Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Using adaptive knowledge graphs in neural dialogue generation

Master's Thesis

Author
Patrik Purgai

Advisor
Ádám Kovács

December 20, 2020

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Purgai Patrik*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2020. december 20.

Purgai Patrik
hallgató

Kivonat

Az elmúlt években a mély tanuláson alapuló neurális hálózatokkal végzett természetes nyelvfeldolgozó algoritmusok már emberi szintű eredményeket érnek el a szövegértelmezésben és -generálásban. Ezeknek a módszereknek lényege, hogy az említett nyelvmodelleket rengeteg erőforrással, nagy mennyiségű szöveges adaton előtanítják, majd az itt felhalmozott nyelvi ismeretet finomhangolás során kisebb adathalmazokon speciális felhasználáshoz igazítják. Ilyen lehet például dialógus generálás, melyben látványos javulást okoz olyan kontextuális ismeret megléte, mely nem található meg a kisebb dialógus adatbázisokban, viszont előtanítás során a modell memorizálta azt. Ez többnyire változatossá és kontextust megörzővé teszi a párbeszédet, viszont továbbra is sok tényszerű inkonzisztenciára lehetünk figyelmesek. A dolgozatom célja a generált válaszok minőségének további javítása egy külső tudásgráf beépítésével a Transformer architektúrába. A kibővített rendszerrel együtt számos további megközelítést valósítok meg, amelyeket aztán különféle automatikus mérőszámokkal és empirikus értékeléssel hasonlítok össze.

Abstract

In recent years, natural language processing algorithms with deep learning-based neural networks can already achieve human-level results on text interpretation and generation. One of the most significant reasons for this is the use of pre-trained Transformer-based language models. The essence of these methods is that the mentioned neural networks are initially trained on a large amount of textual data, and then the linguistic knowledge accumulated here is adapted for special use on smaller data sets during fine-tuning. Such can be, for example, dialogue generation, in which the existence of contextual knowledge that is not found in the smaller dialogue databases, but which was memorized by the model during pre-training, causes a spectacular improvement. This mostly makes the dialogue varied and context-preserving, but we can still be aware of many factual inconsistencies. The goal of this thesis is to further improve the quality of generated responses by incorporating an external memory source into the Transformer architecture, in the form of a knowledge graph. In conjunction with the extended dialogue system, I implement several baseline approaches, which I then compare through various automatic metrics and empirical evaluation.

Chapter 1

Introduction

Building dialogue models with the capability of simulating the features of human conversation is a challenging and heavily studied area of research. Modern natural language understanding algorithms provide great performance for customer service-oriented specialized chatbots, however these solutions only perform well as long as the conversation does not move too far away from the robot's remit. To create conversational agents that are capable of handling the complexity of open-domain human language, current models leverage the power of deep neural networks. Prior work has shown that well-performing dialogue systems can be achieved for this use-case by having sufficiently large trainable parameter count and extensive pre-training on hundreds of gigabytes of text corpora. Although such end-to-end neural network architectures are the basis for state of the art solutions in the field, the quality and feel of machine-generated dialogue is still easily distinguishable from real human-to-human conversations. Mimicking human-like responses may keep up the illusion of a real chat partner in shorter exchanges, but dragging out conversations to more depth usually brings out discrepancies. There are several factors in human dialogue that is taken as granted by people, the lack of even one of which can be disrupting for the interlocutor. Usually these are the small, but essential elements, that are missed by neural network generated outputs, like forgetting mentioned named entities, rapid context changes or repetition to name a few. Sometimes such faulty expressions can be understood as funny puns due to their absurdity or irrelevance, but their regular occurrence greatly contributes to the deterioration of user experience. The problem stems from the capacity of end-to-end machine learning solutions, which can theoretically be increased to a certain point by using additional parameters, but the computational resource required to use them can quickly exceed the manageable level. Let's suppose we use a hypothetical model that has enough parameters to recall and interpolate word occurrence patterns from huge training data to give an intellectually and grammatically perfect answer to any arbitrary utterance. This model is still unable to contribute any kind of factual information to the conversation, which would not have been included in the training corpus. The following example might provide better insight to this problem. Given that the current conversation context is about animals, a perfectly reasonable response from the model would be: *"Cats have three legs"*. Since the neural network might not have seen this exact utterance in the training data, it can't possibly know how many legs does a cat have, although it correctly assumes that cat is an animal and has legs.

It is difficult to address such issues in end-to-end models, because there is no way to expand the information available in the system other than modifying the training data, that would require fine-tuning of the existing parameters. Initiating training with the extended data is a slow and cumbersome operation due to the sheer size of the corpus. Training only on

the relevant subset could easily lead to other problems like catastrophic forgetting, which means the deletion of previously learned features from the model memory. Assembling dialogue systems from multiple neural network components that are trained on specialized tasks and applying external memory could be the solution to most of the mentioned problems. Maintaining modules that enforce empathetic and contextually correct responses for example can decrease the parameter size, as the training algorithm optimizes for explicit goals. While constructing datasets for such models requires human supervision, relying on pre-trained model architectures alleviates the need for large labeling work. Dynamically adding new or missing knowledge to the model and even deleting from it is possible via external memory sources. These databases are handled by the mentioned sub-models, that can extract useful data during training and inference. One technique to represent structured information is a knowledge graph. Knowledge graphs are a collection of interlinked entity nodes and their relations, where the entities are described with formal semantics, such that it is easily processable by either a computer or a human.

In this work I attempt to tackle the mentioned shortcomings of end-to-end dialogue systems by incorporating external memory in the form of a knowledge graph. In my experiments I establish several baseline architectures for dialogue generation and I construct a custom system that actively leverages its external data source in order to facilitate the generation of factually correct and coherent responses. The implementation is available in a Github repository [1] under the MIT license.

[1] https://github.com/bme-chatbots/dialogue-generation

Chapter 2

Related Work

The development and research of dialogue agents originates back to several decades with the works of Weizenbaum [1966] and Isbell et al. [2000] among numerous other studies. Their methodology usually involved limiting the scenarios to a specific topic and using a pre-defined set of templates as well as manually assembled rules in order to construct responses and engage in user interactions. In contrast to these works Ritter et al. [2011] focuses on a simpler, but potentially more scalable approach by leveraging large amounts of conversational training data to optimize a log-linear model for generating a single response to a given stimulus. Their formulation of training data structure with input and target utterance pairs is still the basis for today's neural network based solutions. Over the years, strong similarity is observable in the architectural choices between machine translation and conversation modeling. It follows that the large increase in translation performance obtained by neural networks also had a strong influence on the field of dialog generation. Among the first to successfully apply end-to-end neural architectures for dialogue modeling was Vinyals and Le [2015], who trained the novel sequence-to-sequence framework from Sutskever et al. [2014] as a conversational model. One of the most influential results in this area came from Bahdanau et al. [2015] that remedied most of the known issues of vanilla sequence-to-sequence architecture and elevated state of the art results beyond the capability of other mainstream statistical approaches. The first great industrial adaptation of this technology was published by researchers at Google in Wu et al. [2016], where the authors describe their new neural network-based backend for the popular Google Translate application. After the initial success of the architecture, however, demands towards language generation systems continued to grow and even better performing models were sought for. The next great stepping stone in the evolution of end-to-end models was the work by Vaswani et al. [2017], that introduced the novel Transformer framework. Compared to its predecessors, the general computational model of transformer-based neural networks have proven to be well suited for scaling their practical representational power through increasing their trainable parameter count. Aside from their theoretical advantages, the computational graph of transformers is easily parallelizable, which allows for efficient horizontal scalability with graphical- or tensor processing units. The Success of this new architecture is indicated by its rapid spread across top positions of almost every natural language understanding benchmarks. Due to the popularity of this new method, extensive research efforts lead to several variations of the original Transformer framework, most notably BERT from Devlin et al. [2018] and GPT-2 from Radford et al. [2019]. By employing these pre-trained models in downstream problems like dialogue modeling, the observed language generation quality has reached an unprecedented level. Another line of research is represented by approaches that separate large-scale world knowledge and

common sense from model parameters into external data sources. The motivation behind these techniques comes from the fact that conversations often revolve around knowledge. Earlier studies in this area have indeed shown that experiments with these settings yield promising results by enhancing the factual accuracy of responses. One of these works is from Ghazvininejad et al. [2017], where the authors' key idea is to use a generalized sequence to sequence framework with a generator model, that is not only conditioned on the dialogue history, but also relevant facts, which are queried from an open-domain knowledge graph. Another similar work is from Zhou et al. [2018], where the retrieved facts are transformed with a series of attention operations and then used for the conditional generation. In the following paragraphs, the paper will present detailed descriptions of the most important and impactful studies, that influenced this work.

Blender Bot from Roller et al. [2020] focuses on collecting recipes for building state of the art dialogue generator agents instead of contributing novel ideas to the field. They step forward from using a single end-to-end model for generation, and incorporate an additional ranker network into the system. The retrieved candidate responses from the ranker are then passed to the decoder as example sentences to enhance the quality of outputs. They also emphasize the importance of blending different tasks, which focus on enhancing personality, engagingness and knowledge through the Blended Skill Talk set-up from Smith et al. [2020].

CCM (Commonsense Conversational Model) from Zhou et al. [2018] uses a commonsense grounded sequence to sequence framework by incorporating a knowledge graph retriever and encoder model into the architecture. Given an input utterance, the model fetches the relevant knowledge graph from the data source, then uses a static graph attention mechanism to augment the semantic content of the input with encoded knowledge triplets. Existing models use KG triplets separately and independently, while the author's approach fetches a whole sub-graph from the database. According to them, this method lets the model encode more structured and connected semantic information from the graphs. They show that their model is capable of generating more informative and accurate responses, than other state-of-the-art methods. The main idea behind CMM is very similar to this thesis, but contrary to my approach, they recompute the knowledge triplets and their scores at each word generation time step.

DyKgChat from Tuan et al. [2019] uses a knowledge grounded dialogue generation framework with a sequence to sequence architecture. In addition to leveraging external data source to enhance factual accuracy, they propose a novel method with zero-shot adaptation to dynamic graphs. Instead of relying on attention mechanism over encoded knowledge graph triplets, their model employs the concept of multi-hop reasoning from Lao et al. [2011]. Due to the lack of existing conversational dataset with paired dynamic knowledge graphs, they assemble a new corpus from TV series dialogs, called DyKgChat. They collect scenes from the life of sitcom characters, to contain dialogues, speakers, locations and the known relationships between them as knowledge graphs.

KGLM from IV et al. [2019b] aims to ease the difficulty for language models to recall information from an external source. They use a mechanism called KGLM for selecting and copying facts from a knowledge graph that are relevant to the current context. With this method the model is able to use information, that it has never seen in training time, as well as to generate words, which are not in the models vocabulary. They also propose a novel dataset, called Linked WikiText-2, which is an annotated text with alignment to the WikiData knowledge graph.

GraftNet from Sun et al. [2018a] tackles the problem of question answering over the combination of a knowledge base and entity-linked text, where an incomplete knowledge graph

is available with a large text corpus. They propose the GRAFT-Net model for extracting and generating responses from the relevant parts of the knowledge base containing text and the mentioned KG entities with their relations.

ConceptFlow from Zhang et al. [2020] is a conversation generation model, which uses commonsense knowledge graphs to model the flow of conversations. They ground conversations on concept representations, such that, their model is trained to predict and traverse the concept space along commonsense relations. This is done by using graph attentions in their concept graph, moving towards relevant paths in the aforementioned representation space. They perform experiments on Reddit conversation datasets to show the performance of their technique. According to their results, this method outperforms end-to-end GPT-2 based techniques for dialogue modeling, while using marginally less parameters.

Chapter 3

Background

In this chapter the paper will introduce the basic building blocks of this thesis, by covering the theoretical background of the relevant neural network architectures and the main intuition behind knowledge graphs. Section 3.10 presents the Transformer framework in conjunction with preceding sections about machine learning algorithms. Section 3.11 discusses the idea behind knowledge graphs and their great potential in machine learning problems.

3.1 Natural Language Processing

When we talk about natural language processing (NLP), we mean the collection of algorithms, that are aimed at manipulating or understanding natural language. The term language in this context is used for communication between humans, like the English or German. While artificial programming languages such as Java or Python have formal syntax, natural languages are much more loose in this regard, therefore processing them with explicitly written rules and algorithms is not a trivial problem. The complexity can range from simply counting word frequencies up to understanding intent or emotions in utterances to the extent of generating responses to them.

Chatbots (i.e.: or also known as conversational agents, dialog systems) are special kind of NLP applications that try to mimic or simulate human-like conversation to provide a textual or verbal interface for the users. Nowadays the most used application of dialogue generation systems are providing customer service for government administration or business enterprises.

The purpose of chatbots may vary from responding to customer questions and providing information about specific products, to navigating users through complex websites. Since these agents can only act in their own narrow field of interests, they are not capable in any other domain of discourse. Consequently, this type of dialog agents are called task-specific chatbots and are created for the sake of providing an easily accessible and user friendly interface for making hotel bookings or restaurant reservations to name a few examples. Today most of these systems can be accessed through Facebook and other messaging platforms, or as a widget, which is integrated into an organization's website. Such conversational programs can be constructed as simple systems, which check for specific words in the input sequence, and then respond with an utterance, that is chosen by predefined hard-coded rules from a fixed template database. This approach yields the previously mentioned weakness of narrow range of domain, and does not perform well in

any other dialogue context. This rule-based approach faces great challenges in scalability and generalization as well, thus more sophisticated approaches are required for an agent, which would be able to converse in an open-domain dialogue setting.

The other main case for generating responses is when the utterances may come from a wider range of topics (or as previously mentioned, open-domain). One of the most traditional family of algorithms in natural language processing are language models. The main idea behind these methods is to construct a statistical model, that given the previous words in a sequence, predicts the upcoming word. By training these models on conversational data with various topics, they can be leveraged as the backbone of a dialogue agent, that generate responses word by word instead of choosing from pre-defined set of sentences. While in theory this approach seems like a great solution for the problem of scalability, in practice, ensuring the good quality of outputs by the model is definitely a difficult problem. Even though modern language models have been shown to be able to produce convincingly good phrases of coherent text, using them in an interactive dialogue environment implies several factors in which the model must be able to perform well.

3.2 Machine Learning

In classical programming methods, there are explicitly defined rules and instructions, which are written by humans for the computer to execute. The field of machine learning uses a different approach, as humans only define a family of possible rules for the computer, along with inputs and expected output pairs. It is then the machine's task to find the most optimal set of rules, which are best at mapping the provided inputs to their corresponding outputs. Notice the difference between the two cases, as in a conventional programming setting, great effort is put into the explicit instructions, whereas in machine learning, data engineering plays a much more prominent role.

Probably one of the most important components for clarifying how of machine learning algorithms work, is understanding the way information is represented to them. While classical programming methods have no problem processing any kind of structured data consisting of various types, machine learning-based solutions are constrained by their nature to only accept number-type input values. Although this stipulation might be a challenge in some cases, there are several practical techniques for converting data to the required format. In machine learning problems' raw input features can actually fall into two main classes, either numerical or categorical.

For the purpose of explaining the difference between them, consider the following weather forecasting model example. In this setting $temp_{i=1...5}$ denotes temperature values from the previous five hours and as the name suggests, $part\_of\_day$ marks the part of the day (e.g: morning, afternoon, night). These attributes form the input features of the model, and the task is to predict the temperature value for the next hour. Here, the five temperature input values are numerical features, since they are naturally represented as numbers and arithmetic operations can be interpreted on them. On the other hand, $part\_of\_day$ is considered as a categorical feature, because its values are inherently discrete concepts, which are represented in a textual format. A very common way of converting categorical features to numbers is by using dummy variables. This way, the given input attribute is divided into as many new features as the number of values that can be taken by the original feature. This is basically a mapping between the category values and dummy features, where the value of a dummy feature belonging to a given category is 1 or 0, depending on whether the data example contains that particular category value. The previously

described method is also called as one-hot encoding, which is a way of storing input feature values in sparse vectors. In the given example with three possible parts of day, the one-hot encoded values would look like the following: morning $\rightarrow [1, 0, 0]$, afternoon $\rightarrow [0, 1, 0]$, night $\rightarrow [0, 0, 1]$. Notice that for features with larger categorical value count, these vectors could become extremely sparse, which decreases memory efficiency. Fortunately, there is a trick to overcome this issue, which is storing only the index of the active dummy variable. In section 3.5 I go into more detail about this topic, as input representation is a very important aspect in natural language processing algorithms. However, in the upcoming sections I expand the picture of machine learning through the introduction of basic machine learning models and other important concepts.

3.2.1 Linear Regression

One of the simplest machine learning algorithm is linear regression. This approach relies on the assumption, that there is a linear dependency between the input and expected output values. Given that the input features are numericalized, this dependency can be expressed as a function in the following form:

$$f(x) = Wx + b \tag{3.1}$$

W and b are the degrees of freedom in the model, which refers to the values that can be taken by these parameters determine the possible assignments in the model. Since there is a corresponding output value for each input, the goal of optimization is to find these values. This can be interpreted as the model parameters define a space, in which an optimal point has to be found based on the chosen metric. This metric is usually called a loss function, which is typically the mean squared error (MSE) for linear regression. The main purpose of this function is to penalise the model for examples, where the output and model prediction are distant. By evaluating the model from equation 3.1 on each input (denoted by x) and output (denoted by y) pair with N cardinality, the MSE loss for a particular parameter state is given by equation 3.2.

$$loss(f, x, y) = \frac{1}{N} \sum_{i=0}^{N} (f(x_i) - y_i)^2 \tag{3.2}$$

This is an optimization problem, which in the case of simple linear regression can usually be solved analytically in closed from. If so, then that means the function defined by the loss and model parameters is quadratic. Obviously, this is not always the case, and there are times when it is still convex optimization, but no longer quadratic. For such problems iterative methods like gradient descent (described in section 3.2.2) are usually used. The main intuition behind these algorithms, is to start at a random point on the loss surface, and then walk towards the optimum point by making small adjustments to the model parameters. If the optimized function is convex, then it also guaranteed that the model converges to the optimum point, which it will eventually find.

3.2.2 Gradient Descent

Gradient descent or also called as steepest descent is an iterative first-order optimization algorithm, which is often used in machine learning for finding the minimum point of a function. The algorithm makes small adjustment on the optimized parameters in every iterations by analysing the gradient of the objective, which provides an efficient way of learning the optimal value set for the degrees of freedom. This feature can be interpreted as gradient descent knows how to modify the model parameters, such that they always move in the best direction towards the local minimum of the error function. The main idea behind this approach is shown in 1, where where $f(x)$ is the function to be minimized and α is the step size hyper-parameter.

Algorithm 1: Gradient Descent

for $k = 0, 1, 2, \ldots$ **do**
  $g_k \leftarrow \nabla f(x_k)$
  $x_{k+1} \leftarrow x_k - \alpha g_k$
end

3.2.3 Interpretability

Just as the data-oriented machine learning models described in section 3.2 do not allow for evaluation based on explicitly programmed rules, the question arises as to exactly what information and internal processes are used by the model came to a certain prediction. This is an important aspect for their application in industrial settings, where better insight is often required for fixing or explaining incorrect behaviours.

The linear regression model presented in section 3.2.1 allows for great interpretability, as the weight values for their corresponding input features directly indicate their contribution to the final product. Although this is a great advantage for the model, the nature of this overly simple calculus prevents its applicability in such problems, where the correlations in the data are deeper and more meaningful.

Based on linear regression and the models described in the upcoming sections, we could come to the conclusion, that methods, which possess strong predictive power usually do so at the expense of their interpretability. While this trade-off is generally true, there are several exceptions, for which examples I do not discuss in this work.

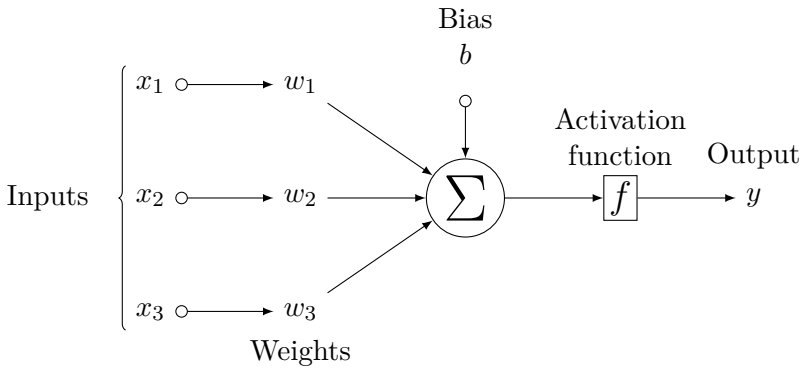
3.3 Deep Learning

Deep learning belongs to the family of machine learning methods, which consists of highly parameterized neural networks, that are capable of modeling complex relations in data. Although its wide spread use appeared only in the past decade, the theoretical background for them originates back to the 1960s. The reason for their relatively late adaptation comes from their high computational requirements, which was not present in that era.

