Chapter 2

Background

This chapter introduces core techniques used in this thesis. Section 2.1 introduces function approximation via neural networks and backpropagation. Subsequently, section 2.2 introduces function-free first-order logic. Finally, section 2.3 discusses prior work on automated knowledge base construction with neural networks, linking the first two sections together.

2.1 Function Approximation with Neural Networks

In this theses, we consider models that can be formulated as differentiable functions $f_{\boldsymbol{\theta}} : \mathcal{X} \to \mathcal{Y}$ parameterized by $\boldsymbol{\theta} \in \Theta$. Our task is to approximate such functions, *i.e.*, learn parameters from a set of training examples $\mathcal{T} = \{(x, y)\}$, where $x \in \mathcal{X}$ is the input and $y \in \mathcal{Y}$ some desired output. Both, x and y, can be structured objects. For instance, x could be a fact about the world, like $\texttt{directedBy}(\textsc{Interstellar}, \textsc{Nolan})$, and y a corresponding target truth score (*e.g.* 1.0).

2.1 **FIXME** proper set notation?

We define a loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \times \Theta \to \mathbb{R}$ that measures the discrepancy between a provided output y and a predicted output $\hat{y} = f_{\boldsymbol{\theta}_t}(x)$ on a training example given a current setting of parameters $\boldsymbol{\theta}_t$. We seek to find those parameters $\boldsymbol{\theta}^*$ that minimize this discrepancy on the training set. Our learning problem can thus be written as

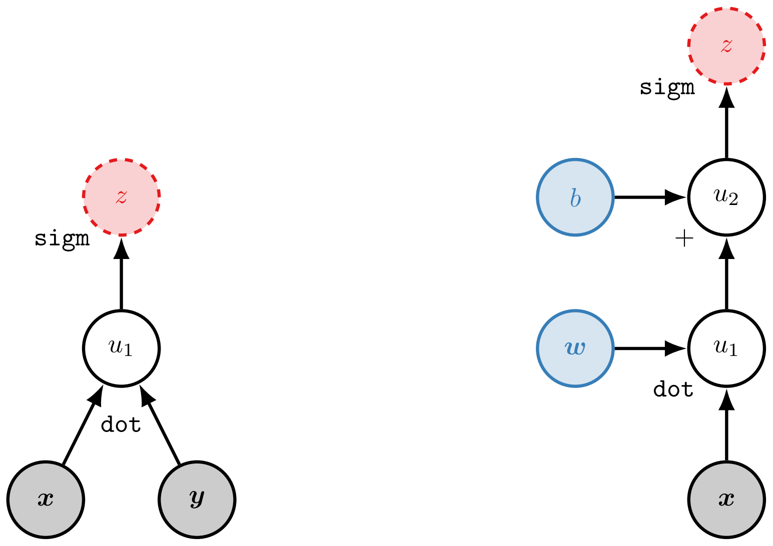
$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \frac{1}{|\mathcal{T}|} \sum_{(x,y)\,\in\,\mathcal{T}} \mathcal{L}(\hat{y}, y, \boldsymbol{\theta}) \tag{2.1}$$

Note that $\mathcal{L}$ is also a function of $\boldsymbol{\theta}$, since we might not only want to measure the discrepancy between given and predicted outputs, but also use a regularizer on the parameters for improved generalization. As $\mathcal{L}$ and $f_{\boldsymbol{\theta}_t}$ are differentiable functions, we can use stochastic gradient descent [Nemirovski and Yudin, 1978] for iteratively updating $\boldsymbol{\theta}_t$ based on mini-batches $\mathcal{B} \subseteq \mathcal{T}$ of the training data

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \epsilon\, \nabla_{\boldsymbol{\theta}_t} \frac{1}{|\mathcal{B}|} \sum_{(x,y)\,\in\,\mathcal{B}} \mathcal{L}(\hat{y}, y, \boldsymbol{\theta}_t) \tag{2.2}$$

where ϵ denotes a learning rate, and $\nabla_{\boldsymbol{\theta}}$ the gradient of the loss with respect to parameters given the current batch at time step t.

