformer can be detected. In the conducted experiments, GCT achieves an AUC-ROC of 78.2 %, while Transformer reaches 78.8% on the test set. The main difference between Transformer and GCT lies in the prior knowledge that is given to the network. While Transformer starts with zero prior knowledge about the adjacency matrix of the graph structure, GCT

initially exploits precalculated probabilities of co-occurring codes on the train set.
This observation leads to the hypothesis that the prior knowledge does not provide essential information that could enhance prediction performance. In their work, Choi et al. [CXL+19] observe the same phenomenon. In the mortality prediction task on the publicly available eICU data set [TJPB18], GCT only slightly outperforms Transformer. Choi et al. conduct readmission prediction as further task and notice a larger advantage using GCT compared to Transformer.
Furthermore, the regularization coefficient λ is tested over the a wide range of recommended values. Even though the tests cannot be considered as an exhaustive exploration, variations in λ do not lead to a large performance difference, see Appendix V. A large λ forces the adjacency matrix to stay close to the conditional probability matrix, while a small value gives room for large divergence between the initial probability matrix and the learned adjacency matrices. Since Transformer employs no restrictions regarding the learned values of the adjacency matrix, GCT with a small value or even zero for λ has similar properties as Transformer.
One can refer from those observations that GCT is not equally helpful for arbitrary prediction tasks. It might depend on the task itself or the underlying data set whether regularization of GCT by introducing precalculated probabilities helps a model to find a conclusive graph structure.

9.6 Comparison of Graph Convolutional Transformer to Logistic Regression

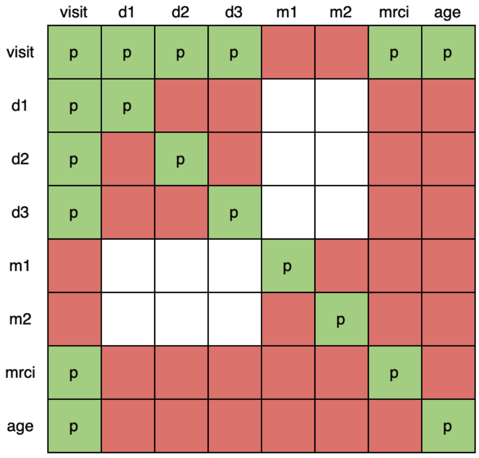
In contrast to Transformer and GCT, logistic regression does not belong to the class of deep learning methods and does also not learn or exploit the inherent graph structure of EHR. Comparing GCT to logistic regression allows to investigate two points. Firstly, the comparison shows whether graph structure enhances prediction quality. Secondly, it can be tested whether deep learning models that find representations of input data are in general suitable for the mortality prediction on the data set of PHD.
The logistic regression results in an AUC-ROC of 80.0 % on the test set, whereas the best overall AUC-ROC obtained with GCT is 78.8 %. Consequently, the logistic regression shows superior performance in comparison to GCT.
The following hypothesis could explain this observation. In theory, a neural network is able to approximate a logistic regression model. In the architecture of GCT, ICD and drug codes are fed into the network, embedded and represented in a hierarchical manner. The representation follows the learned graph structure supported by graph convolution. The logits are calculated by a (logistic) sigmoid function. By contrast, the logistic regression takes the medical codes as raw one-hot-encoded vectors and maps them to logits. From the fact that the logistic regression outperforms GCT in this experiment, one can derive the following assumption. In mortality prediction on the specific underlying present data set, building a representation upon the graph structure might not help the model or even inhibit the model from learning the mortality prediction task. The additional variance introduced by the complex structure of the neural network might complicate the prediction task and lead to inferior performance. Nonetheless, in another prediction task, the graph structure might introduce useful hints to solve classification problems. The extension of GCT on other prediction tasks based on the data set in this work remains future work, see Section 10.

10 Future Work

Without further investigation, the general suitability of GCT as an predictive model applied to EHR cannot be assessed without any degree of certainty. This section explores further extensions of GCT and future experiments that are required to assess the properties of GCT applied to EHR data.