A neural network is a biologically inspired model, that uses smaller computational nodes or neurons for representation learning. These smaller individual nodes are organized into a chain of transformations, to create a single nested composition of functions. Evaluating this chain of operations on a single data example is done by propagating the input forward through each composed transformation, which eventually results in the output

of the model. This process is commonly called as the forward pass in a neural network. While computing predictions with given parameter values seems straight forward, finding the optimal values poses a greater issue. In section 3.2.1, I presented two main types of optimization problems, however finding the best parameter state for neural networks usually belongs to a third category. In this case the error function is not convex, in other words there is no known algorithm, that is guaranteed to find the global optimum. The current state of the art methods for optimizing neural networks rely on variations of gradient descent or more generally first order techniques. Using the algorithm from 1, the first step is to compute the partial derivative of the error function for a given input-target pair with respect to the model parameters. Highly efficient execution of this process is done through the back-propagation algorithm, which is an optimized way of calculating partial derivatives by applying the chain rule to function compositions in a neural network setting. The second step of gradient descent is adjusting the model parameters with an update, that is proportional to the produced gradient-based error signal.

Figure 3.1: Example of an artificial neuron from Medina.



The strength of these methods comes from these stacked computations and efficient optimization techniques, that allow the model to automatically learn useful representations of the data. While this approach increases the modeling power of the algorithm, it also comes with the lack of explainability for the predictions. The applicability of deep learning solutions is therefore narrowed to cases, where interpretability can be sacrificed in exchange for better modeling and learning properties. In the following paragraphs of this section, the paper shares more detail about the components of a neural network in general.

Figure 3.1 shows a depiction of a single neuron, the main building block of a neural network. By looking at the visualization, the similarities are striking between the previously discussed linear regression from 3.2.1. The main difference is the inclusion of an activation function, which introduces non-linearity, that allows the model to learn more complex relations from the data. The role of this component is vital, as there may be cases where the model reaches much faster convergence with a given activation function. The question then naturally arises from this, is there a best activation function, and if so, which one? Since there are usually no clear answers to architectural and neural network topology questions of this nature, the choice of activation function is highly problem dependent. By the work of decades long neural network research, however, there are some empirically well-established choices. At the dawn of this field, sigmoid 3.3 and tangent hyperbolic 3.4 were the most popular activations. The reason for this might have been their resemblance to naturally occurring patterns and processes in the human brain, as well as their relatively good analytical properties.

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{3.3}$$

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3.4}$$

While these traditional activations (3.4, 3.3) can be utilized in smaller and shallower architectures, training deeper models leads to optimization issues with gradient-based techniques. One of the most prominent problems is the phenomena of vanishing gradient, which prevents certain model parameters from receiving updates. An example of its cause is the gradient range of the tangent hyperbolic function with $(0, 1)$, which for a model with n number of compositions, would decrease the gradient signal exponentially with n. This has the effect of considerably slowing down or even stopping the frontal parameters from training. A partial solution is the application of the rectified linear unit in 3.5, which yields fewer optimization problems. Another advantage is its simplicity and speed, as it only requires a comparison, addition and multiplication operation. Although ReLU manages to address saturation caused by the latter activations, it comes with the disadvantage of non-differentialability at 0 and the dying ReLU problem. It is another form of vanishing gradient, that happens as a result of certain neurons are stuck in the negative range, which stops the flow of gradient. At the expense of speed, this may be prevented by using the leaky variation of ReLU, that assigns a small positive number to values, which are smaller than 0.

$$relu(x) = max(0, x) \tag{3.5}$$

In addition to choosing the activation function, a lot of decisions have to be made when designing a neural network, most of which can often be critical for the proper learning of the model. In general, making these decisions requires an in-depth study of the problem, which in the case of neural networks can often be achieved through experimentation and other empirical methods. In the following sections the paper discusses some of the well proven architectures and techniques, that are relevant to this thesis.

3.4 Fully Connected Network

The fully connected neural network is one of the most traditional architectures in deep learning. A general overview can be seen in figure 3.2, which visualizes the interconnected neurons in the model. The depicted structure is composed into several layers, where the beginning of the model is an input layer, which is followed by several hidden layers until the final output layer. Each of these layers may have several hundreds or even thousands of artificial neurons, which propagate their received inputs from the previous layer by transforming and serving the result as input for the following layer of the architecture. The main idea behind of this layered sequential architecture, is to incline the model to learn different abstractions of the data. This approach yields a complex learning architecture, with usually millions of trainable parameters. The name *fully connected* originates from

the process of information passing between the layers, as the output of neurons at each layer is connected to every input node of the subsequent layer.

As a way of gaining intuition behind the computations performed by this model, the following paragraph briefly discusses the technical details of the operations in a fully-connected neural network. Let us consider a data sample with n number of features, which are organized into a single vector that is denoted by X. w marks the parameter values connecting the first layer to the subsequent, which are organized into a matrix W. There are n number of neurons for each input node, so the size of this matrix is $n \times m$, where m is the number of neurons in the next layer. It is also usually useful to include a bias element, which prevents the output space of the summation operation from skewing, by offsetting its value in a specific direction. Let b, o, f denote the bias, output and an arbitrary activation function respectively.

$$f\left(\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} \\ w_{21} & w_{22} & \dots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{bmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}\right) = \begin{pmatrix} o_1 \\ o_2 \\ \vdots \\ o_m \end{pmatrix} \tag{3.6}$$

Equation 3.6 shows the simplified version of the matrix transformations, that take place in a single layer of a neural network. By nesting these computations, we can actually arrive to the operation in equation 3.7, that describes a simple fully connected neural network.

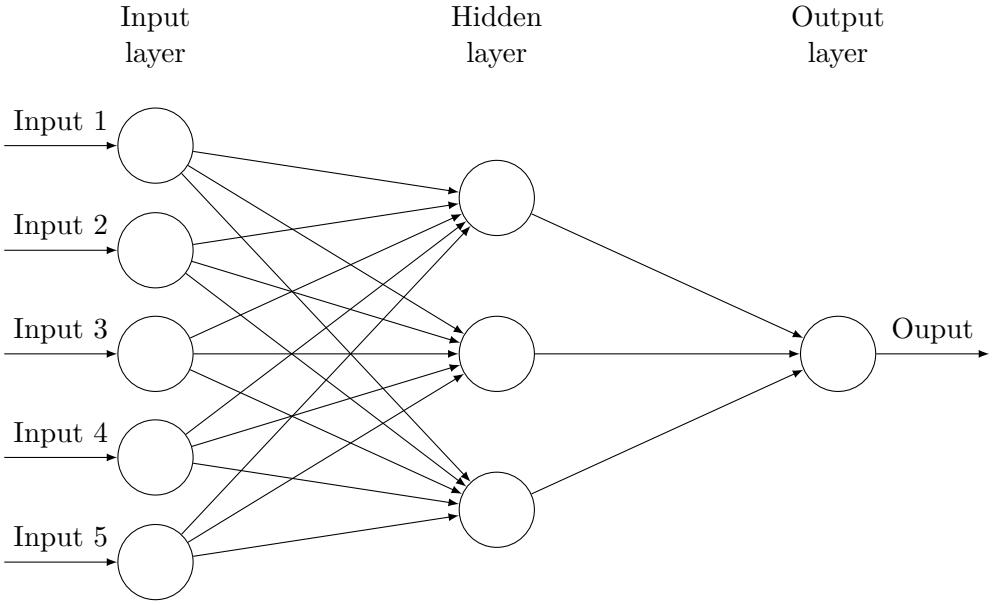
$$f(f(xW_1 + b_1)W_2 + b_2) = o \tag{3.7}$$

The reason behind aggregating the individual parameter values and input features into matrices and vectors, has rather not theoretical, but technical motivations. By handling closely related values in these dense data structures, the computations can leverage hardware acceleration via highly parallel computational units, like GPUs. By the advancements in silicon-based computer chip technology, high-performance machines are available at the tip of our fingers, which enabled not only the enterprise, but private usage of large neural networks.

Compared to the initially discussed linear regression model from 3.2, fully-connected neural network represents a considerably more competent system, which is capable of fitting highly complex datasets, that are out of reach for simpler algorithms. The predictive predictive power can be further increased through adding degrees of freedom to the neural network. There are generally two main characteristics, that define the number of parameters in a model, which is its depth and the neuron count of each layer. With only a single hidden layer, the illustrated model in figure 3.2 is considered a shallow architecture, while deep neural networks generally have much more. However, by attaching several such layers in succession, the activation function must be carefully chosen to avoid optimization related problems for the model. As shown in equation 3.7, the output of the architecture is computed from a composition of $h + 2$ transformations, where h denotes the number of hidden layers in the model.

Although having strong modeling capabilities is beneficial, adding too many parameters might result in bad overall model performance. In section 3.6 the paper discusses this topic in more detail, while presenting some techniques for mitigating this issue.

Figure 3.2: Example of a fully-connected neural network from Medina.



3.5 Word Embeddings

Before going forward in the presentation of more complicated neural network architectures and their integration with natural language processing systems, it is inevitable to come across the concept of embeddings.

There is a natural discrepancy between the way general machine learning-based algorithms accept inputs and the way it is available in a language processing environment. The problem of input representation is already discussed in section 3.2, where I introduce a technique for handling categorical data in machine learning methods. As shown, this is relevant for linear regression 3.2.1 and fully-connected networks 3.2, since they rely on number-type vectorized data, while natural language is generally in textual format. Therefore, it is necessary to convert between the two, otherwise these algorithms are not able to process the presented information. As a solution to this, words or other textual units can be interpreted as a categorical feature, therefore they can be associated with one-hot encoded vector representations. While there are numerous details concerning the exact implementation of this procedure, it actually is the basis for embedding techniques.

The first emerging problem resides in the size for such vectors. As there is a vast number of possible words even in a single language, it is infeasible with our current technology to handle vectorized data of this magnitude. This issue gets even worse, when the nature of NLP applications are taken into account. In most of the cases analysis and outputs are expected in real-time, therefore speed and memory efficiency are key concerns. To mitigate these problems, let us first consider Zipf's law in a language environment. According to this empirical observation, the frequency of a given word is inversely proportional to its rank in the ordered list of word frequencies. Therefore, it is plausible, that by using only a subset of words, we could create a vocabulary for the model, which includes a great majority of the relevant entries for the given task. The size of effective vocabulary can then be inferred from the actual training dataset or computed from relevant documents

in the specific domain. Those words, which are not part of this chosen range, can then be associated with a shared out_of_vocab encoding.

By limiting the possible words for the model, the issue of scalability is solved, but vector representations still store a wealth of redundant information due to their sparsity. Before discussing a solution for this problem, lets consider how a fully-connected neural network would process one-hot encoded vectors. The following equation depicts the computations in the input layer of the previously mentioned model. The inputs are provided as one-hot encoded vectors denoted by x_i^n, where i is the index of the active value and n is the vector size. Looking at the equation from 3.7, the forward pass for a single word would be computed as $x_i^n W^{n \times m}$ with m number of neurons in the input layer. The product of this matrix multiplication is a vector with m dimensions, which is actually the values from the i^{th} row of the W matrix. Notice that by using a simple shortcut of not manifesting the one-hot encoded vectors, but using the category value rank for slicing the i^{th} row of the matrix, we actually arrive at the same m dimensional vector. This way, the words are basically embedded through their ids in a latent space, which is actually a major technical concept behind embedding methods.

While memory efficiency-wise this is a great improvement, the strength of embeddings resides in the information that is stored in the individual vectors. By leveraging them as the part of a machine learning model, values of embeddings can be trained jointly with the parameters of the core algorithm, therefore inclining the vectors to learn useful representations of their associated words. In natural language processing there are various techniques, which can be used to obtain such word embeddings.

One of the most prominent work in this field is from Mikolov et al. [2013a], where the authors present Word2Vec, which is a family of algorithms for learning dense word representations from linguistic context. These are generally shallow neural networks with two layers, that take a large text database as input, and embed each word in the previously described vector space. The main goal of this model is to produce embeddings, such that words with similar common context have small distance between their corresponding vectors. Another interesting property of the learned representations by Word2Vec model, is the semantically meaningful interpretation of arithmetic operations between them. A commonly used example for demonstrating this phenomena is shown in equation 3.8, where the upper right arrow signals the embedding vector for the corresponding word.

$$\vec{King} - \vec{Man} + \vec{Woman} \approx \vec{Queen} \tag{3.8}$$

In the following paragraphs, the paper discusses two types of strategies for Word2Vec, which were introduced by Mikolov et al. [2013a].

Skip-gram is a special formulation of learning embeddings, where the training objective is to optimize word representations, such that they are useful at predicting the surrounding words in a given document. According to the formal definition provided by the authors, given sequence of $w_1, w_2, w_3, \ldots, w_N$ words, the goal is to maximize the average log probability. The exact equation can be seen in 3.9, where c is the context size and w_n is the middle word of the context.

$$\frac{1}{N} \sum_{n=1}^{N} \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{n+j} | w_n) \tag{3.9}$$

Using the basic Skip-gram training method, the value of $p(w_{n+1}|w_n)$ is calculated by a softmax function over the words in the vocabulary, that contains K number of entries.

$$p(w_O|w_I) = \frac{\exp\left(v'_{w_O} v^T_{w_I}\right)}{\sum_{k=1}^{K} \exp\left(v'_k v^T_{w_I}\right)} \tag{3.10}$$

3.10 shows this equation, where v'_{w_o}, v'_{w_i} and v'_k are the input, output and k^{th} word vectors from the embedding. As the authors suggest, this would be computationally expensive operation, as the value of K is usually large. An alternative to this approach is to use negative sampling technique, which replaces each computation of the $\delta \log p(w_O|w_I)$ term with Negative Contrastive Estimation. The idea behind NCE is that the model is trained to differentiate data from noise. By applying it to Skip-gram, the objective of the model is to distinguish invalid candidate words from the real target word. Negative sampling is an important method during my experiments as well, since several components of my proposed architecture leverage it as training objective.

Continuous Bag of Words is the other proposed approach for Word2Vec, which contrary to Skip-gram, learns to predict a single target word based on a provided input context. The effect of this change can be interpreted as the single learned representation is smoothed out between the most frequently occurring examples, which might result in worse overall representations for rare words. According to Mikolov et al. [2013a], Skip-gram is several times slower to train, but provides better accuracy for smaller training data and less frequent words.

3.6 Reguralization

During the optimization of neural networks and machine learning algorithms in general, a great care must be dealt with to the problem of overfitting. Overfitting occurs when the given machine learning algorithm is too closely fit to a particular set of data and therefore unable to generalize to previously unseen examples. The root cause of this problem is from having more degrees of freedom than it is justified for a given dataset, which results in a final parameter state, that basically remembers each data point instead of learning its underlying model. The counterpart of this issue is underfitting, when the model is unable to learn the latent data distribution, due to its low parameter count.

To mitigate these problems, a general machine learning approach is to maintain a separate training and validation set from the data. This way, the model is fit to only a slice from the original examples, while the other split is only used for evaluation during training, to check for potential overfitting.

In addition to data splitting and lowering parameter count, there are several tools to prevent models from memorizing data. One such family of methods, which aim at constraining the model to reduce its modeling power is called reguralization. Widely used reguralization approaches are L1 and L2 penalty, where there is an extrinsic term added to the optimized error function, that imposes a constraint for parameter values against moving out of a certain bound, thereby making the representation space overly complex. Another method is called Dropout from Srivastava et al. [2014]. The main idea behind this technique is to prevent neurons in the neural network from co-adaptation by randomly removing them at each iteration during the training phase. This way, the model is essentially split into exponentially many different sub-versions of itself, thereby creating

an ensemble of thinned neural networks. During test time the predictions of these smaller models are approximated by simply using a single model with averaged parameter values.

3.7 Recurrent neural network

In the previous sections the paper introduces the main idea behind simple neural network architectures and the capabilities of fitting to complex dependencies in the provided data. By taking for example the fully-connected model, it is considered a very general machine learning algorithm, which makes almost no assumptions about the underlying data structure. While this is typically not a problem, as theoretically it is an universal function approximator, in practice it is often beneficial to design the model architecture in a way, that it accommodates well to the specific task. This is particularly the case in problems, where there is a temporal dependency between the provided inputs features. Natural language processing is a very relevant example for such use-case, as data usually comes in streams of words, where each sample is dependent on the previous elements. Recurrent neural networks (RNNs) are specifically designed for making use of sequential data, and working with arbitrarily long inputs. The basic RNN model is almost entirely identical to the fully connected neural network, as the only difference is a recurrent connection in the hidden layer. This new addition serves as a memory for the model, which enables it to maintain information from previously computed hidden states.

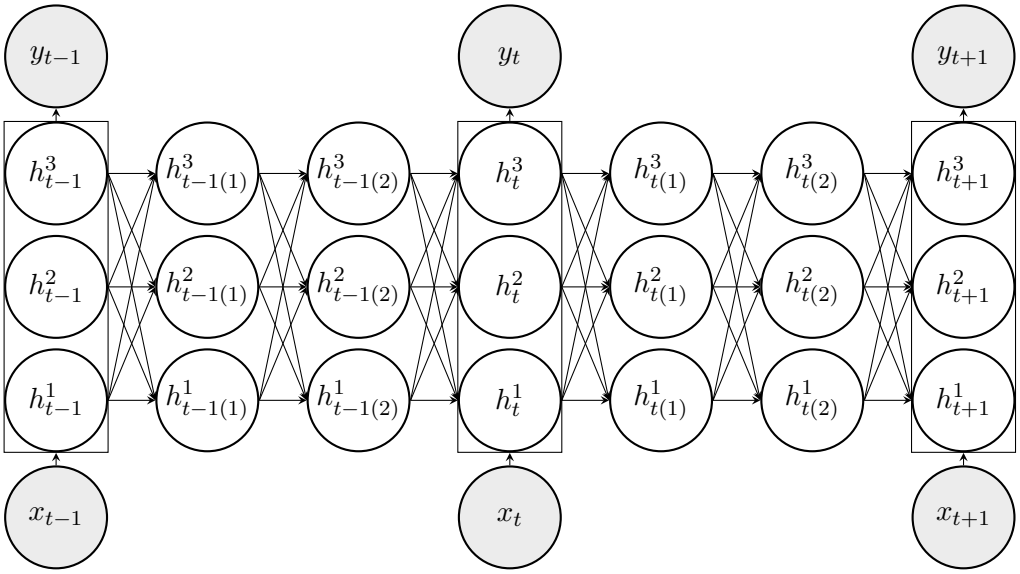


Figure 3.3: Illustration of a recurrent neural network unfolded in time from Wibrow.

In the following paragraph I explain the notations of figure 3.3. In contrast to the simple fully-connected model, the computational graph of the RNN has a horizontal axis as well as a vertical axis. As the depiction shows, the latter visualizes the forward propagation in the model, which is identical to the operation in the fully-connected architecture. The horizontal axis, however illustrates the progression in time. The input at a given timestep t is denoted by the variable x_t. At every t a different sample from the sequential data is processed, which updates the recurrent state of the model. This way at time step t, the model can store information from each of the previous input data points. The model in figure 3.3 also applies two intermediate transformation steps to obtain the final hidden state, which consists of three hidden neurons denoted by h^i. The output for a given t is

denoted by y_t. Considering the parameter matrix between the input x and hidden state h as W^{xh}, the parameters between hidden state and output as W^{hy} and the recurrent connection by W^{hh}, the equation in 3.12 shows the computations performed by a simple recurrent neural network. The bias element and the activation function is denoted by b and f.

$$f(x_t W^{xh} + h_{t-1} W^{hh} + b^{xh}) = h_t \tag{3.11}$$

$$f(h_t W^{xh} + b^{hy}) = y_t \tag{3.12}$$

3.7.1 Backpropagation through time

The simple backpropagation algorithm is basically the efficient implementation of computing derivatives with the chain rule for neural networks. Backpropagation through time is a modification of this approach, which accommodates for the architectural difference of RNNs. In order for the model to be able to learn dependencies in sequential data, the error signal must be propagated backward in the unrolled computation graph of the neural network. Therefore, the the final gradients for the parameter matrices in the t^{th} time step come from the loss, which is computed at the current and each subsequent time-steps. Notice, that for longer sequences this formulation yields the already discussed vanishing gradient problem, which prevented the wide-spread adaptation of recurrent neural networks for a long time. By propagating the error signal over several time steps, the value of the gradient gradually diminishes due to the derivatives of sigmoid and tangent hyperbolic functions, and the network will not be able to learn longer dependencies. This issue is circumvented by the invention of a more sophisticated architecture, which is presented in section 3.7.2.