2.1 more generally: minimize expected loss over full data distribution $\mathbb{E}_{(x,y)\sim p_{\text{data}}}$



(a) $z = \sigma(\boldsymbol{x}^\top \boldsymbol{y})$ **(b)** $z = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$

Figure 2.1: Two simple computation graphs with inputs (gray), parameters (blue), intermediate variables u_i, and outputs (dashed). Operations are shown next to the nodes.

2.1.1 Computation Graphs

A useful abstraction for defining models as differentiable functions are computation graphs [Goodfellow et al., 2016]. In such a directed acyclic graph, nodes represent variables and directed edges from one or multiple nodes to another node correspond to a differentiable operation. As variables we consider scalars, vectors, matrices, and, more generally, higher-order tensors. We denote scalars by lower case letters x, vectors by bold lower case letters $\boldsymbol{x}$, matrices by bold capital letters $\boldsymbol{X}$, and higher-order tensors by Euler script letters $\mathcal{X}$. Variables can either be inputs, outputs or parameters of a model.

2.1 **TODO** and structured objects over these: tuples and lists

Figure 2.1a shows a simple computation graph that calculates $z = \sigma(\boldsymbol{x}^\top \boldsymbol{y})$. Here, σ and `sigm` refer to the element-wise (or scalar) sigmoid operation

$$\sigma(\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{x}}} \tag{2.3}$$

and `dot` and $\bullet^\top\bullet$ denote the dot product between two vectors. Furthermore, we name the ith intermediate expression as u_i. Figure 2.1b shows a slightly more complex computation graph with two parameters $\boldsymbol{w}$ and b. This computation graph in fact represents logistic regression $f(\boldsymbol{x}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$.

2.1 **TODO** use another example from AKBC instead?

2.1.2 Backpropagation

For learning from data, we need to calculate the gradient of the loss with respect to all model parameters. As all operations in the computation graph are differentiable, we can recursively apply the chain rule of calculus.

2.1 **TODO** how does it relate to nodes in the computation graph and their dependence on these parameters?

Chain Rule of Calculus Assume we are given a composite function $\boldsymbol{z} = f(\boldsymbol{y}) = f(g(\boldsymbol{x}))$ with $f : \mathbb{R}^n \to \mathbb{R}^m$ and $g : \mathbb{R}^l \to \mathbb{R}^n$. The chain rule allows us to

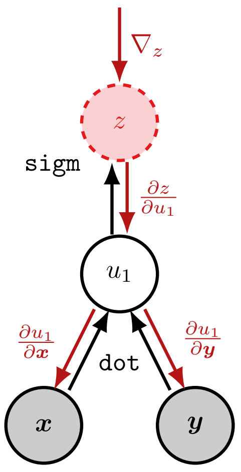

∇_z
z
sigm
$\frac{\partial z}{\partial u_1}$
u_1
$\frac{\partial u_1}{\partial \boldsymbol{x}}$
dot
$\frac{\partial u_1}{\partial \boldsymbol{y}}$
$\boldsymbol{x}$
$\boldsymbol{y}$

Figure 2.2: Backward pass for the computation graph shown in fig. 2.1a.

decompose the calculation of $\nabla_{\boldsymbol{x}} \boldsymbol{z}$, *i.e.*, the gradient of the entire computation $\boldsymbol{z}$ with respect to $\boldsymbol{x}$, as follows [Goodfellow et al., 2016]

$$\nabla_{\boldsymbol{x}} \boldsymbol{z} = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \right)^{\top} \nabla_{\boldsymbol{y}} \boldsymbol{z} \tag{2.4}$$

Here, $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the Jacobian matrix of g, *i.e.*, the matrix of partial derivatives, and $\nabla_{\boldsymbol{y}} \boldsymbol{z}$ is the gradient of $\boldsymbol{z}$ with respect to $\boldsymbol{y}$. Note that this approach generalizes to matrices and higher-order tensors by reshaping them to vectors before the gradient calculation and back to their original shape afterwards.

