Università degli Studi di Pisa

DIPARTIMENTO DI INFORMATICA
Corso di Laurea Magistrale in Informatica LM-18

Conditional Variational Autoencoders for Tree-Structured data

Candidato:
Michele Colombo

Relatore:
Davide Bacciu

Anno Accademico 2017–2018

Conditional Variational Autoencoders for Tree-Structured data

Michele Colombo

November 30, 2018

Abstract

The thesis deals with the design of a deep learning model that can learn a generative process realizing unconstrained tree transductions. The model is based on an extension of the popular Variational Autoencoder framework to allow conditioning the generative process on tree-structured inputs and to generate tree-structured predictions. An efficient Tensor-Flow implementation of the proposed model has been realized and validated on Arithmetic Expression trees and Neural Machine Translation.

Contents

1 Introduction

Several kinds of interesting data have an inherent dynamical structure which is often ignored or simplified. Mainly due to the challenges related to dealing with data whose structure changes from one sample to another.
However exploiting the relational information made available by structured data might considerably boost the final performance in the target task and in some scenarios ignoring them might even tamper the learning process.

Sequential data represents the simplest form of dynamically structured data. Indeed they have been the first to be attacked and nowadays they are the most popular dynamically structured data you can find in the deep learning panorama. Much work has been done and many different models and approaches have been proposed, all falling in general under the class of Recurrent Neural Networks. Some of them emerged as widely accepted fundamental tools to consider when developing a deep learning application. Most popular frameworks natively support them providing ready to use implementations: from Simple Recurrent Networks [1], [2] to more involved architectures such as Long Short Term Memory [3] and Gated Recurrent Unit [4].
The kind of structural information sequences convey is related to a total ordering among a varying number of elements. Yet they turned out to be really powerful and useful. Many real life applicative scenarios have a good fit on sequential data: for instance we can employ them whenever we are dealing with time series or with natural languages.

In fact sequential models provide a simple way to deal with any kind of structured data: they are able to handle sequences of a variable number of elements, thus we can employ them with some linearisation of our arbitrarily structured data. For instance when dealing with trees we can linearise nodes according to some visit order, or just take leaves left-to-right ignoring internal nodes. This is often the easiest approach due to the popularity of sequential model.

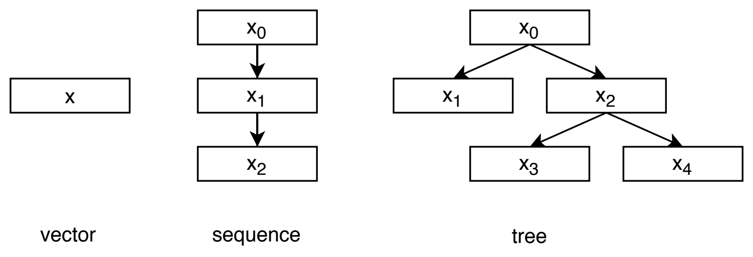

Figure 1: Simple example of how samples can be structured. From a single flat vector to multiple vectors arranged in a sequence or in a tree. Artificial Neural Networks deal with them with increasing difficulty, in the order: vectors, sequences, trees. Note that sequence and tree structures are possibly different across different samples.

However, approaches based on structure linearisation tend to discard structural information which might be relevant to increase the predictive task accuracy. A model which exploits those structural information might better performs.
The intuition to understand the disruptive effect of linearisation is that by such an approach one typically alters the semantic proximity of the composing elements. Elements close in the original structure get furthered away in the linearised one and vice versa.
The impact on the learning process might be significant: Recurrent Neural Networks employed to deal with sequences are usually able to better capture dependencies among close elements and often struggle to remember and use distant information.

Another interesting aspect to consider is that the linearised data structure introduces artificial and unrequired dependencies in the computation. In fact it introduces the maximum amount of synchronized computations, making it extremely inefficient due to poor parallelization. For instance if we consider a complete binary tree of depth k in the sequential approach we would have a computation graph of depth 2^k, while in the tree approach would be only k.
All these last considerations are those which drove us to investigate deep learning models for more complex data structures, more specifically for tree structured-data.

As soon as we move a step toward more involved structures, such as trees, the popularity drops and fewer results can be found in the literature. In particular for unconstrained tree transduction, the setting we are interested into, some models and approaches have been proposed but none of them has yet emerged. In fact, to the extent of our knowledge, the most popular deep learning frameworks do not provide any out-of-the-box mechanism to employ dynamically structured data different from sequences.
Models capable to deal with such data have to face several challenges. Compared to the sequential models it is much more demanding to find an effective design and to have an efficient practical implementation. This, along with the fact that a sequential model often leads to a reasonable result with less efforts, is what possibly kept deep learning models for more involved structured data apart from the main scene.

However trees provide a quite natural way to add hierarchical information to otherwise flat data. They find really wide use in many different fields, for instance when dealing with context-free languages they can be used to prove the validity of a sentence providing its derivation tree, while with programming languages they are used to represent programs in the form of Abstract Syntax Trees. Moving to natural languages they are used to augment sentences with grammatical and semantic information by means of constituency and dependency trees. More in general they can model entities relationships, as for instance in a 2d or 3d scene; in chemistry they find their use in modelling several molecule structures.

In all the aforementioned examples, trees are a natural way to structure the information at hand. Directly employing them in our deep learning models, instead of some possibly lossy surrogate, might considerably boost the model performance. Motivated by this rationale, in this thesis we focus on building a deep learning model able to deal with tree-structured data both in input and output.