3.7.2 Long short-term memory

In Hochreiter and Schmidhuber [1997] the authors describe a new architecture, the long short-term memory (LSTM) cell, which is aimed at solving the vanishing gradient phenomena for recurrent neural networks. They introduce an extension to the vanilla RNN model, by adding a new path for the gradient flow, without any intermediate activation functions. This alternative connection is called cell state, which is denoted by c. The LSTM model is able to write or delete information from this state by transforming it with various computational gates. In the following paragraphs, the paper covers the different operations in this architecture. One of them is the forget gate, which is designed to delete parts of the passed cell state, through point-wise multiplication with a vector of ones and zeroes.

$$\sigma(W_f * [h_{t-1}, x_t] + b_f) = f_t \tag{3.13}$$

The following operation is the input gate, which adds information to the cell state from the current time step. This is performed by a sigmoid layer, which creates a similar point-wise mask like the output gate, for determining the values to be updated. A tangent hyperbolic operation then creates a new candidate cell state, that is merged with the output of the input gate through a point-wise multiplication.

$$\sigma(W_i * [h_{t-1}, x_t] + b_f) = i_t \tag{3.14}$$
$$tanh(W_C * [h_{t-1}, x_t] + b_f) = \tilde{C}_t \tag{3.15}$$

The cell state is then constructed by the addition of results from the input- and forget gate.

$$f_t * C_{t-1} + i_t * \tilde{C}_t = C_t \tag{3.16}$$

The final output is then generated from the resulting cell state, by transforming it through a point-wise multiplication with the masked version of the previous output state.

$$\sigma(W_o * [h_{t-1}, x_t] + b_o) = o_t \tag{3.17}$$
$$o_t * C_t = h_t \tag{3.18}$$

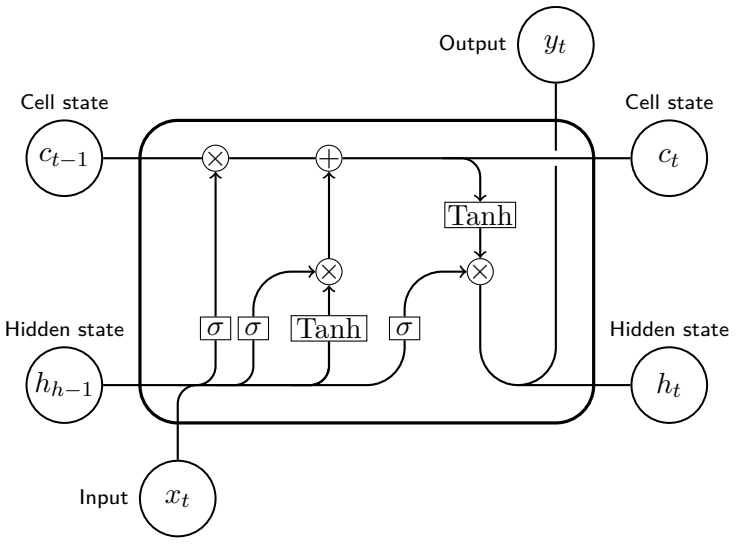


Figure 3.4: Illustration of an LSTM cell from V.

While this new architecture is theoretically able to solve the vanishing gradient problem and learn long-term dependencies in sequential data, in practice this is not always the case. Still, this architecture is a great leap compared to the vanilla RNN architecture, and outperforms it in every related problem.

3.8 Sequence to Sequence network

While LSTM models perform well on various natural language processing problems, such as language modelling, they cannot be used as simple end-to-end models for several use-cases like machine translation or dialogue generation. One of the great issues with single LSTMs in such problems, is that they are not designed to map inputs to output sequences with different lengths. By considering the following notation of input values x and output values y, the model has to compute the conditional probability for $p(y_1, \ldots, y_T | x_1, \ldots,$

$x_{\dot{T}}$), such that $y_1, \ldots, y_T$ is the output sequence, $x_1, \ldots, x_{\dot{T}}$ is the input sequence and T may not be equal to $\dot{T}$.

Sutskever et al. [2014] introduced a new architecture, that aims at solving the previously stated problem. The main idea behind their novel technique is to compose an encoder-decoder style architecture, by attaching two separate recurrent layers. The first RNN processes the provided input sequence, but instead of using its time step output, the final hidden state is passed to the second RNN module as initial state. This could be considered as the first model is the encoder, which collects the relevant information from the input sequence into a fixed-size latent vector, which is then leveraged by the second model that extracts the accumulated representation. Equation 3.19 illustrates a standard sequence generation with sequence-to-sequence architecture, where the encoded vector is denoted by v and every $\mathrm{p}(y_t|v, y_1, \ldots, y_{t-1})$ distribution is a product of softmax function over the possible output elements of the model.

$$p(y_1, \ldots, y_T | x_1, \ldots, x_{\dot{T}}) = \prod_{t=1}^{\dot{T}} p(y_t | v, y_1, \ldots, y_{t-1}) \tag{3.19}$$

This process is usually called as decoding, which is basically the same iterative operation, that is performed by a single recurrent neural network. The model also learns to terminate decoding, by adding a special marker token to the possible output values, which means the generation is finished, when the model generates this value.

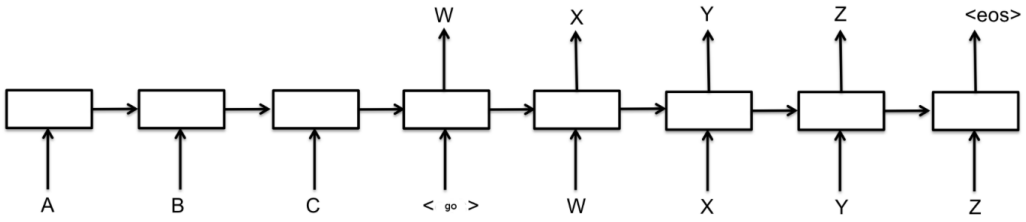


Figure 3.5: Illustration of Seq2Seq architecture. Sutskever et al. [2014]

Figure 3.5 is the illustration of the full Seq2Seq architecture, which receives the A, B, and C values as input, and generates the target W, X, Y and Z sequence with the final *<eos>* marker token. While the core idea behind this model has been considered as the state-of-the-art approach for various sequence-to-sequence problems, like neural machine translation, the basic model still suffers from the problem of RNNs, which correctly remembering and handling long-term dependencies. By considering a simple use-case, where each input for the model is a single word from a sentence, it is fairly common to have longer sequences of inputs or outputs for the model. Due to the number of transformations for the hidden state in such scenario, the information in the beginning of the sentence is likely to be distorted by final parts of the input sequence. There are several approaches, which can circumvent this issue up to an extend, like applying a backward RNN model to the input for creating a bidirectional encoding of the source. The two resulting hidden states are then merged into a single vector, which is passed to the decoder as discussed previously. In the following section I describe a different technique, that has risen sequence-to-sequence architectures to the state of the art end-to-end systems in machine translation Wu et al. [2016].

3.9 Attention Mechanism

The origin of attention in natural language processing goes back to the Seq2Seq architecture from Sutskever et al. [2014]. They proposed a new kind of neural network model, which has an encoder-decoder structure for mapping together variable-length sequences. The model is auto-regressive at every time step, as it receives the previously generated symbols when generating the next output. This new method allowed the implementation of well-performing end-to-end neural machine translation algorithms by mapping the utterances of a source language to a target language. At the time, the state-of-the-art systems in machine translation were still based on statistical methods, however among others, Bahdanau et al. [2015] pointed out potential issues with the vanilla sequence-to-sequence approach. The main problem of the architecture is that it has to compress all necessary information from the input sentence into a single fixed-length latent representation. As shown by Cho et al. [2014] this proved to be a performance bottleneck in the model, since by increasing the length of the input sequence the performance of the architecture deteriorates compared to shorter sequences of input. The experiments in the mentioned work were carried out on neural machine translation task, however the results can be generalized to other natural language related problems like abstractive summarization or dialogue generation.

To address this problem, Bahdanau et al. [2015] proposed an extension to the encoder-decoder structure called attention mechanism. Each time the model generates an output, it searches the input sequence for the most relevant part, where the necessary information is concentrated. The context vector is then computed from the weighted sum of these associated source positions for every predicted output vector, rather than single time at the end of encoding as it happens in the vanilla Seq2Seq model. This whole mechanism is differentiable and is trained jointly with the given task, thus the letting the model learning to attend by itself. This adaptive approach relieves the encoder from having to squash the full content of the source sequence into a single representation. The following equation in 3.20 shows the computations for the alignment weights, which is used to assign a weight value to every vector in the input sequence.

$$a_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T_x} exp(e_{ik})} \tag{3.20}$$

where $e_ij = a(s_{i-1}, h_j)$, and s_i is the hidden state of the decoder at the i-th time step. a is an arbitrary similarity function, that assigns higher value to those h_j values, which should heavily influence the content of the created context vector.

The work of Bahdanau et al. [2015] heavily influenced the field of natural language processing, and published numerous follow-up work on this area. Using attention mechanism in sequence to sequence architecture became a staple, as more effective similarity functions arose. Among others, Luong et al. [2015] summarized existing attention mechanism architectures and also introduced several better techniques than their predecessors. In 3.21 and 3.22 two Luong-style attention compatibility score methods are shown. 3.21 uses a trainable parameter matrix and is referred to in their work as general attention mechanism, and 3.22 that uses the dot product of the s_i and h_j vectors. Note that the latter approach places a constraint over the encoder and decoder, such that the produced representation of the decoder should be similar to the relevant parts of the encoder states.

$$e_{ij} = h_j W_a s_i \tag{3.21}$$

$$e_{ij} = h_j^T s_i \tag{3.22}$$

$$e_{ij} = v_a^T tanh(W_a[h_t; h_s]) \tag{3.23}$$

3.10 Transformers

The next breakthrough in sequence to sequence architectures was the work of Vaswani et al. [2017], which introduced the Transformer model. They replaced the recurrent neural network encoder and decoder in the Seq2Seq model with several layers of fully connected neural networks and attention mechanisms. It still has an encoder-decoder structure, that given a sequence of inputs, generates a sequence of output auto-regressively. The model has a stack of identical blocks in the encoder and the decoder as well, with an embedding layer and a position encoder layer. The latter is required, since the lack of recurrent and convolution operations negate the model from having spatial information about the input sequence.

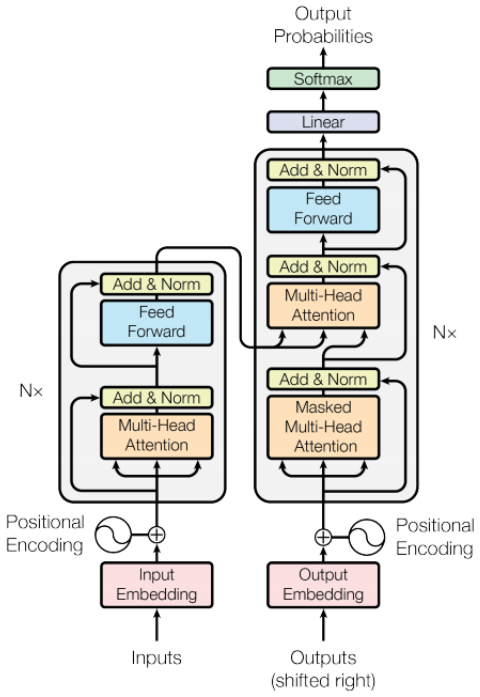


Figure 3.6: Self-attention and multi-head attention from Vaswani et al. [2017].

According to the authors description, the encoder has 6 layers, where every layer consists of two sub-layers. The first is a multi-head self-attention, followed by a fully-connected neural network. To mitigate the problem of vanishing gradients, they also use a residual connection around both sub-layers. The output is then transformed by a layer normalization block, which is described in Ba et al. [2016]. All of the mentioned input- and output tensors have an identical dimension, – as the residual connections require it – that is $(batch\_size, seq\_len, hidden\_dim)$.

Just like the encoder, the decoder has a stack of 6 layers, however in addition to the multi-head self-attention and fully connected layers, there is also a third sub-layer between the two, that performs multi-head attention over the output of the encoder. These sub-layers are also paired with a residual connection. The attention mechanism is modified to involve a masking operation as well, which is to prevent the decoder during training from attending to subsequent positions. This ensures, that the predictions of the model are not dependent on positions that are forward into the future, as it obviously does not have access to such information during inference.

3.10.1 Scaled Dot-Product Attention

According to Vaswani et al. [2017] an attention function is an operation that maps a query and a set of key and value pairs to an output, where every component of the operation is a vector. The produced output is created by the weighted sum of the values, where a the weights for each value is computed by a compatibility function between the query and the corresponding key. So far this technique is in line with the attention mechanism described in the previous section. In Transformer architecture, the authors add minor improvements to this method to get their special type of attention mechanism – called scaled dot-product attention –, where the inputs are key and query vectors with size of d_k, and values with size of d_v. As the name suggests, they use dot product as their compatibility measure between the queries and keys, and normalize the result with $\sqrt{d_k}$. The regular softmax function is then applied to obtain the weight for the values. The attention computation mentioned in the paper is shown in equation 3.24, where z_i is an output element, and x_j is the part of an input sequence $x = (x_1, ..., x_n)$ with size n. For every j, $x_j \epsilon R^{d_x}$ and every i, $z_i \epsilon R^{d_z}$. As a side note in practice typically it is true that, $d_x = d_z = d_k = d_v$.

$$z_i = \sum_{j=1}^{n} a_{ij}(x_j W^V) \tag{3.24}$$

Each coefficient a_{ij} is computed via equation 3.20, where e_{ij} is computed by the mentioned dot-product compatibility function. This is shown in equation 3.25, where $W^Q, W^K, W^V \epsilon R^{d_x \times d_z}$ are parameter matrices.

$$e_{ij} = \frac{(x_i W^Q)(x_j W^K)^T}{\sqrt{d_z}} \tag{3.25}$$

As Vaswani et al. [2017] states, additive attention 3.23 and dot product attention 3.22 are the most commonly used techniques. The latter is actually the basis for scaled dot product attention, with the addition of the d_k normalizing factor. They claim that having a feed-forward network with a single hidden layer in additive attention is similar to the multiplicative method in theoretical complexity, but computing a single dot product is faster and way more memory efficient.

According to their experiments, the two mechanism yields similar performance, as long as the value of d_k is low, however additive attention outperforms the dot product method for larger d_k values. This comes from the fact that the dot product may grow large for such d_k values, bringing the softmax output space into regions, where the gradients are too small. This is countered by having a $\frac{1}{\sqrt{d_k}}$ scaling factor.

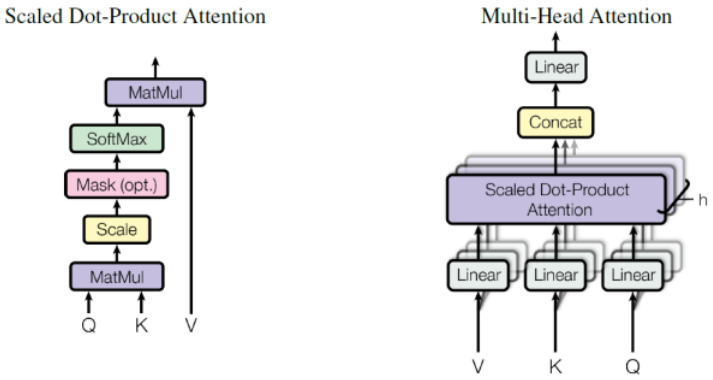


Figure 3.7: Self-attention and multi-head attention from Vaswani et al. [2017].

3.10.2 Multi-Head Attention

Another idea Vaswani et al. [2017] employed in the attention layer of the Transformer architecture is instead of a single attention function, there are multiple different projections of the queries, keys and values. The general idea is to split d_k and d_v dimensional vectors into n linear projections, where n evenly divides d_k and d_v to obtain $\frac{d_k}{n}$ independent attention heads. These projections are then used simultaneously in the computations until they are projected and concatenated once again back to the original d dimensions. Usually to take advantage of highly parallel hardware acceleration, it is a common practice to apply the attention operation to a batch of queries at the same time, packed into a single matrix Q, while the keys and values are packed into matrices K and V. For a better overview 3.26 shows the mechanism of the full multi-head attention with the scaled-dot product in 3.26, 3.27 and 3.28.

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W_o \tag{3.26}$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{3.27}$$

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{3.28}$$

The main goal of multi-head attention is to have different representations at different positions, and allow the model to aggregate compatible vector position in different ways. Voita et al. [2019] examines this behaviour and shows that the different heads often play important and linguistically-interpretable roles. They identify several possible functions, which heads might be playing, that are signaling word positions, syntax and rare tokens. Positional information refers to heads that always point to the adjacent words of a position. Other heads might point to tokens with specific syntactic relation to another, like in case of sub-word tokenization, one of the heads often learns to group divided parts of the word. They also hypothesize that when syntactic heads are present in the Transformer, it may be responsible for the disambiguation of the source sentence's syntactic structure. Rare words are also often highlighted in some attention heads to point out the least frequent tokens in a sequence.

3.10.3 Layer Normalization

Normalization is a key technique in machine learning for handling unbalanced data scales in training data. By re-scaling and re-centering input features around an uniform value, the training of neural networks is made more stable and faster. The main concept behind the preliminary work for layer normalization builds on this idea, which is the batch normalization from Ioffe and Szegedy [2015]. The work discusses the issue of internal covariate shift, and presents their solution as the latter technique. They recognize that parameter intialization and distribution changes in the input of each layer can affect the learning effectiveness of the model. This problem is severely present in deep multi-layered architectures, where the parameter change of certain layers affect the input distribution of the following layers, thereby the model has to constantly adjust to new distributions. To demonstrate the effect of batch normalization, let us consider a fully-connected neural network with an input vector x and output vector y. a^l denotes the summed output of neurons and h_l is the bottom-up input in the l^{th} layer. The summed outputs are computed accordingly to equation 3.7, which is depicted in equations 3.29 with the current notation.

$$a^l = h^l W^l \qquad h^{l+1} = f(a^l + b^l) \tag{3.29}$$

Batch normalization method re-scales the values of a^l according to its variance from the distribution of training data. Equations in 3.30, 3.31 and 3.32 shows the performed operations.

$$\overline{a}^l = \frac{g^l}{\sigma^l}(a^l - \mu^l) \tag{3.30}$$

$$\mu^l = \mathop{\mathbb{E}}_{x \ P(x)\in A}[a^l] \tag{3.31}$$

$$\sigma^l = \sqrt{\mathop{\mathbb{E}}_{x \ P(x)}[(a^l - \mu^l)^2]} \tag{3.32}$$

$\overline{a}^l$ is the normalized summed neuron output and g is the gain parameter that is used for scaling the outputs of the activation function. While the equations in 3.30 show, that computations require an expectation under the given parameter state over the whole training data, it is impractical to calculate it, as it would require evaluating the model on the dataset between optimization steps. Therefor μ and σ are approximated by samples from the currently processed batch of inputs. While this approach is easily applicable in certain machine learning settings, using it for time-step based auto-regressive models is considerably inefficient and harder.

The layer normalization method from Ba et al. [2016] is designed to overcome these issues. They propose that changes in the output of a given layer shows high correlation with the changes in the output of the next layer. Therefore they argue, that covariate shift problem can be mitigated by setting the mean and variance of the output values in each layer separately. Equations in 3.33 and 3.34 show the computations of normalization parameters in a given layer l.

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \tag{3.33}$$

$$\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2} \tag{3.34}$$

H is the size of the l^{th} hidden layer. As it is depicted in the above equations, the normalization terms μ and σ are shared across the values in each layers. Unlike in batch normalization, layern normalization does not constrain the batch size and it can also be used with a single training example as well.