Backpropagation uses the chain rule to recursively define an efficient calculation of gradients of parameters (and inputs) in the computation graph by avoiding recalculation of previously calculated expressions. This is achieved via dynamic programming, *i.e.*, storing previously calculated gradient expressions and reusing them for later gradient calculations. We refer the reader to Goodfellow et al. [2016] for details. In order to run backpropagation with a differentiable operation f that we want to use in a computation graph, we need to be able to differentiate that function with respect to every of its inputs.

2.1 **TODO** should provide more details here

Example Let us take the computation graph depicted in fig. 2.1a as an example. Assume we are given an upstream gradient ∇_z and want to compute the gradient of z with respect to the inputs $\boldsymbol{x}$ and $\boldsymbol{y}$. The computations carried out by backpropagation are depicted in fig. 2.2. For instance, by recursively applying the chain rule we can calculate $\nabla_{\boldsymbol{x}} z$ as follows:

$$\nabla_{\boldsymbol{x}} z = \frac{\partial z}{\partial \boldsymbol{x}} = \frac{\partial z}{\partial u_1} \frac{\partial u_1}{\partial \boldsymbol{x}} = \sigma(u)(1 - \sigma(u))\boldsymbol{y} \tag{2.5}$$

Note that the computation of $\frac{\partial z}{\partial u_1}$ can be reused for calculating $\nabla_{\boldsymbol{y}} z$. We get the gradient of the entire computation graph (including upstream nodes) with respect to $\boldsymbol{x}$ via $\nabla_z \nabla_{\boldsymbol{x}} z$.

2.1 **TODO** also note that gradients from different upstream nodes are added (need example for that)

1	$\texttt{fatherOf}(\textsc{abe}, \textsc{homer})$.
2	$\texttt{parentOf}(\textsc{homer}, \textsc{lisa})$.
3	$\texttt{parentOf}(\textsc{homer}, \textsc{bart})$.
4	$\texttt{grandpaOf}(\textsc{abe}, \textsc{lisa})$.
5	$\texttt{grandfatherOf}(\textsc{abe}, \textsc{maggie})$.
6	$\texttt{grandfatherOf}(X, Y) :– \texttt{fatherOf}(X, Z), \texttt{parentOf}(Z, Y)$.
$\mathcal{P}$	$\{\texttt{fatherOf}, \texttt{parentOf}, \texttt{grandpaOf}, \texttt{grandfatherOf}\}$
$\mathcal{C}$	$\{\textsc{abe}, \textsc{homer}, \textsc{lisa}, \textsc{bart}, \textsc{maggie}\}$
$\mathcal{V}$	$\{X, Y, Z\}$

Table 2.1: Example knowledge base.

In chapter 3 we will use backpropagation for computing the gradient of differentiable propositional logic formulae with respect to vector representations of symbols to define models that combine representation learning with first-order logic.

2.2 Function-free First-order Logic

We now turn to a brief introduction of function-free first-order logic to the extent it is used in the subsequent chapters. This section follows the nomenclature of Datalog [Gallaire and Minker, 1978].

We start by defining an atom (short for atomic rule) as a predicate symbol and a list of terms.[1] A term can be a constant or a variable. For instance, $\texttt{grandfatherOf}(Q, \textsc{bart})$ is an atom with the predicate $\texttt{grandfatherOf}$, and two terms, the variable Q and the constant BART, respectively. We define the arity of a predicate to be the number of terms it takes as argument. Thus, $\texttt{grandfatherOf}$ is a binary predicate. A literal is defined as a negated or non-negated atom. A ground literal is a literal with no variables (see the first five rules in table 2.1). Furthermore, we consider first-order logic rules (also called clauses) of the form HEAD :– BODY, where BODY is a possibly empty conjunction of literals represented as list that we call the condition or premise of the rule, and HEAD is a literal called the conclusion, consequent or hypothesis.[2] An example is the sixth rule in table 2.1. Variables only appearing in the body of a rule are existentially quantified ($\exists Z$ in rule 6), whereas variables outside are universally quantified ($\forall X, Y$ in rule 6). A rule is a ground rule if all its literals are ground. We call a ground rule with an empty body a fact, so the first five rules in table 2.1 are facts. We define $\mathcal{S} = \mathcal{C} \cup \mathcal{P} \cup \mathcal{V}$ to be the set of symbols, containing constant symbols $\mathcal{C}$, predicate symbols $\mathcal{P}$, and variable symbols $\mathcal{V}$. In the context of first-order logic, we will use lower case names to refer to predicate and constant symbols, and upper case names for variables.