10.1 Extension of Graph Convolutional Transformer with Patient Information

So far, all approaches of GCT are applied to eICU data in [CXL+19] or to PHD data and only make use of medical codes. Since one property of EHR data is its heterogeneity, other data sources could be added to the input data. PHD, for instance, contains additional information, such as MRCI and age that are already proven to serve as meaningful features in the previous work of Geneves et al. [GCL+18]. As expected, Geneves et al. [GCL+18] found a correlation of both MRCI and age to the mortality. The positive correlation to the outcome variable indicates that the variables might enhance the prediction performance of GCT. In contrary, the MRCI is an aggregate of the drug prescriptions. Consequently, a correlation with the prescribed drugs is assumed. It should be tested whether the MRCI provides additional information or introduces unnecessary invariance.



	visit	d1	d2	d3	m1	m2	mrci	age
visit	p	p	p	p	🟥	🟥	p	p
d1	p	p	🟥	🟥			🟥	🟥
d2	p	🟥	p	🟥			🟥	🟥
d3	p	🟥	🟥	p			🟥	🟥
m1	🟥				p	🟥	🟥	🟥
m2	🟥				🟥	p	🟥	🟥
mrci	p	🟥	🟥	🟥	🟥	🟥	p	🟥
age	p	🟥	🟥	🟥	🟥	🟥	🟥	p

Figure 22: The adjacency matrix of GCT with the additional features age and MRCI. The inner matrix is identical to the original approach of GCT. The red cells indicate masked connections, while the green cells are assigned with a prior scalar, as already described in Section 8. MRCI and age are only connected to the visit node.

Consequently, the original GCT approach [CXL+19] can be extended by adding age and MRCI to the model. Each patient contains a MRCI and an age value, see Figure 7 in Section 7. At first, the values of MRCI and age are rescaled to values in the range of [0,1], since large input values can lead to large network weights. During training, large weights can slow down or even prevent convergence. A min-max scaler is used to transform original values x to values x' in an interval of [0,1] [TK01]:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{42}$$

The maximum and minimum values equal x_{max} and x_{min} of the train set for both MRCI and age. To extend the existing structure explained in Section 8, the goal is to include the additional information in the uppermost visit representation that reflects the underlying graph structure and execute mortality prediction on this final representation. Thus, MRCI and age are are modeled as as nodes that are connected to the previously represented graph. The links to the additional nodes are also specified in the adjacency matrix. As well as other edges, the links to additional nodes are restricted. In order to be included in the visit representation, links between the visit node and the age and MRCI node are set to a prior scalar in the initial step. Furthermore, the nodes of MRCI and age are not connected to ICD and drug codes. Consequently, these connections are masked in the adjacency matrix. The initialization of the adjacency matrix can be modeled as depicted in Figure 22 whereas the inner matrix remains as described in Figure 8 of Section 8.2.1. The final implementation of this approach is currently in progress.

10.2 Handling of Imbalanced Classes

In this work, the class imbalance is handled by undersampling the overrepresented class of alive patients. Undersampling has the disadvantage that a large amount of data is not considered. Moreover, the initial class ratio contains valuable information and can be considered as an apriori probability. Since the underlying data set from PHD is especially large compared to publicly available EHR data sets, it would be particularly interesting to take advantage of this data source. Several other techniques exist to cope with imbalanced classes. Consequently, other procedures to handle imbalanced classes should be taken into account as future work.

Choi et al. [CXL+19] also worked with a highly imbalanced data set but did not use undersampling. To counteract the problem of imbalanced classes, the AUC-PR is considered as performance metric. As explained in Section 6, the AUC-PR is invariant to the imbalanced class distribution and makes models comparable without prior resampling.

Furthermore, deep learning models offer the possibility to solve the imbalanced class issue by adapting class weights. The idea behind class weights is to vary the weight for each training sample when calculating the loss. By default, each element carries the weight one. Multiplying samples of the underrepresented class by a weight larger than one gives the sample a higher influence on the loss. The weights can be obtained from the frequency distribution of the outcome variable in the train set. In case of predicting the mortality of a patient, for example, the importance of dead patients could be increased to deal with the class imbalance [AAB+15]. Nevertheless, if another method is applied to deal with imbalanced classes and that uses the entire data set, the previously described problem of the conditional probability matrices exceeding the memory capacity, see Section 7, will reoccur.

10.3 Application of Graph Convolutional Transformer to Further Prediction Tasks

A main goal of representation learning is to find representations that are generic for several prediction tasks. To test whether GCT returns general-purpose visit representations, it should be applied to further prediction tasks in addition to mortality prediction. The underlying data set of PHD contains several potential binary outcome variables. Besides mortality prediction, GCT could be used to predict hospital-acquired infections, admission to ICU and pressure ulcers.

Furthermore, in their approach MiME, Choi et al. [CXSS18] conducted auxiliary prediction tasks to enhance MiME to find more generic representations. The prediction of conditional probabilities $P($ ICD code $\mid$ drug code $)$ was included in the representation mechanism

of MiME. This property allows to find representations that are not only useful for solving one specific prediction tasks but enables to find multi-functional representations. As well as for MiME, GCT could be extended with auxiliary prediction tasks.
Another possible extension could be the prediction of masked ICD codes, as proposed by [CXL+19]. The idea behind this procedure is masking codes in a graph and predicting them based on the neighboring nodes similar to skip-gram and find general-purpose representations. The application of GCT to the mentioned tasks remains future work.