3.10.4 Position Embedding

As mentioned before, it is necessary to provide information to the model about the word order in the sequence. This is done by having an extra position embedding layer, with the same dimension as the regular embedding layer, so their output can be conveniently summed. This position embedding also presents an architectural choice, as it can be either absolute or relative, learned or fixed. In the original model, Vaswani et al. [2017] chose a fixed, absolute position embedding method using a signal of cosine and sine function with different frequencies, such that each dimension of the position encoding vector corresponds to a sinusoid. In their experiments this signal has a progression in the wavelength from 2π to 10000 2π. The exact formula from the paper can be seen in 3.35 and 3.36.

$$PE(pos, 2_i) = sin(pos/10000^{2_i/d_{model}}) \tag{3.35}$$

$$PE(pos, 2_{i+1}) = cos(pos/10000^{2_i/d_{model}}) \tag{3.36}$$

They hypothesized that, since this function with a fixed offset k for every PE_{pos+k} can be represented as a linear function of PE_{pos}. The main advantage of using a fixed positional embedding with the above formula, is that it can be easily generalized to longer sentences than seen in training. Vaswani et al. [2017] also experimented with using learned position embeddings, however they produced similar results. Contrary to the author's approach, learned positional embedding does not allow sequences with greater sequence length than the ones seen during training, and there is also a problem with sparse updates for the larger sequence positions.

While the performance of absolute position encoding is sufficient, Shaw et al. [2018] showed that using relative position encoding approach yields better results. Contrary to the original technique, where the position information is only added a single time to the input and propagated to higher layers with residual connections, relative position information is incorporated into every attention computation in each layer. This new mechanism is called relation-aware self-attention, which is an extension to consider pairwise relationships between the elements of the input. This is applicable to any kind of relation, however the author's work is only concerned with position information. To represent pairwise connections for the model, the conventional vector input has to be replaced with a matrix for each sequence. This can technically be considered as a fully connected graph, where x_i and x_j are elements of the input sequence, which are represented by vectors $p_{ij}^V, p_{ij}^K \epsilon R^{d_p}$. d_p is the dimension of the relative position embedding matrix. Note that by having pairwise connections of the input sequence, the memory complexity is increased from $O(hnd_n)$ to $O(hn^2d_a)$, meaning it is great drawback performance-wise. The paper discusses the author's solution for this problem in a following section, after introducing the method's

modification to the regular scaled dot-product attention operation from equation 3.24. In relation-aware self-attention the computation is modified to include edge information, which is shown in 3.37. It is also important to note the modifications in the compatibility function from 3.25, which can be seen in 3.38.

$$z_i = \sum_{j=1}^{n} a_{ij}(x_j W^V + p_{ij}^V) \tag{3.37}$$

$$e_{ij} = \frac{x_i W^Q (x_j W^K + p_{ij}^K)^T}{\sqrt{d_z}} \tag{3.38}$$

As explained by the authors, having simple addition between the key vector product and the relation information matrix enables an efficient implementation, which relaxes the memory consumption problem, that I mentioned previously. By abusing the nature of matrix operation order, they manage to decrease the space complexity from $O(n^2 d_p)$ to $O(n^2 d_a)$ by sharing them across each head. Moreover, the relation matrix can also be shared between each sequence in a batch, thus increasing the space complexity of self-attention from $O(bhnd_z)$ to $O(bhnd_z + n^2 d_a)$. As explained in previous sections regarding self-attention, the operation without relative relation representations can be computed with bh number of parallel matrix multiplications, where b is the batch size, and h is the number of heads in the attention layer. Each matrix product is computed between $n \times d_z$ and $d_z \times n$ sized matrices. This assumes that for each sequence and each head, the same representation is shared for every position in all compatibility function operations. However, this is not true in case of relative positions, since they are not the same for different pairs. Therefore, the e_{ij} can not be computed in a single matrix multiplication. This issue can be solved by using the author's method, which splits the computation of 3.38 into the following terms in 3.39.

$$e_{ij} = \frac{x_i W^Q (x_j W^K)^T + x_i W^Q (p_{ij}^K)^T}{\sqrt{d_z}} \tag{3.39}$$

The first part of the numerator is identical to 3.25, while for the second part tensor reshaping can be used to compute the parallel matrix multiplications of $bh \times d_z$ and $d_z \times n$ sized matrices. These matrix products correspond to the e_{ij} values for every position pair in the sequence.

According to the authors' experiments, the modifications to the original method showed 7% drop in computational speed of the model. This is a modest decrease, considering slight improvements in machine translation performance. As a summary based on the authors' hypothesis and experiments, including relative relation information method in a transformer is can be important for numerous tasks. It is especially the case for problems, where the the model benefits from having access to the type of an edge in downstream encoder and decoder layers, which is not the case with machine translation. In dialogue generation, however encoding the identity to whom a particular utterance in the dialogue belongs to, is essential for the model.

3.10.5 Applications

In the field of computer vision, the trend shows that using pre-trained large models for transfer learning on a particular downstream task achieves better performing models, than

always coming up with new architectures and training them from scratch. ImageNet models for instance played a huge part in bringing computer vision problems more accessible for smaller research teams, companies or academy, where large data and computational resources are not always available. In the recent years the same practice seemed to arise in natural language processing, where one of the first pre-trained model with great transfer-learning potential was ELMo from Peters et al. [2018]. It is important to note that even before this model, pre-trained resources were available for researchers in the form of word embeddings like Word2Vec from Mikolov et al. [2013a]. The problem with these models is that they lack dynamic contextual information, whereas words or expressions are often ambiguous and their meaning is highly dependent on other parts of the sentence. ELMo is among the first models, that aims at solving this problem by computing dynamic meaning representations for each word, based also on the surrounding context. During pre-training it performs language modeling, where the objective is to predict the next element, given the previous elements of the sequence. During this time, the model is able to learn essential properties of the language, which can then be applied on purpose-specific setting with less training data. From the perspective of the downstream task, it is considered as a feature-based method, meaning the representations from the model are used as additional features in the input of the task-specific architecture. Another approach is fine-tuning, where the model uses a minimal set of task-specific parameters and is trained by fine-tuning the whole model on the particular problem. After the success of ELMo, other pre-trained language model architectures seemed to emerge, which employed the advancements of Transformer model from Vaswani et al. [2017]. These models generally perform a variant of language model pre-training objectives, however they usually follow the fine-tuning approach, which tends to give better results. In the following sections I present some of these Transformer-based language models, which I either use in my work, or they influence it heavily.

BERT (Bidirectional Encoder Representations from Transformers) from Devlin et al. [2018] is a recent paper published by Google researchers. They argue that a major limitation for standard language model is using an unidirectional technique, which limits the choice of architecture during pre-training and restricts the effectiveness of trained representations. In a left-to-right model, a token can only depend on previous tokens in self-attention, which is sup-optimal for fine-tuning cases where context is important from both directions. For example, in case of sentence level tasks, when there are ambiguous words in the beginning of the text, the model has a hard time creating good representations, since related information might be at the end of the sentence. They propose a new kind of training method called masked language modeling (MLM), that alleviates the constraint of the unidirectional approach. The main idea behind this technique, is to take the whole sequence during training and apply random masking over some of its elements. The objective of the model is then to correctly find the masked values, based on the surrounding context. Unlike single-direction language models, MLM is able to use the representations from both directions, which is supposed to fix the previously stated problem. The authors also use an auxiliary objective during pre-training, where the model has to predict, whether two given sentences are consequent or not. This is called next sentence prediction, which, hypothetically constrains the model to learn higher level structures in the language. BERT managed to achieve state-of-the-art

The vocabulary of BERT consists of 30000 tokens, which includes special tokens like [CLS], that is used as the final hidden state for aggregating sequence representation for classification related problems. [SEP] is another special token, with the purpose of signaling the separation of two sequences. The authors use WordPiece tokenization algorithm

from Schuster and Nakajima [2012], for handling unknown words, and more efficient word representations.

GPT-2 (Generative Pre-training Transformer) method was first introduced in Radford [2018], which was the basis for the GPT-2 technique. Similarly to the original transformer paper, GPT uses autoregressive language modeling, however it only leverages the decoder part of the transformer architecture and employ minor modifications, like changing the order of layer normalization in the residual feed-forward layer and use 1D convolutions instead of regular fully-connected layer. Contrary to BERT, this model uses unidirectional training method, which makes it inferior in representation related problems. While it is generally outperformed by bidirectional language models on language understanding problems, GPT-2 is currently one of the best generative model. The authors' trained and benchmarked four language models with increasing number of parameters. The smallest model is equivalent to the original GPT, and the second smallest equivalent to the largest model from BERT. The largest model, which they call GPT-2, has over an order of magnitude more parameters than GPT (1.5 B). The model operates on a byte level and does not require lossy pre-processing or tokenization. Byte-pair-encoding from Sennrich et al. [2016] is used, which enables the model vocabulary to only have very minimal number of out-of-vocab tokens in the whole (40 Gb) training corpora.

XLNet is another autoregressive language modeling approach, which is suitable for sequential text generation. Yang et al. [2019] states that models with the capability to model bidirectional contexts, like denoising autoencoding based pre-training with BERT achieves better performance than pre-training approaches based on autoregressive language modeling. However, by corrupting the input text during masked-language modeling neglects dependency between the masked positions and suffers from a pretrain-finetune discrepancy. To avoid this XLNet uses permutation language modeling, so instead of using a fixed forward or backward factorization order as in conventional autoregressive models, XLNet maximizes the expected log likelihood of a sequence with respect to all possible permutations of the factorization order. XLNet also applies modifications to the model architecture, like having a two-stream self-attention for target-aware representations. This technique is required for the model to learn useful representations during permutation language modeling, because standard softmax autoregressive objective approach would fail otherwise. There is also a concrete example given by the authors. Considering two permutations $z^{(1)}$ and $z^{(2)}$, which satisfy the equations.

$$z^{(1)}_{<t} = z^{(2)}_{<t} = z_{<t} \tag{3.40}$$

$$z^{(1)}_{t} = i \neq j = z^{(2)}_{t} \tag{3.41}$$

Then by applying the permutations respectively with the parameterizations the result is:

$$p_\theta(X_i = x|x_{z<t}) = p_\theta(X_j = x|x_{z<t}) = \frac{exp(e(x)^T h(x_{z<t}))}{\sum_{x'} exp(e(x')^T h(x_{z<t}))} \tag{3.42}$$

Meaning that two different positions i and j have the same output prediction, while the target distribution for the two positions should be different. To avoid this this problem,

they propose target aware re-parametrization of the standard softmax next-token formulation so the output will be position aware.

$$p_\theta(X_j = x|x_{z<t}) = \frac{exp(e(x)^T g_\theta(x_{z<t}, z_t))}{\sum_{x'} exp(e(x')^T g_\theta(x_{z<t}, z_t))} \tag{3.43}$$

Where $g_\theta(X_{z<t}, z_t)$ is new type of representation, which also takes the target position z_t as input. This method also requires implementing more complex input pipeline, including the generation of correct masking and relative positional encoding. I explain my take on these technical difficulties in the implementation section.

XLNet uses sentence-piece tokenization, which is an unsupervised text tokenizer and detokenizer for neural language tasks. It is internally based on an efficient re-implementation of byte-pair-encoding and unigram language model. The main difference from the original BPE is sentence piece uses BPE on word piece tokens separately.

3.11 Knowledge graphs

Grasping the exact definition of knowledge graphs is hard, as it is a generalized term with various alternative names. For the purpose of presenting a standardized outline of this subject, the following subsections are based on the work of Bonatti et al. [2018].

In computer science and machine learning related use-cases, knowledge graphs serve as an effective way of representing information about entities through their relations to each other. They are effective in that they present knowledge in a human-readable format, while keeping it easily processable by machines. Knowledge graphs have lately emerged as an essential way of merging separate data-sources and modeling their structure for use-cases like search engines or recommendation systems. They can help the interpretation of natural language queries, by providing contextual understanding of names, entities or phrases in specialized domains.

Knowledge graphs belong to a family of graphs, which is an abstract mathematical terminology. As discussed by Balakrishnan and Ranganathan, it stands for a structure, that is made of a set nodes and edges that interconnects them. Each edge shows a specific relationship between its two nodes, which can be either directed or undirected. Directed relations can be interpreted as an asymmetric connection between two entity nodes, whereas undirected edges are used for signaling symmetric relation type. Another property of graphs is homogeneity. A graph is considered homogeneous if each vertex represents the same type and each vertex is the instantiation of the same relation. For example homogeneous and undirected graphs are usually used in social networks, where acquaintance between two person is modelled as a symmetric edge connecting the corresponding entity nodes. As another example, product recommendation systems might maintain encodings of several different entity nodes, such as retailers, customers and products, where connections between these vertices could involve product availability, purchase history or customer interest. With several entities and multiple potentially asymmetric relations between them, such graph is considered directed and heterogeneous. Notice that in more complex use-cases, it is common to have several different types of relations between a pair of entities. There might also be cases for specific structures, where self-loops are necessary to model a particular relation. Graphs, that contain such connections are called multigraphs, whereas graphs without multiple edges and self-loops are called simple graphs. By using the previously mentioned terminologies, the definition of knowledge graph can be pinned down as

a heterogeneous, directed multigraph, where the types of entities and relations are defined by its domain of usage. For the purpose of better understanding its main idea and functionality, in the following paragraphs the paper discusses the hierarchical categorization of several other related terminologies from Trey Grainger, which will be used as a context for extending the idea behind the previous definition.

The figure 3.8 illustrates a hierarchical structure of various definitions, which are closely related to knowledge graphs. By traversing the depiction from inside to the outer parts, the first element is the alternative labels.

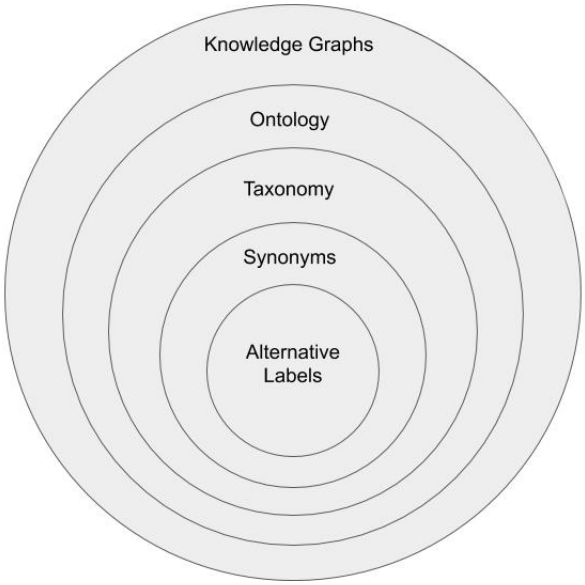


Figure 3.8: Illustration for the hierarchical terminologies related to knowledge graphs based on the presentation of Trey Grainger.

Alternative Labels is basically a simple mapping between different substitute words and their identical meanings. This technique is often used for fixing misspellings, where the frequently incorrectly written word forms are paired with their correct counterparts. Also a typical application for it could be a lookup table between words with different American English and British English spelling e.g: optimize → optimise. Another use-case might be conversion between acronyms and their resolved form: FBI → Federal Bureau of Investigation.

Synonyms List is a larger superset that could include alternative labels. While the latter technique is usually used for word pairs with identical meaning, synonyms list might relax this property, to also include mapping between very similar word meanings. Therefore in addition to the examples from the previous definition, synonym list could also include the following: drink → beverage, humanity → mankind. Maintaining a structured set of substitute words is extremely useful for domain specific and rule-based dialogue models or search engines, where the keyword recognizer algorithm must be flexible enough, to adjust for slight variations in word usage.

Taxonomy is a way of classifying entities or concepts into different hierarchical categories. While taxonomy was initially used in biology as a way of identifying, describing and grouping organisms, it was later adopted in other areas like computer science as well. Search engines for example are greatly benefited by its use, as using taxonomies helps improving

relevance in topical or special domain related queries. Examples for this definition are: The Lord of the Rings → book → object, chimpanzee → mammal → animal

Ontology is a way of defining relationships between specific entity types. While ontologies can be hierarchical similarly to taxonomies, they could also be used to represent more generalized meaning. They basically show relations and properties of a given subject area, by using a set of formally defined concepts and classes as entity nodes in the graph. They can be used to simplify and organize knowledge about a particular field, by reducing complexity of the used language, through a generally defined and controlled vocabulary. Examples for possible relationships, that might be defined by a given ontology: book is_an object, book contains pages, book was_written_in time

Knowledge graphs are very similar to ontologies, however, while the latter define relations between types of entities, knowledge graphs can be thought of as an instantiation of an ontology. By putting it simply, they would also actually contain instances of the generalized classes, that are defined by ontologies. As a clarification, in addition to the relations, that are given as examples for ontologies, a knowledge graph would also contain the following information triplets: The Lord of the Rings is_a book, The Lord of the Rings contains 400 pages, The Lord of the Rings was_written_in 1954.

3.11.1 Knowledge Graphs in Machine Learning

The properties of knowledge graphs give us numerous reasons for integrating them into machine learning algorithms. By providing a sparse, human-readable way of storing information they serve as a valuable extension to existing machine learning datasets. There is also a great effort in assembling large, collaboratively edited general knowledge banks, for example one of which is WikiData from Vrandečić and Krötzsch [2014]. WikiData is a completely free and open database, which serve as the central storage for sturctured data in Wikipedia.

Although there is a great rise in the availability of machine learning datasets, there are several areas, where large and diverse corpus are not existing. Since it is often essential to have an appropriate amount of training examples for the model, the lack of it means a barrier for developing usable machine learning algorithms. While this could be solved by simply adding more data points to existing smaller training datasets, doing so requires expensive human labour, various specialized tools and other resources. The problem of data insufficiency could serve as a great opportunity for augmenting those smaller datasets with pre-assembled knwoledge graphs, such as the mentioned WikiData. Potential impact of this is even greater in specific fields, where the trained model has to be robust to moderate variations in the input data points, which is definitely the case in natural language processing. One example use-case would be switching certain named entities in text to similar candidates, based on the relations inferred from the knowledge graph. This approach provides a great amount of additional examples, which heavily improves robustness.

Another approach is integrating these data banks into the model architecture as an external memory source. In the area of question answering, similar methods are often used in the form of a ranking problem. This basically means that the model has access to large amounts of data examples during making a prediction, and is trained to leverage or extract the required information from them. One way of implementing this system is by providing raw textual documents, such as Wikipedia articles, which are then used accordingly to the previously described method. While this works well in practice, managing and editing the mentioned documents for adding or deleting information is a cumbersome

task. In contrast, a knowledge graph representation of a Wikipedia article contains much less redundancy, and modifying or regulating its content is trivial.

In section 3.2.3 the paper discusses an important subject for machine learning algorithms, which is interpretability. The usage of end-to-end deep neural networks for various natural language understanding problems usually does not leave room for easily explainable predictions. This is especially true for certain generative models, which in my case serve as the basis of the proposed dialogue transformer architecture. By incorporating knowledge graphs into such models, there is at least a slight opportunity for debugging or regulating these architectures, through manually modifying the stored data. For example by assuming that model fully leverages the provided information from the graph, and we query for information that is contained in the graph, we expect the model to produce a corresponding result. This stands in contrast to the general case, where the stored information is encoded into the internal parameters of the model, which can hardly be inspected or modified.

3.11.2 Graph Embedding

In previous paragraphs the paper gave a broad overview about the theory of knowledge graphs and their potential impact on machine learning, however the paper has not covered the technical details about their integration with neural network architectures. During the upcoming paragraphs I introduce several techniques for this task, but before proceeding I clarify some important terminologies. In a knowledge graph setting, the traditional graph theory names of edges and vertices, are usually referred to as entities and relations correspondingly. Another important concept is the *triplet*, which is the name for a pair of entities with a directed relation between them. A triplet is usually represented as a tuple (h, r, t), where h marks the head entity, r is the relation and t denotes the tail entity.

While there is not a single method for converting graphs to a format that is interpretable by a machine learning model, in this work I only focus on graph embedding techniques. As a motivating example for this subject, I reach back to section 3.5, where the paper discusses word embeddings. They enable the conversion of sparse features to dense vector representaitons, which is basically the main concept behind graph embeddings as well. As an introductory example, the paper demonstrates a specific technique for obtaining representations for the nodes of a graph, which heavily builds on the previously described Skip-gram word embeddings. The name of this approach is DeepWalk by Perozzi et al. [2014], that learns to project graph vertices to a latent vector space through performing random walks between the nodes. During each walk, the identifiers of the involved graph nodes are stored as a sequence, which is then used as the input for the Skip-gram model. Notice the similarities between the general word-based approach and DeepWalk, as each sequence of nodes can be considered as sentence of words, where the input and target elements are determined with the same sliding-window method. The main drawback of this technique for my experiments, is that the types of relations between nodes are not taken into consideration, when learning embeddings. This is a problem, since even the smallest knowledge graph almost always has several types of relationships. In the following paragraphs I introduce several other approaches, which overcome this issue and serve as great candidates for my experiments. These generally provide low-dimensional representations for different types of entities and relations through some kind of score function, which measures distance between two entity nodes with regard to their relationship in a vector space. The objective is then to train the embedding parameters, such that entities with a specific relation between them are close to each other, while other entities are distant.