2.2 **TODO** add some historical notes

2.2 **TODO** foreshadow the difference between symbolic and subsymbolic representations

2.2 **TODO** give

[1] We will use predicate and relation synonymously.

[2] We will use rule, clause and formula synonymously.

2.3 Knowledge Base Inference with Neural Networks

Prominent recent approaches to knowledge base inference, for instance for automatically constructing or completing a knowledge base from text, learn representations of symbols vial neural models.

2.3 **TODO** benefits: generalization

2.3.1 Matrix Factorization

In this section we revisit the matrix factorization relation extraction model by Riedel et al. [2013]. We discuss this model in detail, as it the base on which we develop rule injection methods in chapter 3 and 4.

Assume a set of observed entity-pairs symbols $\mathcal{C}$ and a set of predicate symbols $\mathcal{P}$, which can either represent structured binary relations from Freebase [Bollacker et al., 2008] or unstructured OpenIE-like [Etzioni et al., 2008] textual patterns collected from news articles where the pattern appeared between two entities. Examples for structured and unstructured relations are `company/founders` and `#2-co-founder-of-#1`, respectively. Let $\mathcal{O} = \{r_s(e_i, e_j))\}$ be the set of observed atoms. Model F by Riedel et al. [2013] maps all symbols in a knowledge base to sub-symbolic representations, *i.e.*, it learns a dense k-dimensional vector representation for every relation and entity-pair. We will call such vectors representations or embeddings. Thus, a training fact $r_s(e_i, e_j)$ is represented by two vectors, $\boldsymbol{v}_s \in \mathbb{R}^k$ and $\boldsymbol{v}_{ij} \in \mathbb{R}^k$, respectively. The truth estimate of a fact is modeled via the sigmoid of the dot product of the two symbol representations:

$$p_{sij} = \sigma(\boldsymbol{v}_s^\top \boldsymbol{v}_{ij}) \tag{2.6}$$

In fact, this expression corresponds to the computation graph shown in fig. 2.1a earlier with $\boldsymbol{x} = \boldsymbol{v}_s$ and $\boldsymbol{y} = \boldsymbol{v}_{ij}$. The score p_{sij} is measuring the compatibility between the relation and entity-pair representation and can be interpreted as the probability of the fact being true under the model. We would like to train symbol representations such that true facts are scored close to 1.0 and false facts close to 0.0. This is matrix factorization

2.3 **TODO** probabilistic matrix factorization / generalized PCA!?

$$\boldsymbol{K} \approx \sigma(\boldsymbol{P}^\top \boldsymbol{C}) \tag{2.7}$$

Bayesian Personalized Ranking

2.3 **REF** Rendle et al. [2009b]

Typically no negative evidence is available at training time, and therefore a Bayesian Personalized Ranking (BPR) objective Rendle et al. [2009a] is used. Given a pair of facts $f_p := \langle r_p, t_p \rangle \notin \mathcal{O}$ and $f_q := \langle r_q, t_q \rangle \in \mathcal{O}$, this objective requires that

$$\boldsymbol{r_p}^\top \boldsymbol{t_p} \leq \boldsymbol{r_q}^\top \boldsymbol{t_q}. \tag{2.8}$$

Training Loss The embeddings can be trained by minimizing a convex loss function ℓ_R that penalizes violations of that requirement when iterating over the training set. In practice, each positive training fact $\langle r, t_q \rangle$ is compared with a randomly sampled unobserved fact $\langle r, t_p \rangle$ for the same relation. The overall loss can hence be written as

2.3 **FIXME**

$$\mathcal{L}_R = \sum_{\substack{\langle r, t_q \rangle \in \mathcal{O} \\ t_p \in \mathcal{T},\ \langle r, t_p \rangle \notin \mathcal{O}}} \ell_R\big(\boldsymbol{r}^\top [\boldsymbol{t_p} - \boldsymbol{t_q}]\big). \tag{2.9}$$