10.4 Extension of the Input Data with Additional Features

In this work, only ICD codes and drug codes available on the day of hospital admission are taken into account. Choi et al. [CXL+19] make use of all codes reported during a visit. For two reasons, only the codes available on the first day of admission are used in this work. Firstly, the model becomes comparable to the approach of [FGLB18] that predicts outcome variables based on information available at admission. Secondly, the model will be more applicable in practice if it is performed with information available at admission. With this property, GCT can immediately estimate the progress of a patient's health status when the patient is admitted to hospital and enable fast decisions.
Nevertheless, it might be interesting to explore how GCT performs with the information of other available days. The code lists could easily be extended and shuffled, but this procedure would neglect the day of drug prescription or assignment of an ICD code. To consider code lists resulting from different days as time series, a recurrent structure such as an RNN has to be applied.
Besides, the extension of the code lists to a longer time span, more detailed information on each record could be given to the network. For instance, PHD does not only contain the drug codes but also details regarding the prescribed amount and quantity of a drug.

10.5 Extended Hyperparameter Tuning

As already mentioned, instead of fully exploring the search space of hyperparameters, a few combinations are picked and tested in this work. This procedure cannot be expected to lead to an optimal choice of hyperparameters. Large computation capacity and more advanced techniques should be applied to investigate a hyperparameter combination that results in a superior prediction performance. Potential procedures to optimize hyperparameters are grid search, random search [BB12] or Bayesian optimization [PGCP+99].

10.6 Application of Hybrid Optimizer SWitches From Adam To SGD

As already described in Section 9.4.3, it can be observed that gradient descent leads to a superior test performance compared to Adam [KB14]. Keskar et al. [KS17] state the existence of a performance gap between Adam and gradient descent for some applications. While Adam often shows huge improvement in the beginning of training, it sometimes fails to generalize to test data when training proceeds. In contrast, gradient descent is often slow in the beginning but leads to superior performance on unseen test data. Keskar et al. developed an optimizer called Hybrid Optimizer SWitches From Adam To SGD (SWATS) [KS17] to combine the best of both worlds and solve the trade-off between training dynamics and generalization capability. SWATS starts the training with Adam but automatically switches to gradient descent as soon as generalization capability is compromised. Both the switching point and the learning rate for gradient descent after the switch are determined within the training process. It would be interesting to apply SWATS as an optimizer to GCT in this work.

SWATS is not yet implemented in TensorFlow [AAB$^+$15] library. Thus, the implementation of this idea is another possible future work [KS17].

10.7 Extension from Single-Head Attention to Multi-Head Attention

So far, single-head attention was used in this approach, as well as in [CXL$^+$19]. Though, in the original Transformer approach [VSP$^+$17] multi-head attention as an extension of single-head attention is proposed. Multi-head attention executes the attention mechanism multiple times independently and in parallel which allows the model to develop different attention matrices. The output of the heads are combined, usually by concatenation. Expanding GCT from single-head attention to multi-head attention could lead to faster convergence and better results [VSP$^+$17].

10.8 Application of Graph Convolutional Transformer to Multiple Divisions of the Data Set

In this work, the input data set is randomly divided in three subsets: the train set, the validation set and the test set. Then, the models are fitted on the train set and evaluated on the test set. The obtained AUC-ROC of the models are compared. Since the data set is split randomly, it can be assumed that the returned performance metrics are as well subject to randomness. The metrics can be seen as estimators of the true performance that a classifier achieves for the whole data set. To obtain more stable estimators of the performance metrics for each classifier, the models should be fitted and evaluated on different versions of the divisions in train, validation and test set. Furthermore, methods such as repeated holdout or k-fold crossvalidation [Bro00] estimate the performance on multiple versions, or so called folds of the division of the data set in train, validation and test set. Due to performance reasons, GCT was only evaluated on a single division into train, validation and test set in this work. In future, GCT should be trained, evaluated and tested on multiple divisions of the original data set.

Part IV

Conclusion

Initially, the state of the art related to the work of this thesis was illustrated to provide the reader with background information. This included an introduction to GPU computing and the applied deep learning libraries. Besides, the theory behind neural networks in general and in specific graph-related neural networks was presented. Then, relevant deep learning approaches in the domain of healthcare were described.
On the base of this introduction to the state of the art, the following part of the thesis displayed the contributions of this work. The details of the method and the setting of the experiment were described. Furthermore, the results were presented and interpreted. Finally, suggestions on the extension of the method and future experiments were drawn.