TransE from Bordes et al. [2013] attempts to solve the problem of modeling multi-relational graphs by representing entities and relations in the same low-dimensional embedding space, where relations are interpreted as a translation operation. The main idea behind this technique in terms of vector computations, is that by considering a fact as a triplet with the defined (h, r, t) notation, the representation of head entity $\vec{h}$ translated by $\vec{r}$ relation should approximate the tail entity $\vec{t}$. The operation is depicted in equation 3.44 for clarity.

$$\vec{h} + \vec{r} \approx \vec{t} \tag{3.44}$$

The distance function for TransE can then be derived from this, which is shown in equation 3.45.

$$d(h, r, t) = |\vec{h} + \vec{r} - \vec{t}|_2^2 \tag{3.45}$$

Considering a set of triplets S and a margin hyperparameter γ, the objective is then to minimize a margin-based ranking loss over the training examples, which is shown in equation 3.46.

$$\mathcal{L} = \sum_{(h,r,t) \in S} \sum_{(h',r,t') \in S'_{(h,r,t)}} [\gamma + d(h, r, t) - d(h', r, t')]_+ \tag{3.46}$$

By building on the discussed notion of learning the similarity between $\vec{h} + \vec{r}$ and $\vec{t}$, the margin loss introduces corrupted triplets S', such that either h' or t', but not both, is sampled from a set of entities, where (h', r, t') is not an existing triplet. As discussed by the authors, this framework provides an efficient way for learning entity and relation representations for graphs with overwhelming one-to-one connections, however it falls short for more complex, 1-to-N cases. This problem can be accounted for the fact, that by using a single vector for a relation, the model is only capable of learning one aspect of similarity between the two entities.

TransR from Lin et al. [2015] aims at creating a graph embedding technique, which is applicable to 1-to-N type relations as well. They address this problem by maintaining different latent space for relations, which does not need to have the same dimensions as the entity embedding space. In addition to a single vector, TransR learns a projection matrix for each relation, which is used to project the entities to the corresponding space. By using the same notation as in previous paragraphs, this operation can be seen in equation 3.47, where M_r is the projection matrix.

$$h_r = \vec{h} M_r \qquad t_r = \vec{t} M_r \tag{3.47}$$

By building on these computations, the d distance function for TransR method is then formulated in equation 3.48.

$$d(h, r, t) = \|\vec{h} M_r + \vec{r} - \vec{t} M_r\|_2^2 \tag{3.48}$$

Other than this single operation, the training objective and loss function is identical to TransE, which is depicted in equation 3.46. As the experiments by the authors suggest, due to its separated relation spaces TransR is capable of learning multiple aspects of

similarities between entities, however this comes at the cost of increased parameter count and model complexity.

As it is generally true for machine learning problems, choosing one method over another confronts us with various trade-offs, which have to be accounted for ahead of proceeding with a specific approach.

Chapter 4

Implementation

In this chapter the paper discusses the structure as well as the main difficulties and remarks related to the implementation of my solutions. In the upcoming sections, The paper introduces the specifics of my chosen model architecture and go into more detail about several techniques regarding training.

4.1 Overview

As in the previously presented theoretical sections, this thesis focuses mainly on the implementation of a dialogue generation model, that uses knowledge graphs as external data sources for increasing its factual accuracy. The architecture consists of three main modules, an entity linker, a response generator and a graph decoder. In the upcoming paragraphs the paper gives a detailed description of my process for building the components, that are required for my work.

As an initial step, I implement a framework for building baseline dialogue models for my research, with the latest transformer-based language model architectures. By acquiring various conversational datasets, I perform several experiments with variations of GPT-2 and XLNet models in a multi-turn dialogue setting. As part of the procedure I thoroughly investigate different hyperparameter settings for the mentioned architectures, where the resulting model performance is compared with empirical methods in addition to some automatic metrics. Due to the lack of dataset relevant to this specific problem, I observe difficulties in measuring factual correctness of dialogue models through conventional validation techniques.

By acquiring a set of strong baseline models, I proceed to the implementation of a transformer-based architecture, which in contrast to the classical approach, uses continuous outputs in its final layer. After testing the capabilities of this model on the same datasets, which are used for the mentioned baselines, I focus on the inclusion of knowledge graphs.

As a preparatory step, I compare several graph embedding techniques on commonly used KG datasets. In order to better understand how specific embedding algorithms work, I implement some of the most commonly used methods in this field. After successfully reproducing several reported results from the corresponding works, I try out specialized machine learning libraries, that come with optimized implementations of the aforementioned techniques. After adapting the KG dialogue to the required format, I train a specific graph embedding algorithm with optimized hyperparameters.

The second step in creating the knowledge graph-enhanced dialogue transformer architecture, is modifying the encoder and decoder layers with an additional attention mechanism, that aggregate entity information from the provided source and target utterance pairs. While this extended model is capable of producing word representations as a response for the given context when the entity pairs are provided, during inference time, this information must be collected from the raw input text. The solution to this problem consists of two parts, for which I employ two separate neural networks. The first model is responsible for extracting entity mentions from the source utterance and dialogue history. The second model receives the representation of the found entities and performs a search from each entity node, for potentially relevant entities, which can be used for decoding.

After running an extensive set of experiments with different configurations, I arrive at the best performing variation, which is compared to the previously defined baseline models.

4.2 Frameworks

Accessibility of artificial intelligence has shown tremendous increase in the recent years. The spread of open-source deep learning frameworks, have enabled programmers without any background in machine learning to implement state of the art neural networks in their applications. Most of these frameworks are backed by companies like Google or Facebook, and it is in their best interest to extend the usage of these technologies among developers. In pursuit of greater number of users, deep learning library developers raised the bar in quality of documentation, flexibility and ease of use.

This progress caused a closing gap between engineering and research in machine learning, that allowed developers to bring new research ideas into production with a single code base. Today the tools are ready and accessible in all phases in the development cycle of a machine learning algorithm, from infrastructure through testing and diagnostics to serving and visualization. In the following sections I present my choice of frameworks and tools, which I use in my project.

PyTorch from Paszke et al. [2019] is among the most popular deep learning libraries, thanks to its flexibility and intuitive interface. It is the offspring of the Lua-based Torch library, which was brought to Python, as it became the most used programming language for machine learning. It is mainly under the maintenance of Facebook AI Research developers. Like most other framework, it supports reverse-mode automatic differentiation of functions, backed by highly optimized C++ core, with kernels for parallel matrix transformations on the GPU. Compared to other libraries it emphasizes approachability and pythonic syntax, without sacrificing speed, which found the sympathy of several early users amongst researchers, making it today one of the most prominent deep learning framework.

The API of PyTorch is intentionally very similar to NumPy, which is probably the most popular library in the scientific applications of Python. PyTorch's main data structure is the Tensor, which is a multidimensional array, sharing a lot of common features with NumPy arrays. The framework was designed to be extremly easy to start a new project, and roll out the first working prototype of a model. The training code is clean and easy to debug, which mainly the result of its dynamic, define-by-run execution policy. Such technique defines the operations to be evaluated or differentiated simply by running the corresponding Python code, allowing it to include control-flow statements in the training code. This is opposed to the method used by numerous other deep learning frameworks, where the users define a static computational graph, which is symbolically differentiated before execution, and then run several times. While this method spares the cost of dif-

ferentiating in every iteration, it does not allow the users to use control flow constructs or numerous other features of the library. Dynamic execution comes with another advantage, that is eager execution, enabling computations and operations to be executed upon encounter. This enables evaluation of outputs, simply by printing them to console or using break points for error checking. Executing operations immediately also allows the pipelining of CPU and GPU computations, at price of giving up high level optimization.

While PyTorch initially mainly focused on research, the features in the recent releases of the framework also enables easy transition from research or development to production, with high performance C++ runtime, leaving behind the performance bottleneck of Python. With growing number of contributors, the number of cutting-edge PyTorch-based task-specific libraries is increasing, making the work of machine learning practitioners even easier. I extensively use this library in my experiments, namely all of my models are implemented in PyTorch. I also leverage two higher level API-s from this library, which are the newly integrated Apex from mixed precision training and the PyTorch toolset for gradient checkpointing.

In the last years the generation of Volta GPUs introduced a new kind of computational unit architecture called tensor core, which provides an efficient way to perform matrix multiplication that provide 8 time greater throughput than single precision pipelines. This is speedup is possible by performing operations in half-precision format, while keeping minimal information to restore the results back to single-precision. To use half-precision or mixed-precision training mode with a model the weights have to be ported to FP16 data type where appropriate and a loss scaling has to be applied to preserve small gradient values. NVIDIA provides an easy interface to perform this in PyTorch via the apex[1] library. Its usage is also controllable by a flag in my dialogue-generation framework.

Gradient checkpointing is an optimization technique to reduce size of the allocated memory during training. Forward pass in PyTorch works by storing the results of each output node in the computation graph so they will be available during backward pass. However for models with lot of parameters this can be memory intensive. Gradient checkpointing allows to compute forward pass without storing the intermediate results of operations, thereby reducing the allocated memory. Then the values, which should have been cached are recomputed during the backward pass. This means memory is traded for compute, which can be beneficial for GPUs with lower memory.

Transformers is a library from Huggingface, which serves as a unified interface for interacting with different pre-trained transformer-based architectures. It has become one of the most prominent frameworks for NLP-based applications, as it provides an easy way of trying out state of the art techniques from various works within the same codebase.

While it has made hands-on practicioners and engineers possible to simply download and fine-tune models out of the box, due to its clean code base, it has also enabled NLP researchers and educators to study and extend the larse-scale transformer models. According to the authors, the library was built with two strong goals in mind.

One of them is simplifying usage to be as easy and fast as possible. The number of user-facing abstractions is limited, therefore learning them takes less effort. The model has three standard classes, which are required for the usage of a given transformer model: configuration, tokenizer and models. Each of these classes can be easily instantiated through a shared interface, either from a pre-trained state that is given by an URI to the from_pretrained() function or by simply calling the constructor with a given configuration object. Note that in the latter case, the framework takes care of downloading and caching

[1] https://github.com/NVIDIA/apex

the model parameters and pre-defined hyperparameters with other required assets, if they are not present on the local machine. On top of this manual model loading, the library provides a higher level abstraction for creating complete NLP pipelines, by wrapping the configuration, tokenizer and model initialization in a single pipeline object. This pipeline API provides an even faster way for serving or training models with their associated data, by simply requiring the name of an NLP problem and an optional model URI. As a drawback of this high level API, this library is not a modular toolbox of neural networks. Therefore, extending or modifying the defined algorithms requires a lot of effort and potentially duplicated behaviour. According to the authors, for this latter use case, practictioners should rely on lower level deep learning libraries such as PyTorch or Tensorflow for implementing the desired mechanism. Although the core operations in the neural network have to be defined manually, by inheriting from the aforementioned tokenizer, configuration or model classes, much of the functionalities like model loading and saving can be reused.

The other main goal of the library is to provide state of the art models with performances as close as possible to the implementation by the original authors. They host at least one version for each architecture, that closely reproduces the reported result from the creators of the said model. Since they provide a similar implementations in PyTorch and Tensorflow library versions, the model code is usually not idiomatic to the particular framework, which comes with the added benefit of easy movement and understanding between the two variations.

There are a few other main aspects, which make the libary extremely useful for my work. These include for example an easy access to the internal state of a model, such as hidden-states or attention weights. Switching between different model versions is as easy as modifying a single parameter in the tokenizer and model constructors. They enable the modification of the vocabulary and embedding layers with an unified API, therefore special tokens can be easily added for fine-tuning. Pruning trained models is also as simple as calling a single function of the corresponding model object.

In this thesis I use the Transformers library for providing the model implementations of my baseline language models. While in my case, it is necessary to modify the models provided from the library, which, as I mentioned earlier, is not supported by the API, but existing implementations serve as a good starting point for my own experiments.

Hydra is a recent open-source framework from Facebook AI Research for managing configuration files in applications. It comes particularly handy for tasks, where the different settings can easily stack up for the program. Unfortunately, this is usually the case with machine learning training scripts, which leads to hundreds of lines of argument definition code. While Python's argparse library is a great tool for this, the verbosity of creating different argument groups can quickly overgrow. A solution for this is using configuration files, which partially solves this problem, however modifying settings by repetitively opening a file and finding arbitrary parameters is cumbersome. For the sake of reproducibility it is also often necessary to save the parameters of a previous run, which is often done by duplicating the config. If the process and configuration files and their duplicates is not managed correctly, it can quickly lead to a directory full of unused and unorganized duplicates.

Hydra alleviates these problems by migrating argument definitions to a single configuration and allowing users to overwrite parameters through the command line. After parsing the provided and default parameters, the framework saves the modified config file into an organized hierarchy of directories.

This framework quickly became a staple in my workflow, since the functionality it provides allows me to quickly setup a basic set of training parameters, without the need of reinventing the wheel, as the scale and complexity of the tuneable parameters grow.

Ignite is PyTorch-based open-source machine learning framework for reducing boilerplate code in training phase. This library provides a general interface, that uses Python's decorator system for labeling methods with specific functionalities. Ignite revolves around the concept of engines, which are created by defining a callback function for handling a single step in an iteration. After starting the engine on a series of data, this callback function receives every element sequentially. It is also possible to define other callback functions, that are logically separate from the main loop. These usually involve methods like logging, model checkpointing and saving. It also provides pre-implemented convenience features like progress bar for the engine or useful techniques like early stopping for the training loop.

While Ignite provides a great structure for the project by replacing the training and validation loops with a single callback function, it tries to stay algorithm agnostic, meaning the assumptions made by the framework are fairly general. This way it can be easily adopted to a specific task with little additional work, which comes at the cost of out-of-the-box availability of advanced techniques like half-precision training or multi-gpu support.

Pytorch-Lightning is another framework for simplifying training and validation of neural networks, and in addition to the basic features of the Ignite library, half-precision, multi-gpu and even tpu can be enabled by simply setting a flag. While Ignite uses mostly callback-oriented functional interface for describing the behaviours of engines, Lightning focuses on bringing implementation of training code to standardized format via object oriented architecture.

The model and data are separated into two separate classes, which are required by the trainer. This way data is decoupled from the specific model implementation, therefore different datasets and models can be used interchangeably with each other. Compared to PyTorch Ignite, this frameworks comes with several pre-implemented techniques and algorithms, which can be turned off and on with a simple flag in the configuration. This way much of the boilerplate code can be eradicated from the experiments, which provides a much cleaner focus on the relevant methods and techniques.

Datasets is another deep learning oriented library, that focuses on preparing, storing and loading datasets in a concise and Pythonic way. Under the hood it leverages the Apache Arrow library, which is a lower level framework for storing tabular data in memory-mapped files.

Datasets provides an extensible and lightweight interface for creating or extending data for deep learning algorithms. Between the potential transformations of the said datasets, it uses caching to enhance the speed of preprocessing. While in a normal setting, without the use of advanced data loading libraries the only efficient way of feeding data to the model is by allocating examples as close to the compute unit as possible. This means the size of the corpus is limited by the operating random access memory size, which is an issue for larger training data in pre-training procedures, as without sufficient hardware they have to be stored on the disk. Datasets library provides a solution, by relying on memory-mapped files with rapid serialization techniques. This way reading and writing overheads are near or equal to RAM storage, which grants a great speed boost for systems with insufficient memory capacity.

During my experiments I extensively use this framework for storing and feeding dialogue examples to my architecture.

Open Neural Network Exchange (ONNX) is a standardized format for representing machine learning algorithms for inference, while staying agnostic to the origin framework of model the implementation. Before its creation, after training a neural network using a deep learning library, users were locked into that host framework for inference and prediction as well. Since training and serving environment might be drastically different, this is could easily limit the choice of deep learning frameworks. The most prominent library, which suffered from this problem was PyTorch, that excelled in fast prototyping and easy-to-read implementations, however it lacked essential tools for moving the trained model into production. This is particularly the case, when practitioners have access to high-performance cloud platforms with GPU clusters, while deploying the resulting model on low-powered edge devices or smartphones might not be possible due to hardware or software constraints. These problems are solved by ONNX, that uses standardized neural network operations, which is handled by the ONNX runtime for model serving.

ONNX models are created by exporting existing deep learning models from a specific library. In case of PyTorch, which was one of the first adopters of this format, this is done by essentially tracing the neural network by feeding it dummy data, so ONNX can interpret the used operations, in order to generate its computational graph. This process varies from framework to framework, however most popular libraries support it through either direct graph export like Tensorflow, or converter-based method like in PyTorch.

During inference, ONNX models are read from the Protobuf encoded tensor graph format, which are used by the ONNX runtime. It is a high-performance C++ environment, that provides support for various kinds of platform. To achieve better coverage of framework specific features without losing compatibility, defines various versions of operation sets for instructions.

I use ONNX in my work for serving the trained dialogue models through a REST interface to enable easy access for testing purposes.

4.3 Architecture

In this section, I give a detailed description of the different algorithms and techniques, which are used in the KG-enhanced dialogue transformer architecture. While the majority of required theoretical knowledge is covered in the previous chapter, I left out some minor concepts, which are addressed in the relevant parts of this section.

As a preliminary to the detailed overview, I present the high-level depiction of the proposed dialogue system architecture in figure 4.1. Each of the following subsections expand some parts of the illustration with remarks about the specific model architecture, training and implementation difficulties with corresponding solutions.

4.3.1 Data Source

The data source plays an essential role in the design process of my proposed architecture. As mentioned in previous sections of this thesis, a great limitation in this area is the lack of entity labeled conversational dataset. Although there are several available knowledge graph aligned textual corpus, hardly any of them uses a dialogue environment. The most relevant works in this area are DyKGChat method from Tuan et al. [2019] and

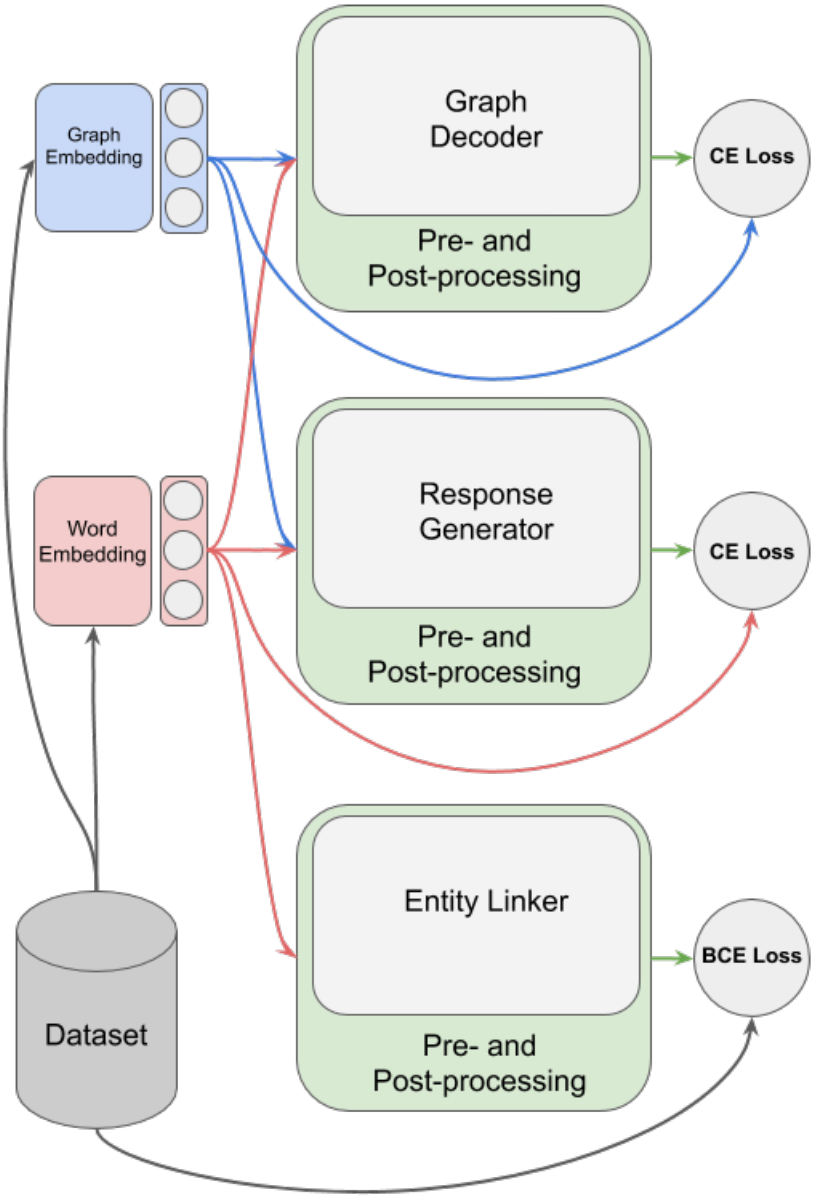


Figure 4.1: Visualization of data flow during training in the architecture components.