In particular we set ourself in the general scenario of unconstrained tree transduction. We want a model capable of learning a mapping between trees. Additionally to the challenge of dealing with dynamically structured data our model has also to be able to learn arbitrary relationship among differently structured data. On this specific task really little results can be found.

We tackled the unconstrained tree transduction problem in a probabilistic manner, modelling it as a generative process characterized by some joint distribution $P(Y|X)$; where both X and Y are tree-structured random variables. Thus our objective is to learn from examples the induced probabilistic mapping $X \rightsquigarrow Y$, indirectly learning to approximate the underlying joint distribution $P(Y|X)$.

These kind of models might turn out to be useful in all of those scenarios where we want to learn a transformation among trees, whether it is deterministic or probabilistic. For instance a Machine Translation task might benefits from the employment of constituency or dependency trees. On one hand this add information from some linguistic model not presents in the sentence it self, on the other hand the structure might better represents token dependencies, possibly helping the learning process to capture them.

Such a problem statement closely matches the capabilities of a well known generative model: Variational Autoencoder [5]. This kind of models aims at learning the input data distribution and once trained allows to generate new samples that likely belong to the original dataset. In particular in their Conditional variant the whole generative process is conditioned in order to model a joint distribution, as required by our target task.
In the few years after the seminal work have appeared, these models gained a lot of attention becoming the most popular generative model along with Generative Adversial Network [6]. Possibly also because of the striking visual results they achieved in image generations, see for instance [7]–[9]. In fact on different domains their capabilities have not been extensively investigated yet.

In this work we designed a deep learning Conditional Variational Autoencoder capable of dealing with tree-structured data. We realized and open-sourced[1] an efficient implementation in TensorFlow [10], one of the most popular machine learning frameworks. To assess the implementation efficiency we ran benchmarks on some different architectures, both employing CPUs and GPUs. To assess its learning capabilities we employed an artificial tasks involving Arithmetic Expression Trees and a real life task of Neural Machine Translation using sentences in their constituency tree.
Experiments and benchmarks outlined our implementation as practically employable to tackle real life tasks in terms of computational efficiency. Its learning capabilities have been proved as well, however experiments suggest that to ob-

[1]`https://github.com/m-colombo/conditional-variational-tree-autoencoder`

tain competitive performances on real life tasks we need to endow it with more powerful mechanisms, such as gates and attention based ones.

Thesis structure In the next section we present the Variational Autoencoder and its Conditional variant, we introduce some background and related works on deep learning model applied to structured data and their intersection with Variational Autoencoders. Section 3 details the specific of our models design, with particular attention on the Recursive Neural Network dealing with the tree-structured data. Section 4 presents some aspects specifically related to our implementation and details the benchmark and he experiments we ran. Finally in the last section we gather up conclusions and set some directions for future works.

Notation

- For brevity we will often write $P(x)$ rather than $P(X = x)$ to denote the probability value of the random variable X having value x;
- we employ the expectation symbol $\mathbb{E}$ to denote the expectation on continuous probability distributions and the average on finite sets;
- we write $x \sim P(X)$ to denote the sampling of a value x from the probability distribution $P(X)$. When written below an operator it instead denotes the range of values and the name they are bound to, for instance $\mathbb{E}_{x\sim P(X)}$ compute the expectation among all the possible values of X, binding them to the name x;
- $\mathcal{D}[P||Q]$ denotes the Kullback-Leibler Divergence [11] among P and Q;
- $\mathcal{N}(X; \mu, \sigma)$ denotes the isotropic normal distribution of the random variable X having mean μ and diagonal covariance σ;
- $\mathcal{N}$ is also employed to denote sub-networks along with $\mathcal{T}, \mathcal{A}, \mathcal{I}$, and $\mathcal{C}$, from the context will be clear whether we are referring to a sub-network or to the normal distribution;
- $x \cdot y$ denotes the vector concatenation.

2 Background

2.1 Variational Autoencoders

The high level architecture of our model comes from a specialization of the classical Variational Autoencoders framework [5]. As their name suggests, one of this kind of models possible interpretations characterizes them as autoencoders with additionally the capabilities of a generative model.

Autoencoders are, in general, a family of neural network architectures that aim at finding an effective encoding for the samples is some dataset at hand. They are able to exploit unlabelled data to learn how to compress them into a smaller space while preserving relevant features.
The overall architecture of the simplest version of an autoencoder, as depicted in Figure 2, consists of an encoder network and a decoder network: the former learns to project the input sample into some smaller space, the latter tries to reconstruct the original sample from this compressed encoding.
Intuitively you can think of it as an information bottleneck game. Both networks do their best to cooperate and find the most effective way to communicate all the relevant information, trough only a limited and usually undersized channel. As long as both the employed networks are differentiable the whole system is differentiable as well. Therefore we can train it with the usual techniques such as Stocastic Gradient Descent. The training procedure simply consists in minimizing the reconstruction error of the original input sample compared with the one obtained from the autoencoder. During the training process the encoder network will learn to extract from the samples the most relevant features accordingly to the employed loss function.

Along with the original Autoencoder architecture [12] some different variants have been proposed trying to find ways to learn better representations of the input samples. For instance: Denoising Autoencoders [13] tries to find more robust representations adding noise to the input, Contractive Autoencoders [14] instead explicitly inject the requirement in the loss function, Sparse Autoencoders [15]–[17] enforce sparsity in the encoding, which turned out to be a useful property. All the aforementioned autoencoders find their use mostly as a sort of preprocessing step of the input: the encoder is first trained along with the decoder and then is employed as the first layer in the network used to tackle the actual task at hand. This setting possibly allows to exploit more data than the ones available for the target task; particularly when it is a supervised one. As in a divide et impera approach we first look for a good input representation, which likely captures meaningful features for the task at hand and makes them easier to be grasped by the target task network. Then we try to solve the target task employing the encoder network as first layer, possibly fine tuning it along with the target task network.
An interesting advantage is that the encoder can be reused and possibly specialized for many different target tasks whose input space is the same.