The main task of this thesis was to conduct mortality prediction on a large data set from PHD based on ICD and drug codes known on the first day of hospital admission. The particular objective was to leverage knowledge from inherent causal graph structure of EHR with the intention to increase predictive performance. The present data set does not explicitly represent the graph structure. Nevertheless, it was assumed that a hidden graph structure exists. GCT was chosen as a method which attempts to learn the graph structure by using the attention mechanism proposed in the Transformer approach. Guided by the learned structure, GCT then aims to derive meaningful representations of the data based on graph convolution that, at best, enhance prediction quality.

Initially, the data set was filtered and arranged to meet the requirements of the task. In particular, the data set was undersampled to obtain a balanced distribution of the outcome classes. Besides GCT, Transformer and logistic regression were trained or fitted on the same train set and evaluated on the same test set to draw comparisons between the models. Evaluated on the test set, GCT leads to an AUC-ROC of 78.8%, Transformer of 78.2% and logistic regression of 80.0%.

These observations were interpreted as follows. Logistic regression outperformed the graph-focused methods. Consequently, a major information gain resulting from the graph structure on mortality prediction is not evident in this work. The similar performance of Transformer and GCT indicate that the EHR-specific regularization of GCT did not guide the model to more conclusive representations. Nevertheless, the implementation of the experiments as well as the properties of GCT still leave room for several improvements. Firstly, GCT was solely applied to mortality prediction. Further prediction tasks should be conducted to assess the general usefulness of GCT in healthcare. Secondly, the chosen hyperparameters for GCT were not determined by an exhaustive hyperparameter search. Hyperparameter tuning is required to obtain an appropriate hyperparameter combination for GCT that applies to the prediction task in this work. Thirdly, even though neural networks are known for profiting from large amounts of data, the underlying data set was drastically reduced due to undersampling and only took advantage of ICD and drug codes available at admission time. Additional methods should be implemented to handle the class imbalance while utilizing more input data. Besides, elevating the number of samples, the information available per sample should be increased.

In conclusion, the experiments in this work were an initial application of GCT to a large data set from PHD and even though, GCT lead to a state-of-the-art prediction performance not far from logistic regression. Although, GCT did not outperform the logistic regression baseline in this approach on a mortality prediction task, the results are encouraging with regard to the fact that GCT can be extended in many directions. It can be concluded that GCT still has a huge potential to serve as an appropriate prediction tool on EHR data in future applications.

References

[AAB$^+$15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. `http://tensorflow.org/`, 2015. Software available from tensorflow.org.

[AJB99] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Diameter of the world-wide web. *Nature*, 401(6749):130–131, Sep 1999.

[Ala18] Jay Alammar. The illustrated transformer [blog post on jalammar]. `http://jalammar.github.io/illustrated-transformer/`, 2018.

[AMHK$^+$17] Julia Adler-Milstein, A Jay Holmgren, Peter Kralovec, Chantal Worzala, Talisha Searcy, and Vaishali Patel. Electronic health record adoption in US hospitals: the emergence of a digital "advanced use" divide. *Journal of the American Medical Informatics Association*, 24(6):1142–1148, 08 2017.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(Feb):281–305, 2012.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[Ben12] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures, 2012.

[Ben15] Yoshua Bengio. Rmsprop and equilibrated adaptive learning rates for nonconvex optimization. *corr abs/1502.04390*, 2015.

[BK18] Andrew L. Beam and Isaac S. Kohane. Big Data and Machine Learning in Health Care. *JAMA*, 319(13):1317–1318, 04 2018.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[BPM04] Gustavo E. A. P. A. Batista, Ronaldo C. Prati, and Maria Carolina Monard. A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explor. Newsl.*, 6(1):20–29, June 2004.

[Bro00] Michael W Browne. Cross-validation methods. *Journal of mathematical psychology*, 44(1):108–132, 2000.

[Bro19a] Jason Brownlee. 8 tactics to combat imbalanced classes in your machine learning dataset. `https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/`, September 2019.

[Bro19b] Jason Brownlee. A gentle introduction to pooling layers for convolutional neural networks. `https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/`, July 2019.

[Bro19c] Jason Brownlee. How do convolutional layers work in deep learning neural networks? https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/, September 2019.

[BTR15] Paula Branco, Luis Torgo, and Rita Ribeiro. A survey of predictive modelling under imbalanced distributions, 2015.