OpenDialKG from Moon et al. [2019]. In chapter 2 I provide a high level overview of both their contributions to the field, and I give several reasons for discrepancies between our goals. While both of them assemble a novel dialogue corpus, DyKGChat relies heavily on repeating fictional characters and locations as entities, which makes it a less desirable candidate for my experiments. On the other hand, OpenDialKG contains more natural conversations about diverse real-world topics, which serves as the main reason for choosing their provided dataset for this work. In the following paragraph, I explain the structure of this data source and discuss its shortcomings for my specific use case.

As previously discussed, the OpenDialKG provides a parallel corpus, where every utterance is associated with the mentioned triplets from the knowledge graph. The description of the graph and dialogue logs are separated into different partitions, where the latter is stored as a CSV-formatted data file. This file contains three columns: *Messages*, *User Rating* and *Assistant Rating*, of which I use only the first in my experiments. Each row of the *Messages* field is a JSON list object, that contains the dialogue turns as its entries. In figure 4.3.1 an example for such entry is shown, where the field values are used as explanations of the specific keys. During the pre-processing phase, the value of the *message* key is converted to a list of words with the english tokenizer from Spacy. Triplets from the *path* key are encoded to identifiers, which are computed from their rank in the provided entity list file.

```
{
  "type": // <str> indicating if it's a message or a KG walk action
  "sender": // <str> indicating if it is sent by "user" or "assistant"
  "message" (Optional): // <str> raw utterance,
  "metadata" (Optional): {
    "path": [
      <float> // path score,
      <list> // of KG triples (subject, relation, object),
      <str> // rendering of the path
    ]
  }
}
```

The exact use case, for which the OpenDialKG corpus was built, is training a graph walker algorithm, that uses the raw textual representation of the utterance as an auxiliary context. This structure goes against my goal, as entities are not associated with their locations in the raw text, therefore training a joint entity linker and response generator is significantly harder. The solution for this issue is explained in subsection 4.3.3.

As a final remark, I explain the specific format, which is used for the utterance and dialogue history features of my processed data. The previous visualization in 4.3.1 clearly shows that the entries of the raw data file only contain a single utterance or an action for OpenDialKg's graph decoder model, therefore a dialogue has to be aggregated from multiple examples. After this step the process expects a configuration parameter max_hist ($dialog\_len > max\_hist > 0$), which is the maximum number of utterances in the dialogue history. Each training example is then generated, by slicing the collected dialogue turns with a sliding window of the chosen $max\_hist + 1$ size. The additional 1 considers the target utterance as part of the selected entries. This way, if the number of utterances in the whole dialogue is denoted by $dialog\_len$, then there are $dialog\_len - max\_hist$ examples for each aggregation. However, according to my previous experiments, the model usually has a hard time in distinguishing the speakers of the merged utterances. In certain cases, when the dialogue history is several turns long, even larger models often disentangle

information and its not rare that they respond to their own outputs. This issue is also presented in Gu et al. [2020], where they propose special marker tokens for signal speaker change information. They leverage BERT architecture in their experiments and expand the model vocabulary with *[SP1]* and *[SP2]* values, which are inserted between each utterance of the merged dialogue turns. This way the model is provided explicit information about the speaker of the utterance, thereby avoiding the previously mentioned cases. I apply exactly the same approach, and maintain separate trainable parameter vectors for the discussed marker tokens. As a final step in the data pipeline, the collected dialogue turn fields of the examples are split into a *source* and *target* features, which serve as the inputs and expected outputs for the architecture. To utilize high-performance computational units, several data examples must be collected into a single batch, which means the word and graph embeddings in *source* and *target* are represented as a tensor with $(B \times S \times H)$, where B is the batch size, S is the sequence length and H is the dimension of the vectors. It is easy to see that the length of these sequence can vary, which may range from a single word up to several sentences, therefore padding must be applied for bringing S to a uniform value. In my implementation, each example is padded to the highest S value from the batch with a $\vec{0}$ vector with size D. Without additional measures, this dummy vector would incur false information to the model, so a mask is provided for disabling the attention operation from attending to the padded locations in the input. Still, the model has to apply the same computations to these values as they were real data from the input, which results in wasted resources. Thus it is beneficial if the batches are formed in a way, that sequences with similar lengths are paired together and the number of padding values are minimized. For this reason I implement a bucketized version of the built-in PyTorch Sampler utility class, which selects indices of examples from the dataset. The buckets boundaries are determined by the percentiles of the examples lengths at given values. This way buckets have dynamic size as well, based on the example length distribution of the dataset. This method also has to work well with the distributed input sampler module, so I have implemented bucketized sampling in a factory method, thus the same behaviour can be applied to the single- and multi GPU scenarios.

4.3.2 Embeddings

As the visualization in 4.1 shows, the first layer, that processes the raw examples from the data source is the embedding layer. There are two different modules at this stage, one of which is responsible for creating the word vector representations from the tokenized *message* values, while the other uses pre-trained graph embeddings for assigning vectors to the ids of the encoded *path* key. In the following paragraphs I present these modules with their particular architectural choices.

In section 3.5, a basic introduction can be found about different word embedding techniques. One of the mentioned concerns for implementing a mapping between words and vector representations is the sheer size of the resulting embedding matrix. A simple solution for this is also discussed in 3.5, where building on Zip's law, vocabulary truncation for a domain-specific use case can be applied without losing much information. However, the latter caveat does not hold for an open-domain setting, since the topic of conversation is not known prior to building the vocabulary. Other than high number of unknown words, misspellings are also a pressing issue in a real-world use case. In Bojanowski et al. [2016], the authors present a solution for this problem through extending the vanilla Word2Vec approach by including sub-word information as well. The name of their method is Fasttext, which has a Python interface with pre-trained word embeddings for numerous languages. In conjunction with using the basic Skip-gram objective, they maintain a different set of

parameters for aggregating word representations from a bag of character n-gram vectors. Assuming these smaller, distributed units are capable of learning useful information about complete words, their use could circumvent previously discussed issues. According to the authors, this is certainly the case, as they manage to train a robust model, which achieves state of the art results on various word representation-related tasks. For a better overview of this approach, the paper covers more technical details in the following paragraphs.

Let's consider a specific setting, where the Fasttext model is working with sub-words, consisting of three characters. As an initial pre-processing step, the given input text is separated into tri-grams, which is demonstrated for the word "tomorrow" in 4.1.

$$<\text{to}, \quad \text{tom}, \quad \text{omo}, \quad \text{mor}, \quad \text{orr}, \quad \text{row}, \quad \text{ow}> \tag{4.1}$$

As it is shown, there is an additional $<$ and $>$ character in the beginning and at the end of the word. This is a simple marker, that distinguishes between the prefix, suffix and intra-word n-grams.

The next step is converting these sub-word units to vector representations. For the purpose of reducing the required memory to store the sub-word embeddings, instead of maintaining a distinct representation for every possible tri-gram, the size is fixed to a certain K number. Sub-words are then converted to integers by a hashing function, which are brought to a range $[0, K]$ with a modulo operation. The resulting values are then used as indices for fetching the corresponding vectors from the embedding matrix. As a final step, these representations are aggregated with a simple summation. Considering an input word denoted by w, the vectors of its n-grams by z_k and a target context word by c, the score of the summed vectors is computed by a scalar product in equation 4.2.

$$s(w, c) = \sum_{k \in K_w} z_k^\top v_c \tag{4.2}$$

After the general overview of Fasttext, I proceed to the discussion of the chosen graph embedding technique. As a preliminary to this subject, the paper covers various approaches during section 3.11, which is the basis of my acquired knowledge in this field. In contrast to the word embedding layer, here I sought simplicity instead of the model best supported by theoretical results, therefore as an initial attempt I leverage TransE method for obtaining the entitiy and relation embeddings for my architecture.

Although there are several existing high-performance graph embedding libraries, like Pytorch-BigGraph from Lerer et al. [2019] or OpenKE form Han et al. [2018], they are designed to operate on much larger structures, which usually don't fit on a single system. As it is advised from the authors of the former, when the available graph contains less entities than 100000, other implementations almost always produce better results. While OpenDialKG is on the border of this limit with 100813 nodes, by including the additional benefit of deeper understanding of graph embedding methods, I proceed to building my own version of the TransE algorithm.

I follow the recipe for training the model from Bordes et al. [2013], which gives a detailed, step-by-step description. I frame my implementation in the context of the PyTorch deep learning ecosystem, with several higher-level and auxiliary libraries. I created two similar versions of this model, one without using any frameworks other than PyTorch, and one

with the integration of the PyTorch-Lightning library. The motivation behind this was to increase the throughput of the model, which in the case of the latter version was insufficient. Slower training is not an issue for OpenDialKg dataset, since it contains a relatively small knowledge graph, however for the purpose of comparing my implementation with the reported results from the original implementation, I involved FB15K corpus in my experiments as well. By using the faster implementation, the training speed increases by a factor of 1.5, which is considerable when performing hyperparameter optimization. In chapter 5 the paper goes into more detail about the results with different parameter settings and their corresponding results. As a general overview of the overall training algorithm, I present the description from Bordes et al. [2013] in 4.2.

Algorithm 1 Learning TransE

input Training set $S = \{(h, \ell, t)\}$, entities and rel. sets E and L, margin γ, embeddings dim. k.

initialize $\boldsymbol{\ell} \leftarrow \text{uniform}(-\frac{6}{\sqrt{k}}, \frac{6}{\sqrt{k}})$ for each $\ell \in L$

  $\boldsymbol{\ell} \leftarrow \boldsymbol{\ell} / \|\boldsymbol{\ell}\|$ for each $\ell \in L$

  $\mathbf{e} \leftarrow \text{uniform}(-\frac{6}{\sqrt{k}}, \frac{6}{\sqrt{k}})$ for each entity $e \in E$

loop

  $\mathbf{e} \leftarrow \mathbf{e} / \|\mathbf{e}\|$ for each entity $e \in E$

  $S_{batch} \leftarrow$ sample(S, b) // sample a minibatch of size b

  $T_{batch} \leftarrow \emptyset$ // initialize the set of pairs of triplets

  for $(h, \ell, t) \in S_{batch}$ **do**

    $(h', \ell, t') \leftarrow$ sample$(S'_{(h,\ell,t)})$ // sample a corrupted triplet

    $T_{batch} \leftarrow T_{batch} \cup \{((h, \ell, t), (h', \ell, t'))\}$

  end for

  Update embeddings w.r.t. $\sum_{((h,\ell,t),(h',\ell,t')) \in T_{batch}} \nabla \left[\gamma + d(\boldsymbol{h} + \boldsymbol{\ell}, \boldsymbol{t}) - d(\boldsymbol{h'} + \boldsymbol{\ell}, \boldsymbol{t'})\right]_+$

end loop

Figure 4.2: Algorithm of learning TransE embeddings from Bordes et al. [2013].

After several training sessions, the parameter state of the best performing model is then exported to a file, which provides the entity and relation embeddings for each of the following components of the architecture.

4.3.3 Response Generator

The backbone of the knowledge graph-enhanced dialauge transformer architecture is the response generator component. The main role of this model is to generate an output text, which is conditioned on the results of the graph and word embedding layers. It is based on the vanilla transformer from Vaswani et al. [2017], which is discussed with more detail in section 3.10. The main difference is the incorporation of an additional multi-head attention module, in both the encoder and decoder layers as well. Its role is to aggregate information from the provided source and target graph node embeddings, which in theory should help the model in generating more factually correct responses. In relevant literature like Sun et al. [2018b], this is considered an early-fusion approach, which stands for merging the external representations with the input of the neural network. The counterpart of this method is late-fusion, which expands the hidden state of the model with external information, just before computing the output representation with a final projection layer. By adapting a modified version of KGLM approach from IV et al. [2019a], late-fusion could technically be integrated into my model architecture as well. The main issue is caused by the lack tightly aligned parallel corpus, where in addition specified

triplets in the dialogue, the mentioned entities or relations are explicitly marked in the utterance as well. This is missing from the OpenDialKG dataset, therefore the generator cannot be trained to render entities as its output during time steps. In the following paragraphs I walk through each layer of the response generator, for the purpose of covering any modification, which I have applied to the basic Transformer model.

As it is shown in 3.10, both the encoder and decoder modules accept their inputs through a position encoding layer, which purpose is covered in section 3.10.4. During my experiments, I have tried out several of the discussed position encoding techniques, one of which is the learned relative embedding. Although its theoretical benefits are promising, application of this method would heavily restrict the possibilities of using pre-trained transformer weights in the response generator module. This assumption comes from the fact, that relative position embeddings require a modified version of the general attention module, which is used by most transformer-based language model, such as BERT or GPT-2. In light of this issue, I opted for a simple learned position embedding layer, which is compatible with most of the mentioned pre-trained methods.

By going forward in the architecture, the following module is the encoder layer. After the incorporation of the position information to the word embedding vectors, the model applies a multi-head attention operation to the graph embeddings. For recalling the details of this mechanism, I refer back to section 3.10.2, where I give an in-depth summary about this subject. Considering the discussed notations for this computation, in case of this specific layer Q, V are the graph node embeddings from the dialogue history and the K vectors are the *source* word embeddings. The resulting representations from this computation are then merged with its input values with a residual connection, which is followed by a layer normalization operation. The main intuition behind this whole procedure, is to incline the model to align the word and graph embeddings, which might increase its emphasis on relevant entities from the *source* word embedding inputs. Subsequent operations are all identical to the encoder layer of the vanilla transformer model, therefore I proceed to the decoder.

Most of the important changes, which are applied to the decoder module are already presented in the previous paragraphs, as the same additional attention layer is used here as well. The main difference is the input of this module, as instead of using the *source*, it relies on the *target* word embeddings with graph entities from the response utterance. Similarly to the encoder, after this operation the method is identical to the decoder part of the vanilla transformer, with the exception of the output layer. Before proceeding to the explanation of this exception, I introduce the two possible implementations for it.

In the general case, the decoder serves as an autoregressive language model, which computes the representations for the target sequence. The most common way of extracting the predicted word from these latent vectors is to project them with a matrix of size $h \times K$, where h is the dimension of the vector and K is the number of entries in the vocabulary. By applying a softmax function at the end of this operation, the resulting output is a probability distribution over the possible predictions. For clarification, the performed computation is shown in equation 4.3, where v_i and p_i is the i^{th} element of the K dimensional input and output vectors.

$$p_i = \frac{e^{v_i}}{\sum_{k=1}^{K} e^{v_k}} \tag{4.3}$$

These distributions can then be used for various decoding or sampling algorithms, which the paper covers in more detail in section 4.3.3. However, there is an issue with this

approach, which is discussed in Li et al. [2019]. The authors emphasize that by computing 4.3 in cases where the value of K is high, there is a great overhead both in speed and memory, simply because of the size of the projection matrix and the softmax function. Their solution to this problem is to abandon these two operations and forcing the model to output continuous word vector representations. The error signal for training is then computed by comparing these vectors to pre-trained word embeddings with some kind of distance measure, like L2 distance or cosine similarity. An improvement of this technique is introduced in Kumar and Tsvetkov [2019], where they propose the usage of novel *von Mises-Fisher* loss, which is a probabilistic variant of cosine loss.

Building on the previously described benefits of continuous outputs, I leverage it in my architecture as well. I apply a negative sampling strategy for the training procedure, which is similar to the mentioned operation used in Skip-gram. According to Mikolov et al. [2013b], they compute minimized error term with equation 4.4, where σ is the sigmoid function from 3.3. w denotes the vector representation of the word and k is the number of negative examples, which are sampled from a probability distribution $P_n(w)$.

$$\log \sigma \left(v'_{w_O} v^\top_{w_I} \right) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)} \left[\log \sigma \left(-v'_{w_i} v^\top_{w_I} \right) \right] \tag{4.4}$$

In contrast to this, I consider a more generalized approach, that is used in StarSpace from Wu et al. [2017]. I follow their formula for computing the error signal for my model, which is shown in equation 4.5.

$$\sum_{\substack{(w_I, w_O) \in E^+ \\ w_k^- \in E^-}} L^{batch} \left(\text{sim}(w_I, w_O), \text{sim} \left(w_I, w_1^- \right), \ldots, \text{sim} \left(w_I, w_k^- \right) \right) \tag{4.5}$$

The former method from Mikolov et al. [2013b] is actually a special case of this error term, where they differentiate between noise and real target, by using logistic regression. In addition to this, the authors of StarSpace mention two variations for both the similarity measure sim and loss function L. For sim, they either use inner product or cosine similarity, while for L they experiment with margin ranking loss and negative log of the softmax. According to them, there are no significant differences between these approaches, therefore I stick with the choice of dot product similarity with the log softmax loss function.

As mentioned earlier, circumventing the manifestation of the $(H \times k)$ sized projection matrix provides a considerable speed boost, and by training the model to output vector representations in pre-trained semantic embedding space, the results and possible mistakes are much more interpretable. In addition, this approach enables the easy adaptation of the modified KGLM technique into my architecture. In conjunction with the previously discussed early-fusion through the multi-head attention mechanism, late-fusion could be applied by measuring the similarity of the output representation with the graph embedding of the mentioned entity in the expected response. During testing, this would result in an additional concatenation of the potential graph entity vectors and the embeddings of the standard vocabulary. The best fit output for a given time step is then sampled from this unified matrix. This idea is heavily inspired by the authors of KGLM, however its applicability is not possible with the currently used dataset, as entity mentions are not labeled in the utterance. I leave the further investigation of this issue to future work.

In the following paragraphs, The paper covers an important subject regarding the process of sampling high-quality utterances from the output distribution of response generator

model. The specific decoding algorithm plays an extremely important role in language models, therefore extensive research has already been done in the field. According to Holtzman et al. [2019] decoding strategies that optimize for the output with highest probability like the simple greedy decoding or even beam search, leads to text that is incredibly low-quality, even when using state-of-the-art models such as GPT-2. This is not intuitive, as it would be expected that such language model should assign high probabilities to the most human-like text. Upon analyzation of human and beam-search generated text the key finding is that human text is more surprising and this greater variance in words is what gives rich character to human text. For the purpose of presenting the main idea behind the most popular methods, the paper goes over each of the decoding algorithm, which I leverage for my experiments.

Greedy decoding is the simplest approach, which calculates the predicted words based on which has the highest score in the probability distribution, that is computed by the final softmax operation of the model. According to my experiments, this method leads to the worst results, as especially for smaller models the responses are repetitive and dull.

Beam Search is a more sophisticated method, since in greedy search the algorithm always takes the most likely output at each time step and move on, whereas beam search considers several alternative candidates. These alternative generation paths are called beams and their number is determined by the beam width hyperparameter, denoted by k. The final path is then chosen, based on the summarized conditional probability of the whole sequence. While as previously mentioned, Holtzman et al. [2019] shows that beam search is inferior to their proposed probabilistic sampling method, Roller et al. [2020] demonstrates that by training good multi-modal dialogue generator models, using beam search provides better results compared to other methods.

Top-k Sampling is used by GPT-2 for increasing diversity in the generated text. This method introduces randomness by selecting an output at each time step from the top k best candidates according to their softmax score values.

Nucleus decoding is from the authors of Holtzman et al. [2019], which can be considered an improved version of top-k. Instead of considering the top k most probable token nucleus decoding selects from the top p portion of the probability mass, which gives a dynamically changing window for the candidate tokens.

4.3.4 Entity Linker

The job of this module is to extract mentioned entities from the utterance and align them with nodes from the knowledge graph. Generally this is considered a complex problem in itself, as it usually requires the usage of several independent sub-models. The usual linking pipeline starts with a named entity recognizer module, which finds the potential surface forms of the mentioned graph nodes. After preforming disambiguation and coreference resolution for narrowing down or extending the potential matches, a separate model classifies each of the found surface forms and decides whether they are a specific node from the graph or not. The currently state-of-the-art system for this problem is from Mulang' et al. [2020], where they leverage pre-trained bidirectional transformer models on the Wikidata-Disamb dataset from Cetoli et al. [2019].