Despite how good might be such models at encoding relevant features of the input into a smaller space there are no easy ways to obtain a generative model. In other words a random sample drawned from the encoding space is not easily

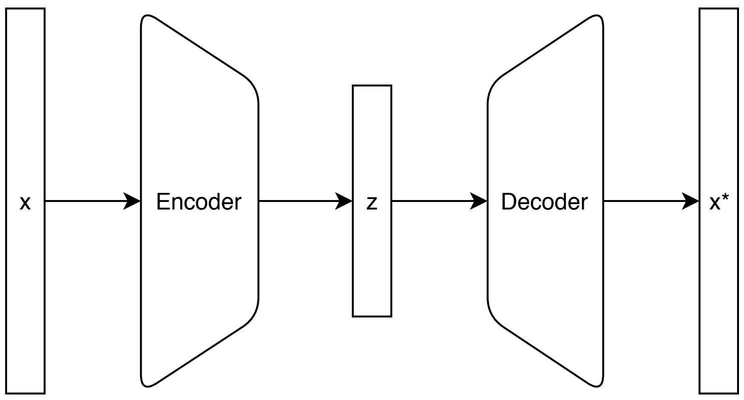

Figure 2: Example of a simple Autoencoder architecture. The input sample x is projected into a compressed encoding z by the encoder network, then the decoder network tries to reconstruct the original sample from it.

guaranteed to generate a reasonable sample when fed to the decoder. Some results towards that direction have been made [18], however in order to generate a proper sample a Markov Chain Monte Carlo is required.

Variational Autoencoders [5], [19] enforce how the mapping to the smaller space is performed and leads to an easy to use generative model. We can sample an encoding from the smaller space and feed it to the decoder, likely we will obtain a reasonable sample from the same space as the input space. Indeed, along with Generative Adversarial Networks [6], Variational Autoencoders have rapidly became one of the most popular generative models leading to some of the most striking Machine Learning application lately these last years. For instance [9] generates images from a description provided as a list of visual attributes, [20] models music, [21] performs trajectory forecasting on static images, [22] introduces a system able to arbitrary change facial expression of pictures, [8] learns to color black and white pictures, [7] combines VAE and GAN to build an unsupervised image-to-image translation model.

An easy yet a bit simplistic interpretation of Variational Autoencoders characterises them as autoencoders enriched with probability. As shown in Figure 3 the encoder network Q is now probabilistic, it computes a distribution over the possible encodings of the input sample. The decoder instead can be deterministic or probabilistic as well, the latter gives some advantages while training although it is often hard to model probabilities in the input-space. The last bit that makes this a generative model is to constrain $Q(Z|x)$ to be close to some prior distribution $P(Z)$, apart from acting as a regularization method this allows us to use $P(Z)$ to generate samples that are likely to belong to the input-space.

2.1.1 Characterization

Although they contains *Autoencoder* in their name and their overall architecture actually resembles an autoencoder architecture, an encoder and a decoder network can be recognized, the reasoning motivating them it is quite far from the

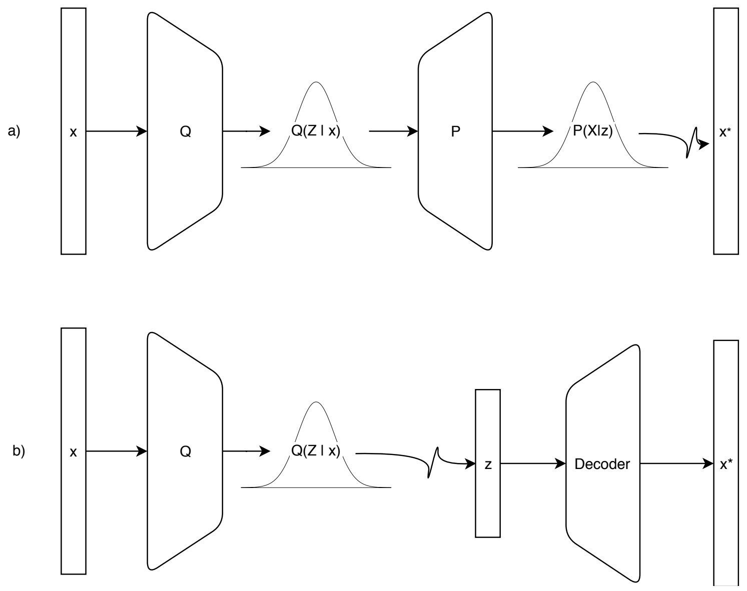

Figure 3: Two architecture variants of a Variational Autoencoder. Bell-shaped figures represent distributions and the curly arrow is the sampling operation. In a) a probability distribution is generated also by the decoder, when it is not feasible to model a distribution in the input-space the b) variant it is employed, sampling directly from the encoding space.

one behind classic autoencoders and more rooted in the Variational Inference world. Since the beginning VAE set their self as a generative model, thus the aim is to fit the dataset distribution $P(X)$ and have a way to sample from it, i.e. to generate sample from the learnt distribution.