[C$^+$15] François Chollet et al. Keras. https://keras.io, 2015.

[CBK$^+$16] Edward Choi, Mohammad Taha Bahadori, Joshua A. Kulas, Andy Schuetz, Walter F. Stewart, and Jimeng Sun. Retain: An interpretable predictive model for healthcare using reverse time attention mechanism. https://arxiv.org/abs/1608.05745, 2016.

[CBS$^+$15] Edward Choi, Mohammad Taha Bahadori, Andy Schuetz, Walter F. Stewart, and Jimeng Sun. Doctor ai: Predicting clinical events via recurrent neural networks. https://arxiv.org/abs/1511.05942, 2015.

[CBS$^+$16a] Edward Choi, Mohammad Taha Bahadori, Elizabeth Searles, Catherine Coffey, and Jimeng Sun. Multi-layer representation learning for medical concepts. https://arxiv.org/abs/1602.05568, 2016.

[CBS$^+$16b] Edward Choi, Mohammad Taha Bahadori, Elizabeth Searles, Catherine Coffey, Michael Thompson, James Bost, Javier Tejedor-Sojo, and Jimeng Sun. Multi-layer representation learning for medical concepts. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1495–1504, New York, NY, USA, 2016. ACM.

[CBS$^+$16c] Edward Choi, Mohammad Taha Bahadori, Le Song, Walter F. Stewart, and Jimeng Sun. Gram: Graph-based attention model for healthcare representation learning. https://arxiv.org/abs/1611.07012, 2016.

[CBS19] Ting Chen, Song Bian, and Yizhou Sun. Are powerful graph neural nets necessary? a dissection on graph classification, 2019.

[CCS16] Youngduck Choi, Chill Yi-I Chiu, and David Sontag. Learning low-dimensional representations of medical concepts. In *CRI*, 2016.

[Cha10] Nitesh V. Chawla. *Data Mining for Imbalanced Datasets: An Overview*, pages 875–886. Springer US, Boston, MA, 2010.

[CSSS16a] Edward Choi, Andy Schuetz, Walter F. Stewart, and Jimeng Sun. Medical concept representation learning from electronic health records and its application on heart failure prediction. https://arxiv.org/abs/1602.03686, 2016.

[CSSS16b] Edward Choi, Andy Schuetz, Walter F Stewart, and Jimeng Sun. Using recurrent neural network models for early detection of heart failure onset. *Journal of the American Medical Informatics Association*, 24(2):361–370, 08 2016.

[CWV$^+$14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014.

[CXL$^+$19] Edward Choi, Zhen Xu, Yujia Li, Michael W Dusenberry, Gerardo Flores, Yuan Xue, and Andrew M Dai. Graph convolutional transformer: Learning the graphical structure of electronic health records. *arXiv preprint arXiv:1906.04716*, 2019.

[CXSS18] Edward Choi, Cao Xiao, Walter F. Stewart, and Jimeng Sun. Mime: Multilevel medical embedding of electronic health records for predictive healthcare. https://arxiv.org/abs/1810.09593v1, October 2018.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[dee20] Deep learning: Algorithms and applications. https://doi.org/10.1007/978-3-030-31760-7, 2020.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[DG10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12(null):2121–2159, July 2011.

[Dra19] Rachel Lea Ballantyne Draelos. The transformer: Attention is all you need [blog post on glass box]. https://glassboxmedicine.com/2019/08/15/the-transformer-attention-is-all-you-need/, August 2019.

[DSR+18] Spiros Denaxas, Pontus Stenetorp, Sebastian Riedel, Maria Pikoula, Richard Dobson, and Harry Hemingway. Application of clinical concept embeddings for heart failure prediction in uk ehr data. https://arxiv.org/abs/1811.11005, 2018.

[ESB+16] Cristóbal Esteban, Oliver Staeck, Stephan Baier, Yinchong Yang, and Volker Tresp. Predicting clinical events by combining static and dynamic information using recurrent neural networks. *2016 IEEE International Conference on Healthcare Informatics (ICHI)*, pages 93–101, 2016.

[FGLB18] Amela Fejza, Pierre Genevès, Nabil Layaïda, and Jean-Luc Bosson. Scalable and Interpretable Predictive Models for Electronic Health Records. In *DSAA 2018 - 5th IEEE International Conference on Data Science and Advanced Analytics*, pages 1–10, Turin, Italy, October 2018. IEEE.

[FML15] Joseph Futoma, Jonathan Morris, and Joseph Lucas. A comparison of models for predicting early hospital readmissions. *Journal of Biomedical Informatics*, 56:229 – 238, 2015.