As there is no readily available linking model for my use case, the implementation or even the adaptation of the previously outlined system is out of scope for my thesis. Therefore, I have to make a compromise in the complexity of my entity linking method. Another limiting factor is the previously mentioned shortcoming of OpenDialKG dataset, as the

mentioned triplets for a given utterance are not labeled in the raw messages. This issue makes the training of such model even harder, since it has to implicitly learn each of the previously mentioned steps of an entity linking system. With the previously mentioned difficulties in mind, I proceed to the discussion of my entity linking algorithm. I leverage a pre-trained BERT model, which receives a tokenized version of *Message* field, which is introduced in section 4.3.1. However, the tokenization for BERT is not done with Spacy, as its pre-trained embeddings were created for WordPiece vocabulary. The paper covers the exact details of this technique in section 3.10.5. In conjunction with the ids of the input tokens, I pass a special *[CLS]* marker, which serves as an aggregation node for the remaining parts of the input. This is a common technique for fine-tuning the model on sequence classification tasks, as the final representation of *[CLS]* serves as a pooled output for the whole sequence. After transforming this vector with a $H \times G$ matrix, where H is the hidden size of BERT and G is the number of nodes in the graph, I apply a sigmoid function 3.3 element-wise to obtain probability scores for each entity of the knowledge graph. The objective is then to optimize the binary cross-entropy loss of these outputs, which basically frames it as a multi-label classification problem. The true values for the loss function are provided by the corresponding ids of the present entities in the *source* field. Further details of the results and training parameters for this component are discussed in chapter 5.

4.3.5 Graph Decoder

The main role of this component of the architecture is to provide a set of entities for the decoder part of the response generator model, which can be potentially included in the output utterance. During training, the relevant entities are already presented to the model from the dataset, however, when testing, these properties must be calculated from the input utterance.

The simplest solution for this problem is to rely solely on the entity linker component, and automatically collect every vertex from the knowledge graph, which are connected to the found entities. While this method serves as quick proof-of-concept solution for providing potentially relevant knowledge graph nodes for the decoder, it would induce disparity between training and testing data. This is because the provided node representations in the training data are actually present in the corresponding utterance, therefore the model is optimized to leverage them in the generated response. This would not be the case for the proposed method, therefore I proceed to the implementation of a more sophisticated system.

In Moon et al. [2019], the authors propose a graph decoder algorithm, which in the absence of their implementation I try to reproduce with several simplifications. They introduce the DialKG Walker algorithm, which returns a set of entities by providing it multiple features of the given dialogue context. The features consist of a set of knowledge graph entities in conjunction with the corresponding utterance at the current time step and a history of previous dialogue turns. These inputs are then provided to an autoregressive neural network through an attention layer for computing a walk path from each initial graph node. The covered vertices are then combined with a zero-shot relevance score, which is computed by the application of a learned transformation function to the entity candidates.

My approach is almost identical to this technique, but unlike them I don't use a zero-shot relevancy score. Instead I consider each of the covered entities as the final result of the algorithm. The autoregressive model is an LSTM network from 3.7.2, which uses the same context features as DialKG Walker. A slight difference is that the dialogue history and the last utterance is merged in the *source* key of the data example, which is leveraged by the

LSTM model through a single attention layer. Entities contained in these utterances are handled the same way. The sequence of the autoregressive model starts with the initial graph node, and outputs a probability distribution over the valid relation scores at each time step. Both context features are merged into the hidden state of the LSTM in a late-fusion manner, just before computing the previously mentioned output probabilities. The model is trained with a cross-entropy loss function. Similarly to earlier components, the results of this approach and practical details are discussed in chapter 5.

4.4 Baselines

In the following sections I am going to present the baseline models for evaluating the performance of different dialogue generator model architectures. Each architecture is transformer based, thereby maintaining a separate section for it in 4.4.1 among other baselines might be a bit confusing. The reason for this is that it actually stands for a specific type of transformer implementation. By the work of Devlin et al. [2018] and Radford et al. [2019] several variations branched off from the original Vaswani et al. [2017] model which is discussed in section 3.10.5. In 4.4.1 the paper goes into more details about the specifics of the computational operations present in the original Transformer.

4.4.1 Transformer

There are two types of models in my experiments that rely solely on the original architecture and the only distinction between them is the type of their output representation. Most of this subject is covered in section 4.3.3, where I present my response generator module, which produces continuous output representations. I consider two baseline models, one of which is identical to the response generator module, but without using the attention layers over the entity embeddings. In the experiments chapter, I refer to this variant as *Transformer-Base-Continuous*, while the other model is *Transformer-Base-Discrete*, which is the same model with discrete output representation. Data is provided in the same manner for these models, as described in 4.3.1, without including the knowledge graph related features. Since the discrete version requires a specified set of words as a vocabulary, I assemble it from the most common 50000 words from the dataset. Each element is then paired with a corresponding index of the vocab, which are used to represent the target words during training. This stands in contrast to the continuous method, where the vectors themselves are provided in the *target* key.

4.4.2 GPT-2

Another baseline is the pre-trained *GPT-2* language model, which is discussed in more detail in section 3.10.5. The data processing for *GPT-2* is considerably different than than the procedure for previously discussed Transformers, as it comes with pre-trained weights with a byte-pair-encoding tokenization algorithm. Byte pair encoding for natural language processing is first presented in Sennrich et al. [2016], which is originally a universal compression method. Various versions of this approach is used in pre-trained language models, as it enables the algorithm to learn a representation for any kind of word with a fixed vocabulary, through encoding words with smaller sub-units. By employing the pre-trained encoder, I convert every message of the dialogues to sub-word ids, which are then aggregated into a single sequence and passed to the *input_ids* argument of the model. In previous sections regarding data pre-processing, the paper covers the usage of speaker

identifier tokens, which are present in this baseline implementation as well. However, in conjunction with the single token between utterances, I leverage the *token_type_id* input of the pre-trained *GPT-2* model. This way, by providing another sequential input matrix to the model with the same dimension as the matrix of the sub-word ids, token type information can be added to the input word embeddings. In my case, this means the creation of a matrix, which is populated by *[SP1]* and *[SP2]*, where the particular value in the *token_type_id* matrix corresponds to the speaker of the sub-word in the original input matrix. This is demonstrated by an example conversation 4.4.2 and its pre-processing result in table 4.4.2.

- How are you?
- Fine thanks.

[SP1]	How	are	you	?	[SP2]	Fine	thank	#s	.
[SP1]	[SP1]	[SP1]	[SP1]	[SP1]	[SP2]	[SP2]	[SP2]	[SP2]	[SP2]

During text generation, the model is then provided with the initial speaker token, which according to my experiments, reduces the potential disentanglement regarding the speakers. Additionally, the model receives an *attention_mask*, which serves the same purpose as the mask discussed in section 4.3.1. Considering the model is trained with mini-batches of size B, each of the mentioned inputs are of size $(B \times S)$, where S is the length of the longest sequence in the batch.

4.4.3 XLNet

Similarly to *GPT-2*, *XLNet* is used in my experiments as an autoregressive language model. Due to its permutational computation order, the inner operations differ from the latter, however most of the pre-processing steps can be reused from *GPT-2*. This can mainly be credited to the Transformers library, which provides an universal interface for every pre-trained Transformer, that enables effortless switching between different model types. In contrast to *GPT-2*, *XLNet* uses SentencePiece tokenization algorithm, which is discussed in conjunction with the model itself in section 3.10.5. In addition to the previously mentioned *input_ids*, *token_type_id attention_mask*, the model expects several other input tensors, which are required for the two-stream self attention operation. One of them is the *permutation_mask*, which is a tensor with 3 dimensions $(B \times S \times S)$. It is populated with boolean values in the following way: the value of the permutation mask b, i, j marks whether the value at the i^{th} index of the b^{th} sequence of the *input_ids* matrix should attend to the value at the j^{th} index of that sequence in terms of attention computation. According to this scheme for the simple single next token prediction task this tensor is *True* everywhere except at padding locations of the input. The last input tensor is the *target_mapping*, that marks the location and order of the expected output. It has three dimensions of size $(B \times S \times S)$ with boolean values, where $target\_mapping[b, i, j] = True$ marks that the i^{th} prediction of the b^{th} example in the batch is on the j^{th} position of the *input_ids*.

Chapter 5

Experiments

In this chapter I discuss the results and settings of my performed experiments with various models. In the first sections, the paper covers the details of the used datasets and introduce the evaluated metrics.

5.1 Datasets

This section presents several data sources, which are used in my framework for training various components. Almost all of them are conversational datasets, which are either used for training my baseline dialogue agents or my main knowledge graph-enhanced architecture.

5.1.1 OpenDialKG

While OpenDialKG dataset from Moon et al. [2019] is thoroughly described in section 4.3.1, here, the paper introduces the technical parameters and specifics of the assembled conversation subjects. The corpus consists of conversations between two agents about various topics with a total of 91000 dialogue turns across 15000 sessions. As discussed earlier, it is a parallel dataset, where each utterance is paired with the corresponding entities and relations that are mentioned in the text. These mentions are organized into walks over the knowledge graph, which is essential for the training of their proposed DialKG Walker algorithm. Each dialogue is collected by crow-workers, where the first agent is provided with an initial entity from the knowledge graph and his or her job is to engage in a conversation about that entity. The second person is then given various facts about this subject, and he or she is asked to use them in the response. The provided facts are also part of the knowledge graph and are either 1 or 2-hop away from the original entity. After each message, several new entities are given for the agents, thereby the whole process repeats itself until the end of the conversation. There are two main tasks in the dataset, one of which is recommendation, where the second agent serves as an assistant, that provides information about movies or books to the user. The other task is an simply an open-domain conversation about a specific subject.

5.1.2 Persona Chat

Persona chat is a conversational dataset from Zhang et al. [2018]. The authors aim is to provide a more engaging and personal open-domain dialogue corpus. It is crowd-sourced from Amazon Mechanical Turk, where two agents converse about an arbitrary topic. Each of the participants are assigned a personality profile at the beginning of a session, according to which the agents have to express their feeling and interests in certain subjects. It consists of 1155 different personas, where each is defined by 5 sentences at least. The data is separated into train, validation and test set, where the latter two has 100-100 profiles, which are not present in the train split. The dataset has an overall 162000 utterances across 10900 dialogues. Out of these examples, 1000 dialogue with approximately 15000 utterances are set aside for the validation and another slice with the same amount for the test split respectively. During my experiments, only the utterances are used by the models, which are pre-processed in the same way as mentioned in 4.3.1. Since the raw format of Persona Chat differs from OpenDialKG, as an initial step I convert it to the same structure, therefore all of the processing operations can be reused for this corpus as well.

5.1.3 Topical Chat

Topical Chat was introduced by Gopalakrishnan et al. [2019], which is a knowledge-grounded conversational corpus, that spans across 8 topics of discourse, however contrary to the previously discussed datasets, participants don't have pre-defined roles during the dialogue. They assemble the dataset through workers on Amazon Mechanical Turk, where each agent is provided a topical reading set with a corresponding entity. There are two workers in every session, who are asked to engage in a natural and coherent discussion over their provided knowledge. Unlike in OpenDialKG and several other knowledge grounded datasets, the given content for the two agents is symmetric or yield slight asymmetry, where a setting is considered symmetric if both workers are provided with the same information and asymmetric otherwise. According to the authors, this is a generalization of the previously discussed user-assistant approach in OpenDialKG, which aims to more precisely reflect real-world conversations. This way, dialogue models can leverage versatile and realistic utterances, from the perspective of both agents. The quality of the collected exchanges are ensured by a scoring system, where dialogue partners are asked to annotate the sentiment of their response, as well as provide a score on a scale from 1 to 5 to grade the other agent's response. The agents

The collected data is separated into a train, validation and test split, where the latter two are further sliced into frequent and rare versions. The difference between the two is that frequent sets contain entities, which are regularly seen during training, while the entities of rare set are infrequent in the training data. During my experiments I disregard all information from the corpus, except the utterances in the dialogue. Both frequent and rare evaluation splits are merged, which results in a training split with 188378 utterances across 8628 conversations, validation split with 1078 conversations across 23373 utterances and test split with 23550 utterances across 1078 conversations. The transformations applied to the raw Persona Chat corpus are present for this dataset as well.

5.1.4 Daily Dialog

Daily Dialog from Li et al. [2017] is a multi-turn dialog dataset, that aims to resemble everyday conversations between two agents. It is crawled from various websites, which are used by English learners to practice the language in daily life. Utterances and word usage is usually informal and often focus on a specific subject. Each dialogue approximately consists of 2-3 topics across 8 turns, which proves to be suitable for training dialogue generator models. After removing duplicate conversations, the dataset contains 13118 multi-turn dialogues.

5.1.5 FB15k

As an exception to the previously discussed datasets, FB15k from Bordes et al. [2013] is not a conversational dataset, but serves as a way to benchmark my implementation of the TransE algorithm. It is collected from Freebase, which a giant knowledge base of general facts, that currently contains more than 1.2 bullion triplets with over 80 million entities. FB15k is a small subset of this, with 592213 triplets, containing 14951 entities and 1345 relationships, which are randomly distributed among training, validation and test splits.

5.2 Metrics

Finding an automatic metric of evaluating the outputs of dialogue generator agents, which correlates well with human judgement has proven to be a difficult problem. This topic is the subject of extensive research, as leveraging human labour for grading the quality of generated dialogues is slow and expensive. While there are a collection of metrics, that are regularly used in machine translation or abstractive summarization which can be useful for this issue as well, there is not a single widely accepted measure for dialogue generation. Therefore, I rely mostly on empirical evaluation when rating my models, but I also calculate some automatic metrics, which I present in the following paragraphs.

Response length is on of the simplest measures, which shows the number of words in the model response. While this value largely depends on the provided source utterance, averaging this metric over the whole test dataset can give a fair insight into the difference between human written and machine generated responses.

Word Mover's Distance from Kusner et al. [2015] is a distance function between documents. It measures the dissimilarity between two texts, by computing the minimum distance that the word embeddings of one text needs to move to reach the word embeddings of the other text. This metric is the special case of Earth Mover's Distance, which is a transportation problem with various efficient algorithmic solutions. The word embeddings in my case are provided by the discussed Fasttext library, and computation of the distance is performed by the Gensim framework.

BLEU is a widely used metric in machine translation, as it is claimed to have a high correlation with human judgement. It computes a modified version of precision, which is often used in classification problems for measuring the fraction of true positive elements among all positively classified elements. In BLEU, precision is used for computing the n-gram overlap between the predicted and reference text. I use the implementation of the NLTK python library, by examining only the mutual 1-gram and 2-gram occurrences. I also apply a weight for each of these, namely 0.6 for the 1-gram and 0.4 for the 2-gram values.

5.3 Results

The training is carried out on the Google Colaboratory platform, which provides free access to high-performance GPUs with certain time limits. It also comes with a complete Python machine learning environment, which can be leveraged through a jupyter notebook-like interface. Colaboratory enables connection to Google Drive, therefore I use it for storing datasets and the output files of experiments, such as model parameters and configurations. Scripts for training a given model are placed in the local memory of the Colaboratory runtime, which are executed with shell commands in the notebook cells. The provided GPU is different for almost every training session, but most of the time it is an NVIDIA Tesla P100 instance or a Tesla T4. Both of these cards enable mixed precision training, which can speed up iterations through increased batch size by lowering the memory cost of computations.

5.3.1 TransE

The first experiment concerning the dialogue transformer architecture is creating the knowledge graph embeddings for the model. As described in previous sections, I rely on my own implementation of TransE algorithm by strictly following training recipes and hyperparameters presented by the original authors. For validating my model, I perform training using their recommended settings on FB15k dataset, with latent dimension $k = 50$, learning rate of SGD optimizer $\lambda = 0.01$, margin parameter $\mu = 1$ and distance metric $d = L_1$. In the original work, the authors report several metrics, out of which I only use the hits@10 value, which measures, whether the correct prediction is in the top 10 candidates produced by the model. Each run is terminated with an early stopping criterion, with a maximum epochs of 1000. Across multiple sessions with different random seeds, my implementation reaches a mean value of 33.34 with a deviation of 0.75. This value resembles the results in Bordes et al. [2013] with a hits@10 value of 34.9, therefore I proceed to training the model on the OpenDialKG dataset. I traverse the same set of hyperparameters as presented in the original work, where the different settings are compared through the mean hits@10 value of 3-fold cross-validations. Based on the final results, the best performing hyperparameters are $k = 50$, $\lambda = 0.01$, $\mu = 1$, $d = L_2$. With various random seeds, this setting achieves a mean hits@10 value of 23.1 with a deviation of 0.92.

5.3.2 GPT-2

There are several pre-trained versions of the GPT-2 model, which are available through the Huggingface Transformers framework. The difference between them is the number and dimension of hidden layers, which are small (124 million parameters), medium (355 million parameters), large (774 million parameters) and extra-large (1.5 billion parameters) sized models. As the memory requirement for larger models is too great for single GPU training, I resort to using the small and medium models for my experiments. I train them on multiple datasets: DailyDialog, PersonaChat, TopicalChat and OpenDialKG in a multi-turn setting with a history size of 4. In my case this means that each training example is generated by taking a maximum of 4 previous utterances into the input sequence as context. I try various learning rate λ and maximum token per batch *mtpb* values for each corpus, and according to my experience a $mtpb = 1536$ with an initial $\lambda = 0.001$ for scheduled learning rate yields uniformly the best performance. In the initial 2 epochs of training, I perform warmup for λ value until $\lambda = 0.008$, which is followed by linear decay until $\lambda = 0.0001$. The optimizer is the Adam method from Kingma and Ba [2017]

and each session is run until 10 epochs. During training I follow the cross entropy loss and per-token accuracy values for the model, which interestingly tend to degrade for the validation split after a few epochs. For the small model this means a hit@1 value of $\sim 70\%$ on the PersonaChat training set and $\sim 40\%$ value on its validation set. While in typical machine learning problems, this phenomena indicates overfitting, I observe better quality responses with sophisticated sampling methods from overfitted GPT-2 models. For this reason, I do not report further training-related metric results, instead I proceed to the outputs of the automatic evaluation methods from section 5.2, which are presented in table 5.1. Each value is obtained by training the model on the discussed datasets, and then evaluating GPT-2 on the held-out test split with the presented sampling techniques. The results are then aggregated by computing the mean values across each dataset.

5.3.3 XLNet

Training hyperparameters and settings for the XLNet model are identical to the previously discussed GPT-2 model. It is also implemented in the Huggingface Transformers framework, with two different model versions. These are the base XLNet model with 110 million trainable parameters and the large model with 340 million parameters. I employ the same datasets and sampling algorithms as discussed in the previous section. By performing the automatic evaluation, the resulting metric values for the two model versions can be seen in table 5.1.

5.3.4 Transformer

Implementing and evaluating an untrained vanilla Transformer architecture on the same set of conversational datasets as the previously discussed two approaches is inevitable for framing my implementation in the context of pre-trained language models. The basis of this architecture is provided by the PyTorch framework, which I employ in two different versions. As discussed in 4.4.1, in conjunction with the vanilla *Transformer-Base-Discrete* model, I maintain the *Transformer-Base-Continuous* version, which uses the same continuous output mechanism as presented in the knowledge graph-enhanced architecture. I also use identical architectural hyperparameters for each of the untrained Transformer models, where the hidden dimension of the model is $h = 256$, number of layers in the encoder is $L_e = 2$ and layers in the decoder is $L_d = 6$. Each self- and multi-head attention mechanism uses $n_{heads} = 8$ for the computations. Similarly to the previously discussed models, I use Adam optimizer, however I do not apply learning rate scheduling. Each model is trained for 20 epochs without early stopping. Due to the smaller parameter count than the pre-trained models, it is possible to maintain a larger *mtpb* value, which is set to 2560. In case of the continuous model, I only apply nearest neighborhood search finding the predicted words during response sampling, while in the case of the *Transformer-Base-Discrete* I try out the same decoding approach as presented for GPT-2. For the *Transformer-Base-Discrete* model, the vocabulary is constructed from the most common 50,000 words of the training dataset, while the continuous version uses the full vector space of the 2,000,000 Fasttext words in addition with the sub-word representation of every token from the training dataset, which are not present in Fasttext. The result of the final evaluation, averaged across each dataset can be seen in table 5.1.