The dataset is assumed to be generated by some two steps random process: first a latent variable z is generated from the prior distribution $P(Z)$ then the actual sample is generated from some conditional distribution $P(X|z)$. Intuitively we can think of the generative process first step as the one choosing the most fundamental features of the final sample, those that are going to affect all the generation process, while the second step chooses the remaining details accordingly to the latent variable value. For instance in the classical toy-problem of MNIST [23] where the dataset is composed of handwritten digits images the latent variable would surely encodes which digit will appear in the final image. In this setting we can decompose our maximization target in terms of the two steps thanks to total probability rule:

$$P(X) = \int P(X|z)P(z)\ dz. \tag{1}$$

The optimization target in this form however is not yet tractable, basically for two reasons: first the integral computation, even approximated, is not feasible in high-dimensional space; second the latent-space prior $P(Z)$ is unknown.
The unusual solution that VAEs employ to solve the latter is to arbitrary choose the latent variable distribution. Intuitively it does not really matter how the

latent variable values are distributed as long as they are able to encode the required information. Moreover, assuming there exists a particular ground truth distribution $P_{gt}(Z)$, it is always possible to map it to any other arbitrary distribution of the same dimensionality [24], the VAE will learn to perform such a mapping alongside the whole training procedure.
The usual choice for $P(Z)$ is a multivariate Gaussian with zero means and diagonal unitary covariance $\mathcal{N}(Z; 0, I)$. However, even if it is able to encode the original prior distribution $P(Z)$, it totally hides any possible interesting latent structure acting as a black box. In fact it is possible to try to shape the latent-space in a more meaningful way, enforcing or inducing some structure. For instance: [25] uses as latent space a discrete language model in order to perform text-summarization while [26] introduces a model and a training methodology to learn interpretable representation of images.

So in the usual scenario employing a multivariate normal prior our maximization target is now:

$$P(X) = \int P(X|z)\mathcal{N}(Z; 0, I)dz. \tag{2}$$

At this point we know all the terms: $P(X|z)$ is part of the model we are optimizing so we can compute it, generally implemented as a neural network.
Conceptually we are able to approximate our target function sampling the integral argument for enough values of z, however for high-dimensional space this can be intractable in practice.
The key idea of Variational Autoencoder comes from the consideration that most values of z bring basically no contribution to the summation, i.e. $P(x|z) \approx 0$, thus we could sample only the values of z that are likely to have generated x.
In other words we would like to sample z from $P(Z|x)$ but since it is unknown we try to approximate it by means of a new learnt distribution $Q(Z|x)$, which in our scenario it is realized by a neural network.
To express that $Q(Z|x)$ has to be close to $P(Z|x)$ we employ the Kullback-Leibler Divergence [11], denoted by $\mathcal{D}$, and with some algebraic transformations we will end up with the core equation of Variational Autoencoders:

$$\begin{aligned}
\mathcal{D}[Q(Z|x)\|P(Z|x)] &= \int_{-\infty}^{+\infty} Q(z|x) \log \frac{Q(z|x)}{P(z|x)} dz && (3)\\
&= \mathop{\mathbb{E}}_{z\sim Q(Z|x)} \left[\log \frac{Q(z|x)}{P(z|x)}\right] && (4)\\
&= \mathop{\mathbb{E}}_{z\sim Q(Z|x)} [\log Q(z|x) - \log P(z|x)]. && (5)
\end{aligned}$$

As usual $\mathbb{E}_{z\sim Q(Z|x)}$ computes the expected value with respect to the distribution $Q(Z|x)$. Now applying Bayes' rule we can get rid of the unknown term $P(z|x)$:

$$\begin{aligned}
\mathcal{D}[Q(Z|x)\|P(Z|x)] &= \mathop{\mathbb{E}}_{z\sim Q(Z|x)}\left[\log Q(z|x) - \log\frac{P(x|z)P(z)}{P(x)}\right] &(6)\\
&= \mathop{\mathbb{E}}_{z\sim Q(Z|x)}[\log Q(z|x) - (\log P(x|z) + \log P(z) - \log P(x))] &(7)\\
&= \mathop{\mathbb{E}}_{z\sim Q(Z|x)}[\log Q(z|x) - \log P(x|z) - \log P(z)] + \log P(x). &(8)
\end{aligned}$$

With some more algebraic transformations we can get another Kullback-Leibler Divergence term from the right hand side, it often comes in handy since for some distributions, e.g. multivariate Gaussian, it can be computed analytically:

$$\begin{aligned}
\mathcal{D}[Q(Z|x)\|P(Z|x)] &= \mathop{\mathbb{E}}_{z\sim Q(Z|x)}[\log Q(z|x) - \log P(x|z) - \log P(z)] + \log P(x) &(9)\\
-\mathcal{D}[Q(Z|x)\|P(Z|x)] &= \mathop{\mathbb{E}}_{z\sim Q(Z|x)}[\log P(x|z) - (\log Q(z|x) - \log P(z))] - \log P(x) &(10)\\
&= \mathop{\mathbb{E}}_{z\sim Q(Z|x)}[\log P(x|z)] - \mathop{\mathbb{E}}_{z\sim Q(Z|x)}\left[\frac{\log Q(z|x)}{\log P(z)}\right] - \log P(x) &(11)\\
&= \mathop{\mathbb{E}}_{z\sim Q(Z|x)}[\log P(x|z)] - \mathcal{D}[Q(Z|x)\|P(Z)] - \log P(x). &(12)
\end{aligned}$$