[FWH+16] Wael Farhan, Zhimu Wang, Yingxiang Huang, Shuang Wang, Fei Wang, and Xiaoqian Jiang. A predictive model for medical events based on contextual embedding of temporal sequences. In *JMIR medical informatics*, 2016.

[Gam18] Thomas Gamauf. Tensorflow records? what they are and how to use them. https://medium.com/mostly-ai/tensorflow-records-what-they-are-and-how-to-use-them-c46bc4bbb564, 2018.

[GBC] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press.

[GCL+18] Pierre Genevès, Thomas Calmant, Nabil Layaïda, Marion Lepelley, Svetlana Artemova, and Jean-Luc Bosson. Scalable machine learning for predicting at-risk profiles upon hospital admission. *Big Data Research*, 12:23–34, 2018. https://hal.inria.fr/hal-01517087/document.

[GNS+18] Marzyeh Ghassemi, Tristan Naumann, Peter Schulam, Andrew L. Beam, Irene Y. Chen, and Rajesh Ranganath. A review of challenges and opportunities in machine learning for health, 2018.

[GPB+04] Johnson George, Yee-Teng Phun, Michael J Bailey, David CM Kong, and Kay Stewart. Development and validation of the medication regimen complexity index. *Annals of Pharmacotherapy*, 38(9):1369–1376, 2004. PMID: 15266038.

[GSM+17] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Elliot Karro, and D. Sculley, editors. *Google Vizier: A Service for Black-Box Optimization*, 2017.

[Gu18] Mandy Gu. Neural networks for word embeddings: Introduction to natural language processing part 3. `https://medium.com/analytics-vidhya/neural-networks-for-word-embeddings-4b49e0e9c955`, 2018.

[Gé19] Aurélien Géron. Hands-on machine learning with scikit-learn, keras, and tensorflow : concepts, tools, and techniques to build intelligent systems, September 2019.

[Has17] Trevor Hastie. *The elements of statistical learning : data mining, inference, and prediction.* Springer series in statistics. Springer, New York, NY, second edition, corrected at 12th printing 2017 edition, [2017].

[HG09] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, Sep. 2009.

[Hol19] Tony Holdroyd. Tensorflow 2.0 quick start guide. get up to speed with the newly introduced features of tensorflow 2.0, March 2019.

[HT13] Karimollah Hajian-Tilaki. Receiver operating characteristic (roc) curve analysis for medical diagnostic test evaluation. *Caspian journal of internal medicine*, 4:627–635, 09 2013.

[HYV] J. Henry, Pylypchuk Y., and Searcy T. Patel V. Adoption of electronic health record systems among u.s. non-federal acute care hospitals: 2008-2015.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Premier healthcare databse: Data that informs and performs. `https://www.premierinc.com`, 2015.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[jac17] *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.* OpenReview.net, 2017.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[Koe18] Will Koehrsen. Neural network embeddings explained. how deep learning can represent war and peace as a vector. *Towards Data Science*, 2018.

[KS17] Nitish Shirish Keskar and Richard Socher. Improving generalization performance by switching from adam to sgd, 2017.

[KSS+20] Anand J. Kulkarni, Patrick Siarry, Pramod Kumar Singh, Ajith Abraham, Mengjie Zhang, Albert Zomaya, and Fazle Baki. Big data analytics in healthcare, 2020.

[LF19] Agnes Lydia and F Sagayaraj Francis. Adagrad-an optimizer for stochastic gradient descent, 2019.

[LKEW15] Zachary C. Lipton, David C. Kale, Charles Elkan, and Randall Wetzel. Learning to diagnose with lstm recurrent neural networks, 2015.

[LLH+19] Luchen Liu, Haoran Li, Zhiting Hu, Haoran Shi, Zichang Wang, Jian Tang, and Ming Zhang. Learning hierarchical representations of electronic health records for clinical outcome prediction, 2019.

[LRK+18] John Boaz Lee, Ryan A. Rossi, Sungchul Kim, Nesreen K. Ahmed, and Eunyee Koh. Attention models in graphs: A survey. https://arxiv.org/abs/1807.07984, 2018.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. https://arxiv.org/abs/1301.3781, 2013.

[MCZ+17] Fenglong Ma, Radha Chitta, Jing Zhou, Quanzeng You, Tong Sun, and Jing Gao. Dipole. *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*, 2017.

[MHGK14] Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. Recurrent models of visual attention. In *NIPS*, pages 2204–2212, 2014.

[mom99] On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145 – 151, 1999.