Model	Decoding	Δ Length		WMD		BLEU	
		Mean	Std	Mean	Std	Mean	Std
GPT-2 (small)	Greedy decoding	9.8	3.7	1.9	0.3	0.2	0.1
	Beam Search $k = 3$	11.4	4.3	1.6	0.4	0.1	0.0
	Beam Search $k = 5$	10.2	4.2	**1.3**	**0.3**	0.2	0.1
	Top-k Sampling $k = 100$	**16.4**	**4.6**	1.5	0.6	0.3	0.2
	Nucleus decoding $p = 0.1$	15.3	3.5	1.4	0.5	**0.3**	**0.1**
XLNet (base)	Greedy decoding	10.0	3.4	1.8	0.3	0.2	0.1
	Beam Search $k = 3$	13.2	5.0	1.7	0.3	0.1	0.0
	Beam Search $k = 5$	14.1	3.6	1.5	0.4	**0.3**	**0.1**
	Top-k Sampling $k = 100$	16.2	4.1	1.4	0.7	0.2	0.1
	Nucleus decoding $p = 0.1$	16.1	3.2	1.5	0.7	0.3	0.2
Transformer	Greedy decoding	6.7	2.9	2.2	0.6	0.0	0.0
	Beam Search $k = 3$	9.1	2.5	1.9	0.4	0.1	0.0
	Beam Search $k = 5$	9.9	3.1	2.0	0.4	0.1	0.0
	Top-k Sampling $k = 100$	11.0	6.7	2.2	0.5	0.1	0.0
	Nucleus decoding $p = 0.1$	12.5	5.9	2.1	0.5	0.1	0.0
Continuous Transformer	Similarity Search	11.8	3.7	1.7	0.4	0.2	0.1

Table 5.1: Comparison between the mean evaluation scores across every presented dataset for the baseline models.

5.3.5 KG Dialogue Transformer

After obtaining the graph node representations from the TransE model and establishing the baseline language models, I proceed to training the knowledge graph-enhanced dialogue transformer architecture. Hyperparameters are identical to the previously discussed *Transformer-Base-Continuous*, therefore I do not cover them in this subsection. Contrary to previous models, training this architecture is only possible on the OpenDialKG dataset, as alignment with a knowledge graph is not present in the other dialogue datasets.

Before training the response generator component of the architecture, I create the entity linker and graph decoder components. This comes with the benefit of using them actively during the training of response generator, such that occasionally the input source and target entity nodes are sampled from these models, instead of the original examples. Theoretically this technique helps the model to adapt to the input distribution, which is present during testing. For the entity linker, I use the bert-base-cased version of the BERT model, which has approximately 110 million parameters. I train it with a batch size of 64 until a maximum of 50 epochs or until the early stopping criteria. I apply the Adam optimizer with the same learning rate scheduling as presented for GPT-2. During training, I track the mean precision, recall and F1 scores of the individual binary classes, in conjunction with the Jaccard similarity between the set of ground truth and retrieved entities. The best model is trained for 42 epochs and reaches a $precision = 0.23$, $recall = 0.19$, $f1 = 0.21$ and $J = 0.11$. For the rest of this paragraph, I proceed to the introduction of training details concerning the graph decoder module. As discussed in section 4.3.5, it is a recurrent neural network, more specifically an LSTM model. Its only architectural hyperparameter is the hidden dimension, which I set to 256. I train it with a batch size of 256 with a cross-entropy loss function and Adam optimizer. During training, I track the hits@1, hits@10, hits@50 scores and the Jaccard similarity between the ground truth

and retrieved entity sets. The best model is reached at epoch 24 with a $hits@1 = 0.11$, $hits@10 = 0.37$, $hits@10 = 0.67$ and $J = 0.12$.

The final milestone is obtaining the response generator module for the architecture. As mentioned previously, architectural and training related hyperparameters are taken from the continuous baseline transformer model. Like the other untrained models, it also trained for 20 epochs, for which the final results are visualized in table 5.2. In addition to the metrics for this model, the evaluation results of baseline language models are presented as well, which in this case are exclusively computed on OpenDialKG dialogues. As discussed in earlier sections, each model is trained with a maximum history context of 4.

Model	Decoding	Δ Length		WMD		BLEU	
		Mean	Std	Mean	Std	Mean	Std
GPT-2 (small)	Greedy decoding	10.2	3.5	2.0	0.4	0.1	0.0
	Beam Search $k = 3$	12.1	3.6	1.7	0.5	0.1	0.0
	Beam Search $k = 5$	12.0	3.5	**1.6**	**0.4**	0.1	0.0
	Top-k Sampling $k = 100$	14.2	3.8	1.6	0.6	0.1	0.0
	Nucleus decoding $p = 0.1$	12.1	3.9	1.7	0.6	**0.2**	**0.1**
XLNet (base)	Greedy decoding	9.4	3.4	2.2	0.3	0.1	0.0
	Beam Search $k = 3$	11.2	3.2	1.9	0.4	0.1	0.0
	Beam Search $k = 5$	12.9	3.3	1.8	0.3	**0.2**	**0.1**
	Top-k Sampling $k = 100$	**14.3**	**4.3**	1.9	0.6	0.1	0.0
	Nucleus decoding $p = 0.1$	14.1	4.5	1.8	0.5	0.1	0.0
Transformer	Greedy decoding	7.1	2.2	2.3	0.2	0.0	0.0
	Beam Search $k = 3$	8.5	2.0	2.3	0.4	0.0	0.0
	Beam Search $k = 5$	9.4	2.8	2.2	0.4	0.0	0.0
	Top-k Sampling $k = 100$	9.4	4.1	2.3	0.4	0.1	0.0
	Nucleus decoding $p = 0.1$	9.6	4.7	2.3	0.3	0.1	0.0
Continuous Transformer	Similarity Search	9.8	3.3	2.1	0.3	0.0	0.0
KG-Enhanced Transformer	Similarity Search	12.8	3.4	**1.6**	**0.4**	**0.2**	**0.1**

Table 5.2: Comparison between the evaluation scores on the OpenDialKG dataset for the baseline and the knowledge graph-enhanced transformer models.

Looking at the table, it can be seen that the knowledge graph-enhanced dialogue transformer architecture performs similarly to the larger language model variants, even without relying on pre-training and much smaller parameter count. While these results are not entirely representative of the true response qualities, it clearly shows that this technique improves on the baseline transformer models. For better insight into the difference between each technique, I present sampled dialogues in table 5.3. Further samples are available in the appendix.

According to my empirical analysis, the most linguistically correct responses are the results of pre-trained language models. While these techniques can produce generally sensible outputs, they lack factual accuracy, however the knowledge graph-enhanced transformer is able to inject somewhat correct named entities into the utterance as well. This is effect is enhanced for the OpenDialKG dataset particularly, as it contains large amounts

Input	Method	Model Output	Target
I like the movie "The Martian", could you recommend a movie similar to it?	GPT-2 (small) Beam Search $k = 5$	You might enjoy Cloverfield with Drew Barrymore or The Martian. They are both Sci-Fi films.	Do you like The Martian because it stars Matt Damon?
	GPT-2 (small) Nucleus decoding $p = 0.1$	Sure, do you prefer Sci-Fi or Adventure movies?	
	XLNet (base) Beam Search $k = 5$	It is starring Michael Fassbender, who also starred in X-Men: First Class and X-Man 2. Do you like movies that he stars in?	
	XLNet (base) Nucleus decoding $p = 0.1$	I do! I love Sci Fi movies as well. Have you seen either of those?	
	Transformer Nucleus decoding $p = 0.1$	This is a good one	
	Continuous Transformer	Have you seen The Running Man?	
	KG-Enhanced Transformer	It is written by James Patterson, starring Matt Damon and also stars Tracie Thoms. Do you like them?	

Table 5.3: Comparison of single turn responses for a validation example from the OpenDialKG dataset.

of proper nouns. In the absence of resources for conducting a proper empirical analysis of the models, I leave further consideration of the generation quality to the reader, as evaluation of responses can be highly subjective.

Chapter 6

Conclusion

As an overall summarization of my work, I consider it successful in various regards. While the basic idea of integrating knowledge graphs into dialogue generator models is an existing approach, my experiments with the presented baseline architectures and the implementation of the knowledge graph-enhanced model helped me deepen my knowledge in the field. I have created a modular framework for training and evaluating each of my discussed dialogue generators, which can be easily reused for further experiments in this subject. Most of my work can be accessed on a code sharing platform called Github, which has already proven to be useful for dozens of other developers and machine learning enthusiasts. I have gathered a wealth of well-established recipes for training dialogue-generating models that can speed up the work of not only myself but others as well. Over the course of the semester, I have improved the code quality and added several auxiliary techniques based on the feedback of several other contributors to the project. With the experience gained so far, I can now confidently identify and correct either hyperparameter related issues or algorithmic errors in several neural network architectures. While I was not able to achieve great improvements with the knowledge graph-enhanced dialogue transformer architecture compared to baseline pre-trained language models, there is still a great potential for further enhancements of this technique.

Chapter 7

Future Work

One of the greatest shortcomings of my project is the unavailability of correctly formatted conversational dataset with labeled knowledge graph entities. As discussed in earlier sections, OpenDialKG is not sufficient in this regard, as well as its topics mainly focus on enquiries for specific items, which in my opinion is not representative of casual conversations. For this purpose, I intend to create a new dialog corpus, which satisfies my expectations. Upon obtaining such dataset, I would like to improve the entity linker and graph decoder components, which are constrained by my currently used data.

Bibliography

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

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2015.

R. Balakrishnan and K. Ranganathan. *A Textbook of Graph Theory*. URL `http://meskc.ac.in/wp-content/uploads/2018/12/A-Textbook-of-Graph-Theory-R.-Balakrishnan-K.-Ranganathan.pdf`.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *CoRR*, abs/1607.04606, 2016. URL `http://arxiv.org/abs/1607.04606`.

P. Bonatti, S. Decker, A. Polleres, and V. Presutti. Knowledge graphs: New directions for knowledge representation on the semantic web (dagstuhl seminar 18371). *Dagstuhl Reports*, 8:29–111, 2018.

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 2787–2795, Red Hook, NY, USA, 2013. Curran Associates Inc.

A. Cetoli, Stefano Bragaglia, Andrew D. O'Harney, M. Sloan, and M. Akbari. A neural approach to entity linking on wikidata. In *ECIR*, 2019.

KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014. URL `http://arxiv.org/abs/1409.1259`.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL `http://arxiv.org/abs/1810.04805`.

Marjan Ghazvininejad, Chris Brockett, Ming-Wei Chang, Bill Dolan, Jianfeng Gao, Wen-tau Yih, and Michel Galley. A knowledge-grounded neural conversation model. *CoRR*, abs/1702.01932, 2017. URL `http://arxiv.org/abs/1702.01932`.

Karthik Gopalakrishnan, Behnam Hedayatnia, Qinlang Chen, Anna Gottardi, Sanjeev Kwatra, Anu Venkatesh, Raefer Gabriel, and Dilek Hakkani-Tür. Topical-Chat: Towards Knowledge-Grounded Open-Domain Conversations. In *Proc. Interspeech 2019*, pages 1891–1895, 2019. DOI: `10.21437/Interspeech.2019-3079`. URL `http://dx.doi.org/10.21437/Interspeech.2019-3079`.

Jia-Chen Gu, Tianda Li, Quan Liu, Zhen-Hua Ling, Zhiming Su, Si Wei, and Xiaodan Zhu. Speaker-aware bert for multi-turn response selection in retrieval-based chatbots, 2020.

Xu Han, Shulin Cao, Lv Xin, Yankai Lin, Zhiyuan Liu, Maosong Sun, and Juanzi Li. Openke: An open toolkit for knowledge embedding. In *Proceedings of EMNLP*, 2018.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. DOI: 10.1162/neco.1997.9.8.1735.

Ari Holtzman, Jan Buys, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration, 2019.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

Charles Lee Isbell, Jr. Michael Kearns, Dave Kormann, Satinder Singh, and Peter Stone. Cobot in lambdamoo: A social statistics agent. In *In Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 36–41. AAAI Press, 2000.

Robert L. Logan IV, Nelson F. Liu, Matthew E. Peters, Matt Gardner, and Sameer Singh. Barack's wife hillary: Using knowledge-graphs for fact-aware language modeling. *CoRR*, abs/1906.07241, 2019a. URL http://arxiv.org/abs/1906.07241.

Robert L. Logan IV, Nelson F. Liu, Matthew E. Peters, Matt Gardner, and Sameer Singh. Barack's wife hillary: Using knowledge-graphs for fact-aware language modeling, 2019b.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

Sachin Kumar and Yulia Tsvetkov. Von mises-fisher loss for training sequence to sequence models with continuous outputs. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=rJlDnoA5Y7.

Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 957–966, Lille, France, 07–09 Jul 2015. PMLR. URL http://proceedings.mlr.press/v37/kusnerb15.html.

Ni Lao, Tom Mitchell, and William W. Cohen. Random walk inference and learning in a large scale knowledge base. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 529–539, Edinburgh, Scotland, UK., July 2011. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/D11-1049.

Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alexander Peysakhovich. Pytorch-biggraph: A large-scale graph embedding system. *CoRR*, abs/1903.12287, 2019. URL http://arxiv.org/abs/1903.12287.

Liunian Harold Li, Patrick H. Chen, Cho-Jui Hsieh, and Kai-Wei Chang. Efficient contextual representation learning with continuous outputs. *Transactions of the Association for Computational Linguistics*, 7:611–624, March 2019. DOI: 10.1162/tacl_a_00289. URL https://www.aclweb.org/anthology/Q19-1039.

Yanran Li, Hui Su, Xiaoyu Shen, Wenjie Li, Ziqiang Cao, and Shuzi Niu. Dailydialog: A manually labelled multi-turn dialogue dataset. *CoRR*, abs/1710.03957, 2017. URL http://arxiv.org/abs/1710.03957.

Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, page 2181–2187. AAAI Press, 2015. ISBN 0262511290.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015. URL http://arxiv.org/abs/1508.04025.

Gonzalo Medina. Diagram of an artificial neural network. URL https://tex.stackexchange.com/questions/132444/diagram-of-an-artificial-neural-network.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013a. URL http://dblp.uni-trier.de/db/journals/corr/corr1301.html#abs-1301-3781.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013b. URL http://arxiv.org/abs/1310.4546.

Seungwhan Moon, Pararth Shah, Anuj Kumar, and Rajen Subba. OpenDialKG: Explainable conversational reasoning with attention-based walks over knowledge graphs. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 845–854, Florence, Italy, July 2019. Association for Computational Linguistics. DOI: 10.18653/v1/P19-1081. URL https://www.aclweb.org/anthology/P19-1081.

Isaiah Onando Mulang', Kuldeep Singh, Chaitali Prabhu, Abhishek Nadgeri, Johannes Hoffart, and Jens Lehmann. Evaluating the impact of knowledge graph context on entity disambiguation models. In *Proceedings of the 29th ACM International Conference on Information Knowledge Management*, CIKM '20, page 2157–2160, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368599. DOI: 10.1145/3340531.3412159. URL https://doi.org/10.1145/3340531.3412159.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652, 2014. URL http://arxiv.org/abs/1403.6652.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *CoRR*, abs/1802.05365, 2018. URL http://arxiv.org/abs/1802.05365.

Alec Radford. Improving language understanding by generative pre-training. 2018.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Alan Ritter, Colin Cherry, and William B. Dolan. Data-driven response generation in social media. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 583–593, Edinburgh, Scotland, UK., July 2011. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/D11-1054.

Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Kurt Shuster, Eric M. Smith, Y-Lan Boureau, and Jason Weston. Recipes for building an open-domain chatbot, 2020.

Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *International Conference on Acoustics, Speech and Signal Processing*, pages 5149–5152, 2012.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. DOI: 10.18653/v1/P16-1162. URL https://www.aclweb.org/anthology/P16-1162.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. *CoRR*, abs/1803.02155, 2018. URL http://arxiv.org/abs/1803.02155.

Eric Michael Smith, Mary Williamson, Kurt Shuster, Jason Weston, and Y-Lan Boureau. Can you put it all together: Evaluating conversational agents' ability to blend skills, 2020.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014. ISSN 1532-4435.

Haitian Sun, Bhuwan Dhingra, Manzil Zaheer, Kathryn Mazaitis, Ruslan Salakhutdinov, and William W. Cohen. Open domain question answering using early fusion of knowledge bases and text. *CoRR*, abs/1809.00782, 2018a. URL http://arxiv.org/abs/1809.00782.

Haitian Sun, Bhuwan Dhingra, Manzil Zaheer, Kathryn Rivard, Ruslan Salakhutdinov, and William Cohen. Open domain question answering using early fusion of knowledge bases and text. pages 4231–4242, 01 2018b. DOI: 10.18653/v1/D18-1455.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014. URL http://arxiv.org/abs/1409.3215.

Lucidworks Trey Grainger. Natural language search with knowledge graphs. URL https://youtu.be/5noi2VM9F-g.

Yi-Lin Tuan, Yun-Nung Chen, and Hung yi Lee. Dykgchat: Benchmarking dialogue generation grounding on dynamic knowledge graphs, 2019.

J Leon V. How do i draw an lstm cell in tikz? URL https://tex.stackexchange.com/questions/432312/how-do-i-draw-an-lstm-cell-in-tikz.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Oriol Vinyals and Quoc V. Le. A neural conversational model. *CoRR*, abs/1506.05869, 2015. URL http://arxiv.org/abs/1506.05869.

Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. *CoRR*, abs/1905.09418, 2019. URL http://arxiv.org/abs/1905.09418.

Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, September 2014. ISSN 0001-0782. DOI: 10.1145/2629489. URL https://doi.org/10.1145/2629489.

Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, January 1966. ISSN 0001-0782. DOI: 10.1145/365153.365168. URL https://doi.org/10.1145/365153.365168.

Mark Wibrow. How to draw recurrent neural network.

Ledell Wu, Adam Fisch, Sumit Chopra, Keith Adams, Antoine Bordes, and Jason Weston. Starspace: Embed all the things! *CoRR*, abs/1709.03856, 2017. URL http://arxiv.org/abs/1709.03856.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL http://arxiv.org/abs/1609.08144.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019. URL http://arxiv.org/abs/1906.08237.

Houyu Zhang, Zhenghao Liu, Chenyan Xiong, and Zhiyuan Liu. Grounded conversation generation as guided traverses in commonsense knowledge graphs. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2031–2043, Online, July 2020. Association for Computational Linguistics. DOI: 10.18653/v1/2020.acl-main.184. URL https://www.aclweb.org/anthology/2020.acl-main.184.

Saizheng Zhang, Emily Dinan, Jack Urbanek, Arthur Szlam, Douwe Kiela, and Jason Weston. Personalizing dialogue agents: I have a dog, do you have pets too? *CoRR*, abs/1801.07243, 2018. URL http://arxiv.org/abs/1801.07243.

Hao Zhou, Tom Young, Minlie Huang, Haizhou Zhao, Jingfang Xu, and Xiaoyan Zhu. Commonsense knowledge aware conversation generation with graph attention. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 4623–4629. International Joint Conferences on Artificial Intelligence Organization, 7 2018. DOI: 10.24963/ijcai.2018/643. URL https://doi.org/10.24963/ijcai.2018/643.

Appendix

List A.1: Samples from the KG-Enhanced Dialogue Transformer with a history size of 4, trained on the OpenDialKG dataset.

- [**Human**] Hi
- [**Model**] Hello, have you heard of the movie The Hobbit: Desolation of Smaug?
- [**Human**] No, can you tell me more about it?
- [**Model**] It is a fantasy adventure film starring James Nesbitt. Do you like this actor or would you like to know other movies he has starred in?
- [**Human**] List some of his movies please.
- [**Model**] Sure! He also starred in The Way and The Amateurs. Have you seen either of those?
- [**Human**] What is "The Way" about?
- [**Model**] It's a teen film, but it was directed by Marc Forster who also wrote the movie The Valley of Elah which stars Tom Hanks.
- [**Human**] Oh I see. Can you recommend me a Sci-Fi movie?
- [**Model**] Sure! White Televison and Undertow are good ones as well. Have you seen either of those?

List A.2: Samples from the GPT-2 (small) baseline model with Nucleus decoding $p = 0.1$, history size of 4, trained on the Persona Chat dataset.

- [**Human**] hi
- [**Model**] greetings, i'm gerald.
- [**Human**] how are you ?
- [**Model**] good, thanks. getting ready for another hike, i love hiking!
- [**Human**] oh realy? where are you going?
- [**Model**] pikes peak. i am curently exploring famous mountains when i have time.
- [**Human**] that is cool are you going alone?
- [**Model**] my brother is in the us, we go hiking with him every month.
- [**Human**] how old are you by the way?
- [**Model**] a little under 46 years old
- [**Human**] are you in a relationship?
- [**Model**] yes i am married, 3 boys, i am pregnant again. i am too busy to focus

List A.3: Samples from the Transformer baseline model with Beam Search decoding $k = 5$, history size of 4, trained on the Dialy Dialog dataset.

- [**Human**] hi how are you ?
- [**Model**] fine. i got a promotion at work earlier this week.
- [**Human**] where do you work ?
- [**Model**] I work at the polling place.
- [**Human**] where is that ?
- [**Model**] I live in the dorms across the street.
- [**Human**] do you like it ?
- [**Model**] it's great. I heard it is far from here.
- [**Human**] are you a student ?
- [**Model**] yes. I am a student too.