With a final rearrangement of terms we obtain the equations at the core of Variational Autoencoders:

$$\begin{aligned}
\log P(x) - \mathcal{D}[Q(Z|x)\|P(Z|x)] &= \mathop{\mathbb{E}}_{z\sim Q(Z|x)}[\log P(x|z)] - \mathcal{D}[Q(Z|x)\|P(Z)] &(13)\\
\log P(x) &\geq \mathop{\mathbb{E}}_{z\sim Q(Z|x)}[\log P(x|z)] - \mathcal{D}[Q(Z|x)\|P(Z)]. &(14)
\end{aligned}$$

We started with the aim to make $Q(Z|x)$ approximates $P(Z|x)$ and after some algebraic manipulations we end up with something more interesting. Equation 13 tells us that if we want to maximize $P(x)$ we can maximize $\mathbb{E}_{z\sim Q(Z|x)}[\log P(x|z)] - \mathcal{D}[Q(Z|x)\|P(Z)]$ given that we have an error due to our approximation of the posterior $P(Z|x)$ with $Q(Z|x)$. Equation 14 is just a reformulation expressing the so called variational lower bound. Luckily we will be actually able to compute and optimize the right hand side of Equation 13 and 14, just a few more things are needed.

Reparameterization trick Let us focus on the first term $\mathbb{E}_{z\sim Q(Z|x)}[\log P(x|z)]$: it is an expectation and as usual in stochastic gradient descent we can approximate it only using few samples from $Q(Z|x)$. The fundamental bit is that our sampling technique should let the gradient flows throw the network implementing $Q(Z|x)$. The game changing trick introduced in [5] is to refactor the

randomness seed in a way it does not stop the gradient.
In the case of $Q(Z|x)$ being a multivariate Gaussian with mean μ and diagonal covariance σ, sampling from $z \sim Q(Z|x)$ can be implemented as sampling from another distribution of the same dimensionality as $n \sim \mathcal{N}(N; 0, I)$ and them deterministically compute $z = \mu + n\sigma$. The feasibility of this approach clearly depends on the type of distribution we are employing, for instance for discrete distributions there are some results [27], [28].

Kullback-Leibler Divergence We have left to compute the Kullbak-Leibler Divergence term $\mathcal{D}[Q(Z|x)\|P(Z)]$, interestingly when both distribution are multivariate Gaussian of the same dimensionality it can be computed analytically in closed form. Here the choice of how to model the latent prior comes in handy. Be $P(Z) = \mathcal{N}(Z; 0, I)$ and $Q(Z|x)$ an isotropic multivariate normal, which in practice it is estimated by two neural networks computing μ and Σ:

$$Q(Z|x) = \mathcal{N}(Z; \mu(x), \Sigma(x)) \tag{15}$$

we can then directly compute the divergence term as:

$$\begin{aligned}
\mathcal{D}[\mathcal{N}(Z;\mu(x),&\Sigma(x))\|\mathcal{N}(Z;0,I)] = && (16)\\
&= \frac{1}{2}\left(\mathrm{tr}(\Sigma(x)) + \mu(x)^T\mu(x) - k - \log\,\det(\Sigma(x))\right) && (17)\\
&= \frac{1}{2}\left(\sum_k \Sigma(x)_{k,k} + \sum_k \mu(x)_k^2 - \sum_k 1 - \log\prod_k \Sigma(x)_{k,k}\right) && (18)\\
&= \frac{1}{2}\left(\sum_k \Sigma(x)_{k,k} + \sum_k \mu(x)_k^2 - \sum_k 1 - \sum_k \log\Sigma(x)_{k,k}\right) && (19)\\
&= \frac{1}{2}\sum_k \left(\Sigma(x)_{k,k} + \mu(x)_k^2 - 1 - \log\Sigma(x)_{k,k}\right). && (20)
\end{aligned}$$

Auto-Encoding VB Algorithm [5] Now we can actually compute and optimize the right hand side of Equation 13 with Stochastic Gradient Descent. In practice the schematics are those outlined in Figure 3: for a mini-batch of input samples we first compute their encoding distributions $Q(Z|x)$, then we compute the Kullback-Leibler Divergence with the prior $\mathcal{D}[Q(Z|x)\|P(Z)]$ and add it to the minimization target. Afterwards if we are able to differentiably compute $P(x|z)$ we can maximize it, as for variant a) of Figure 3.
In many scenarios however it is not feasible to directly model the likelihood distribution $P(X|z)$ and maximize it for a particular value x. In those cases the mapping from the latent space is assumed to be some deterministic function f, which in our model is implemented by a neural network. To sample from the likelihood we can now sample from the prior $z \sim P(Z)$ and deterministically build the associated sample $f(z)$ as in variants b) of Figure 3. In this way we can indirectly maximize $P(x|z)$ by minimizing, for some proper reconstruction loss $\mathcal{L}$:

$$\mathop{\mathbb{E}}_{z\sim P(Z)} \mathcal{L}(f(z), x). \tag{21}$$

In practice this is achieved by mean of a stochastic approximation, optimizing it only for a finite number of sampled z.

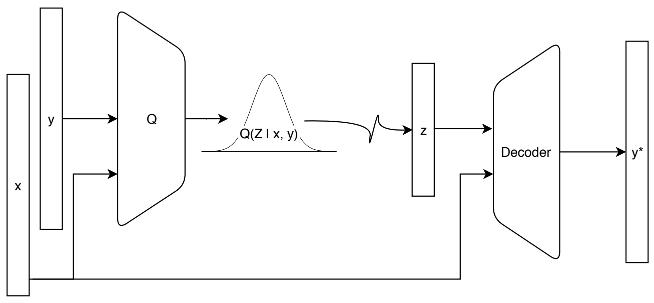

Figure 4: Architecture of a Conditional Variational Autoencoder in the variant employing stochastic approximation to maximize $P(y|z, x)$. Bell-shaped figures represent distributions and the curly arrow is the sampling operation.