[Moo19] Jojo Moolayil. *Learn Keras for Deep Neural Networks : A Fast-Track Approach to Modern Deep Learning with Python.* SpringerLink. Apress, Berkeley, CA, 2019.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[NTWV16] Phuoc Nguyen, Truyen Tran, Nilmini Wickrasinghe, and Svetha Venkatesh. Deepr: A convolutional net for medical records. *IEEE Journal of Biomedical and Health Informatics*, PP, 07 2016.

[Ole17] Galina Olejnik. Word embeddings: exploration, explanation, and exploitation. https://towardsdatascience.com/word-embeddings-exploration-explanation-and-exploitation-with-code-in-python-5dac99d5d795, December 2017.

[PGCP+99] Martin Pelikan, David E Goldberg, Erick Cantú-Paz, et al. Boa: The bayesian optimization algorithm. In *Proceedings of the genetic and evolutionary computation conference GECCO-99*, volume 1, pages 525–532, 1999.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[Pre96] Lutz Prechelt. Early stopping-but when. *Advances in neural information processing systems*, 2:55–69, 1996.

[Pri10] José C. Principe. Information theoretic learning : Renyis entropy and kernel perspectives, 2010.

[PTPV16] Trang Pham, Truyen Tran, Dinh Phung, and Svetha Venkatesh. Deepcare: A deep dynamic memory model for predictive medicine. https://arxiv.org/abs/1602.00357, 2016.

[PTPV17] Trang Pham, Truyen Tran, Dinh Phung, and Svetha Venkatesh. Predicting healthcare trajectories from medical records: A deep learning approach. *Journal of Biomedical Informatics*, 69, 04 2017.

[PVG$^+$11] 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*, 12:2825–2830, 2011.

[Reb19] Gopinath Rebala. *An Introduction to Machine Learning*. Springer eBooks. Springer, Cham, 2019.

[Roc19] Baptiste Rocca. Handling imbalanced datasets in machine learning. *Towards Data Science*, 2019.

[Ron14] Xin Rong. word2vec parameter learning explained. https://arxiv.org/abs/1411.2738, 2014.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016.

[Rui17] Rutger Ruizendaal. Deep learning 4: Why you need to start using embedding layers. and how there's more to it than word embeddings. https://towardsdatascience.com/deep-learning-4-embedding-layers-f9a02d55ac12, 2017.

[RWD$^+$17] D. Ravì, C. Wong, F. Deligianni, M. Berthelot, J. Andreu-Perez, B. Lo, and G. Yang. Deep learning for health informatics. *IEEE Journal of Biomedical and Health Informatics*, 21(1):4–21, Jan 2017.

[Sei18] George Seif. Handling imbalanced datasets in deep learning. *Towards Data Science*, November 2018.

[SHK$^+$14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[SKP97] Daniel Svozil, Vladimir Kvasnicka, and Jiří Pospíchal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems*, 39:43–62, 11 1997.

[SLM$^+$15] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. https://arxiv.org/abs/1502.05477, 2015.

[Smi18] Leslie N. Smith. A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay, 2018.

[SR15] Takaya Saito and Marc Rehmsmeier. The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets. *PLOS ONE*, 10(3):1–21, 03 2015.

[STBR18] Benjamin Shickel, Patrick James Tighe, Azra Bihorac, and Parisa Rashidi. Deep ehr: A survey of recent advances in deep learning techniques for electronic health record (ehr) analysis. *IEEE Journal of Biomedical and Health Informatics*, 22(5):1589–1604, Sep 2018.

[Tau19] Tom Taulli. *Artificial intelligence basics : a non-technical introduction*. Springer eBooks. Apress, New York, 2019.

[18] Premier Applied Sciences ®. Deep residual learning for image recognition. https://arxiv.org/abs/1512.03385, July 2018.

[The16] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[Thi18] Aditya Thiruvengadam. Transformer architecture: Attention is all you need [blog post on medium]. https://medium.com/@adityathiruvengadam/transformer-architecture-attention-is-all-you-need-aeccd9f50d09, 2018.

[TJPB18] Jesse D. Raffa Leo A. Celi Roger G. Mark Tom J. Pollard, Alistair E. W. Johnson and Omar Badawi. The eicu collaborative research database, a freely available multi-center database for critical care research. *Scientific Data*, September 2018.

[TK01] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition and Neural Networks*, pages 169–195. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[TNPV15] Truyen Tran, Tu Dinh Nguyen, Dinh Phung, and Svetha Venkatesh. Learning vector representation of medical objects via emr-driven nonnegative restricted boltzmann machines (enrbm). *J. of Biomedical Informatics*, 54(C):96–105, April 2015.