2.1.2 Conditional Variational Autoencoders

A Conditional Variational Autoencoder, as depicted in Figure 4, just adds a conditioning to all the process: rather then modelling $P(X)$ we are now going to model $P(Y|X)$. So the encoder will compute $Q(Z|y, x)$, the latent prior will be $P(Z|x, y)$ and the decoder will, directly or indirectly, model $P(Y|z, x)$.
This setting allows us to learn how to perform a probabilistic mapping from X to Y accordingly to their joint distribution: given an input x we can draw a code from the prior $z \sim P(Z|x, y)$ and then sample an image $y \sim P(Y|x, z)$ by means of the decoder.

2.2 Neural Networks for Tree-Structured Data

The Conditional VAE described in the previous section defines an abstract architecture for different types of samples, it does not make any assumption on the input data structures. In fact in our model the capability of dealing with tree-structured data is totally hidden in the encoder and decoder modules.
In order to better understand the challenges of designing neural networks for dynamically structured data we start introducing basic feed forward networks and we progress gradually towards enough powerful architectures.

The basic building block of most of the more involved neural architectures is represented by Feed Forward Networks. In general they can be modelled as a function $\mathcal{N}$ mapping vectors from $\mathbb{R}^m$ to vectors in $\mathbb{R}^n$. The most classical example is represented by the Multilayer Perceptron (MLP) [29], a composition of $d \geq 2$ neural layers obtained as a parallel concatenation of artificial neurons as the one depicted in Figure 5. Each layer applies a linear mapping $\omega^{(i)} \in \mathbb{R}^{n_i \times m_i}$ to its input and then point-wisely applies an activation function ϕ_i. Formally:

$$\mathcal{N} : \mathbb{R}^{m_1} \to \mathbb{R}^{n_d} = \phi_d(\omega^{(d)} \ \dots \ \phi_2(\omega^{(2)}\phi_1(\omega^{(1)}x))) \tag{22}$$

$$\forall i \in (1, d] : m_i = n_{i-1}. \tag{23}$$

In order to ensure layers compatibility, in terms of matrix-vector multiplication, we need to add the consistency constraint of Equation 23. This might give us a first hint on how difficult might be for neural networks to deal with more structured and dynamical data. In fact a simple feed forward neural network,

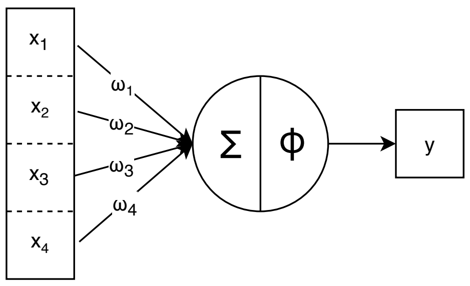

Figure 5: A single artificial neuron depicted in its biological inspired representation. Each of the input elements is sensed by means of a weighted connection which is used to summarize all of them in a single value and then obtain the output by processing it with the activation function ϕ. Shortly it computes $\phi(\sum_i \omega_i x_i)$. Neural layers are obtained as a parallel concatenation of many artificial neurons applied to the same input.

as the one just introduced, can be employed only with regular and flat data, i.e. it can only ingest vectors in $\mathbb{R}^{m_1}$ and outputs vectors in $\mathbb{R}^{n_d}$.

As soon as we have to deal with dynamically structured data it is therefore hard to effectively employ a feed forward network. The input structure and size change from one sample to another, while a feed forward network requires all the input to have the same size. Hence we need some more expressive neural network architecture able to dynamically adapt to the input structure [12], [30], namely Recurrent Neural Network for the sequential case and more in general Recursive Neural Networks .

As anticipated in the introduction section, the simplest and most popular form of dynamically structured data are sequences. Every sample is now an homogeneous sequence of elements:

$$s_k = x_k^{(1)}, \dots, x_k^{(l_k)} \qquad \forall i, k.\ x_k^{(i)} \in \mathbb{R}^m.$$

In simple words every elements among all the sequences must have the same shape but different sequences can have a different number of elements. Note that for readability we will often write x_i to denote $x_k^{(i)}$ when k it is not relevant.
The key idea underlying the efficacy of Recurrent and Recursive Neural Networks is named *weight sharing*: a technique with which the same (sub-)network is instantiated with the same parameters multiple times, assembling a bigger network. This makes it possible to adapt a model with a fixed number of parameters, such a feed forward network, to learn to generalize also with dynamically structured data.
In such a setting usual gradient based optimization techniques can be employed, the only difference is that the gradient will possibly flows multiple times through the same parameters. Figure 6 shows two simple examples of how weight sharing can be employed to deal with sequences.

A closer look at Figure 6 outlines some fundamental aspects of Recursive Neu-

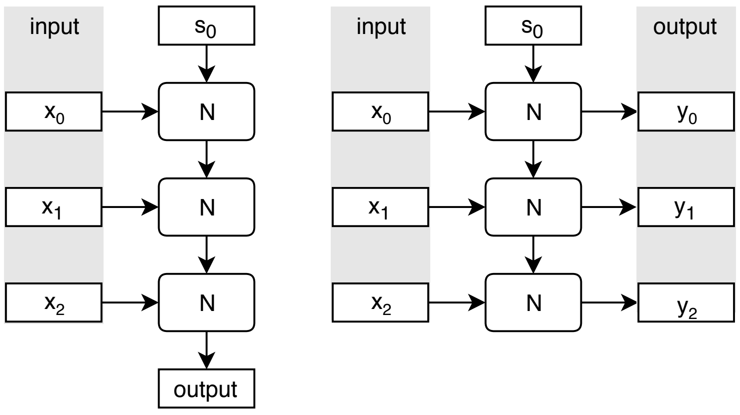

Figure 6: An example of a Recurrent Neural Network applied to a sequence of three elements. On the left it is employed to learn to map a sequence to a single value. On the right it performs sequence transduction. Note the employment of weight sharing: all the sub-networks N are identical feed forward network with the same parameters value.

ral Networks: in order to consider the input sample as a whole, rather than as independent elements, the sub-networks instances have to be connected. This makes it necessary to carefully design the employed sub-networks. Consistency constraints analogously to the one of Equation 23 have to be enforced in order to be able to compute the matrix-vector multiplications at the core of almost every neural network model. The more the involved structure is variable the more care is required to ensure such a compositionality property to hold in all the possible configurations.

Let us consider more formally the sequential transduction case of Figure 6, let $\mathcal{N}$ be a MLP employed as sub-network. Every instantiation expects as input the input sequence next element x_i and the output of the previous instantiation s_i, usually referred as the recurrent state. As output it generates the output sequence next element y_i and the state for the next instantiation s_{i+1}.
Let $\cdot$ the vector concatenation operator then we can characterize a Recurrent Neural Network with the recurrent equation:

$$s_{t+1} \cdot y_t = \mathcal{N}(s_t \cdot x_t) \tag{24}$$

It is arguable that $\mathcal{N}$ by itself is dealing with structured data, it ingests and outputs two different vectors. Indeed it is like that, however since their size is constant in practice we can treat them as a single concatenated vector. The kind of structured data we are considering are those whose shape change from one sample to another and thus can not be concatenated into a fixed size vector.

The recurrent equation in 24 outlines the consistency constraint required in the sequential model as depicted in Figure 6: the shape of the state have to be constant, it has to be the same size both when employed as input and when generated for the next step. For the same reasons note the artificial initial state

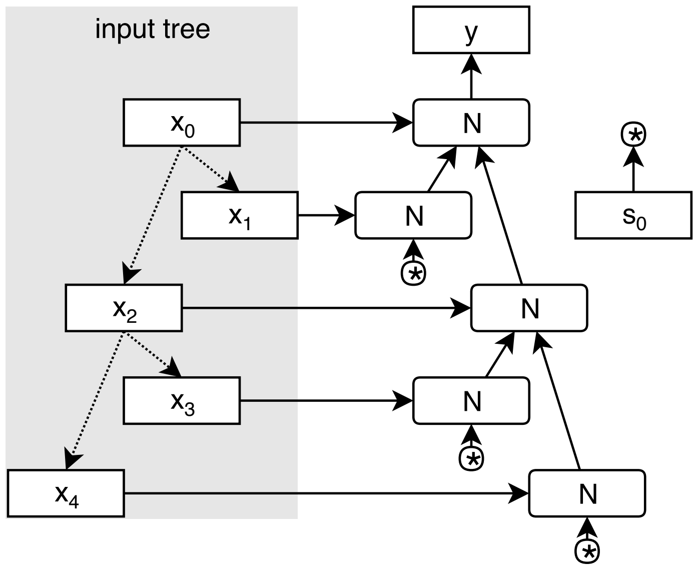

Figure 7: An example of how a Recursive Neural Network can be realized. It maps a single value y from five vectors arranged in a tree structure. Dotted arrows express parent-child relationship, whole solid ones denote the information flow. The circled asterisk is used for readability in place of the initial state s_0. Note that N can not be a simple feed forward network.

s_0, usually set to the all zero vector.

From this overall architecture many sequential models have been developed, for instance: Simple Recurrent Networks [1], [2] employ Multi-Layer Perceptrons as recurrent sub-network; Gated Units, such Long Short Term Memory [3] and Gated Recurrent Unit[4], are more involved and effective sub-networks which control how the information flows, learning to dynamically balance the contribution of the input w.r.t. to the state; Bidirectional Recurrent Neural Network [31] processes the input sequence in both directions at the same time allowing to better capture certain kinds of dependencies; Echo State Network [32] employs a sub-network with randomly generated recurrent connections and surprisingly it turned out to be a really effective approach; Sequence to Sequence [33] models employ a schema resembling an autoencoder to learn mappings among sequences possibly differing in their length.
In order to appreciate how powerful these models are consider that many state of the art results of the last few years for Natural Language Translation employed altogether most of the above approaches [33]–[35].

Recursive Neural Networks generalize the idea underlying Recurrent Neural Networks to more complex data structures [36]–[38]. With the same principle a finite number of sub-networks can be assembled in arbitrary acyclic topologies. Figure 7 shows an example compressing a tree-structured sample into a single vector.
Note how the network topology of Figure 7 closely matches the input sample

structure, no artificial dependencies are introduced. Note also that the same sub-network N is employed multiple times in a weight-sharing setting, sometime with one inpus and sometime with two inputs.
For what we have seen so far this is not possible using only a simple feed forward network, indeed due to the more complex and variable structure of samples some more care is required in order to ensure the compositionality of sub-networks. In general it is achieved employing many and more powerful sub-networks, in such a way that a consistent network can be assembled for every possible topology required by the input sample. For example a sub-network realized as a Recurrent Neural Network would be able to deal with a variable number of ordered inputs.