[Tut11] Gerhard Tutz. *Regression for Categorical Data*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2011.

[VCC$^+$17] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. https://arxiv.org/abs/1710.10903, 2017.

[VSP$^+$17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. https://arxiv.org/abs/1706.03762, 2017.

[Wan20] M. Arif Wani. *Advances in Deep Learning*. Studies in Big Data ; 57Springer eBooks. Springer, Singapore, 2020.

[WGGH17] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. Non-local neural networks. https://arxiv.org/abs/1711.07971, 2017.

[wor] Learn neural networks. word embedding by keras. https://learn-neural-networks.com/world-embedding-by-keras/.

[WPC+19] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. https://arxiv.org/abs/1901.00596, 2019.

[WRS+17] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning, 2017.

[XHLJ18] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? https://arxiv.org/abs/1810.00826, 2018.

[XMS+17] Cassandra Xia, Clemens Mewald, D. Sculley, David Soergel, George Roumpos, Heng-Tze Cheng, Illia Polosukhin, Jamie Alexander Smith, Jianwei Xie, Lichan Hong, Martin Wicke, Mustafa Ispir, Philip Daniel Tucker, Yuan Tang, and Zakaria Haque. Tensorflow estimators: Managing simplicity vs. flexibility in high-level machine learning frameworks. In *Proceedings of the 23th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Halifax, Canada, 2017.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[ZCZ+18] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. https://arxiv.org/abs/1812.08434, 2018.

[Zei12] Matthew D. Zeiler. Adadelta: An adaptive learning rate method, 2012.

Part V

Appendix

In the following, the results of GCT obtained with 10,000 training steps are listed. GCT is trained on the training set and fitted to the test set, as described in Section 7. Only one parameter is varied, while all other parameters remain constant, as follows:

- Mini-batch size:32
- Dropout rate: 0.08
- Embedding size: 128
- Initial learning rate: 0.00022
- Number of Transformer stacks: 3
- Regularization coefficient λ: 0.1
- Number of steps: 10000
- Optimizer: Adam

Mini-batch Size	AUC-PR on Test Set	AUC-ROC on Test Set	Loss
1	75.8 %	76.3 %	1.9974
15	75.3 %	76.7 %	2.0622
32	75.0 %	77.0 %	2.9440
64	71.8 %	74.8 %	4.5445
300	Exceeded memory capacity		

Table 7: Results of GCT with different mini-batch sizes, while all other parameters remain constant.

Dropout Rate	AUC-PR on Test Set	AUC-ROC on Test Set	Loss
0	72.5 %	72.3 %	3.3056
0.007	72.3 %	74.9 %	2.6331
0.05	73.0 %	76.2 %	3.1948
0.08	75.1 %	77.0 %	2.9441
0.1	73.9 %	76.0 %	3.0789
0.9	51.3 %	50.0 %	3.8063

Table 8: Result of GCT with different dropout rates, while all other parameters remain constant.

Embedding Size	AUC-PR on Test Set	AUC-ROC on Test Set	Loss
32	74.8 %	76.7 %	2.6990
64	74.7 %	76.5 %	3.0861
128	75.1 %	77.0 %	2.9441

Table 9: Results of GCT with different embedding sizes, while all other parameters remain constant.

Initial Learning Rate	AUC-PR on Test Set	AUC-ROC on Test Set	Loss
0.00001	66.4 %	66.4 %	2.1162
0.00022	75.1 %	77.0 %	2.9440
0.0001	74.4 %	76.4 %	2.4177
0.001	73.8 %	76.5 %	2.1776
0.01	51.3 %	50.0 %	2.1974
1	51.3 %	50.0 %	532.6621

Table 10: Results of GCT with different learning rates, while all other parameters remain constant.

Number of Transformer stacks	AUC-PR on test set	AUC-ROC on test set	Loss
2	72.2 %	72.7 %	4.2315
3	75.1 %	77.0 %	2.9440
4	74.4 %	76.4 %	2.4177
5	74.2 %	76.2 %	3.1200
6	71.2 %	74.6 %	3.2450
12	73.3 %	75.9 %	2.4107

Table 11: Results of GCT with different numbers of Transformer stacks, while all other parameters remain constant.

Regularization Coefficient λ	AUC-PR on Test Set	AUC-ROC on Test Set	Loss
0.01	74.3 %	76.7 %	2.5226
0.1	75.1 %	77.0 %	2.9440
1	75.3 %	76.7 %	4.1810
10	75.4 %	76.6 %	12.7283

Table 12: Results of GCT with different regularization coefficients λ, while all other parameters remain constant.