Let $\mathcal{N}$ be a sub-network capable to deal with a varying number of fixed size inputs, be $T = [x : t_1, \ldots, t_n]$ a tree rooted in a node with associated a value x and having n children sub-trees $t_1, \ldots, t_n$; we can then formalize the recurrent equation characterizing a Recursive Neural Network as the one of Figure 7 as:

$$y = \mathcal{N}^*(T) = \mathcal{N}^*([x : t_1, \ldots, t_n]) = \mathcal{N}(x, \mathcal{N}^*(t_1), ..., \mathcal{N}^*(t_n)). \tag{25}$$

2.3 Related works

Most common scenarios where Recursive Neural Networks with a tree-shaped composition of sub-networks find their use come from Natural Language Processing applications. Usually in such applications samples are trees augmenting natural language sentences with some additional information, for instance structuring them in their constituency or dependency trees. See Figure 9 and Figure 8 for an illustrative example.

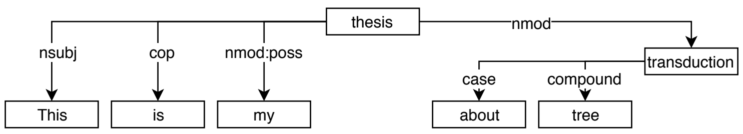

Figure 8: Examples of a dependency tree structuring a sentence accordingly to semantic relationships of its components [39].

In particular there are several models exploiting trees in their input pipeline rather than generating them. For instance [40] introduces a sentiment analysis system working with tree-shaped sentences, the computation reflects such structure so the sentiment of an element is computed starting from its children constituencies; in [41] a question answering system is built using sentences structured in their dependency trees; [42] learns to map nodes of a sentence constituency tree to a semantic vector-matrix space, vectors represent a point in the semantic space while matrices express semantic transformation applied to other close constituents; in [43] the classic attention mechanism is extended to be consistent with the input tree-structure rather than being applied flat to all the nodes; [44] introduces a Neural Machine Translation model exploiting structure information in the input sentences, i.e. constituency parse tree; [45] and [46] extend Long Short Term Memory Networks to works on tree-shaped

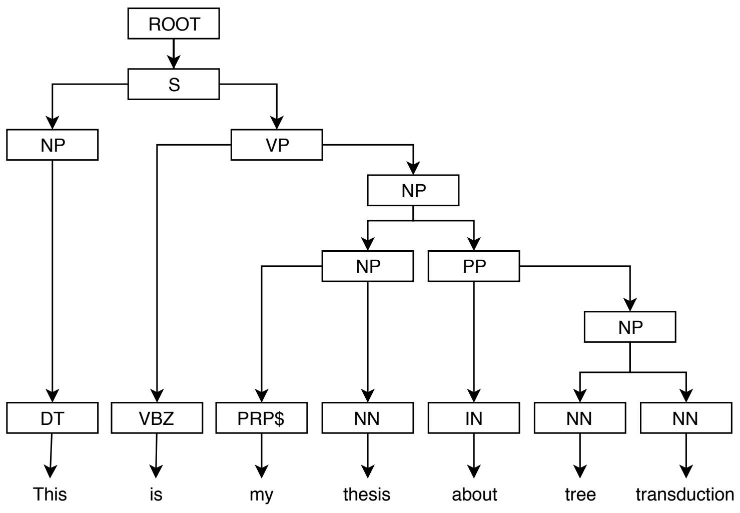

Figure 9: Example of a constituency tree augmenting a sentence with structure accordingly to some linguistic model. Part of Speech tags come from Penn Tree-Bank projects [47], [48].

computation graph.

The symmetric task of generating a tree is more involved since the structure of the sample has to be chose by the network rather then being provided along with the input. Fewer examples are present in the literature, for instance: [49] generates dependency trees along with sentences, while [50] generates constituency trees; [51] translates an input question expressed in natural language to a corresponding tree-shaped logical form; [52] generates trees from a flat embedding with a doubly-recurrent neural network; [53] produces a linearised constituency tree starting from sentences; [54] learns to automatically build parse trees from sequential data with no a priori annotations about their structure.

Some other use cases are instead related to programming languages: programs are often represented in the form of Abstract Syntax Trees. For instance: [55] introduces a model which generates programs from natural language descriptions; [56] and [57] present models which perform program induction, i.e. to generate a program satisfying some specifics provided as a set of input-output examples; [58] introduces a general model to cope with structured output; [59] performs tree-to-tree translation of source codes.

None of the aforementioned models is however a generative model, they can learn to generate a tree associated to a particular input but they can not arbitrarily generate new unseen but likely trees.
Despite the popularity of Variational Autoencoder (VAE) very little results can be found when looking for tree-structured data applications. For instance: [60] presents a model with variable sized latent space applied to tree-structured arithmetic expressions and first-order logic proof clauses; [61] explicitly employs grammars to constrain the generated trees, tested on symbolic regression and

molecular synthesis; [62] and [63] present VAE models that achieved state-of-the-art performance on molecular synthesis related tasks thanks to the explicit constrains imposed in the generated structure.
An interesting aspect emerging from the above results is how the employment of some form of constraint on the output structure might benefits performance. In general the naive approach may generates samples which are not considered valid in the actual target domain. For instance consider molecular synthesis or program source synthesis scenarios, or simply think of an arithmetic expression tree having an operator as a leaf. Indeed a quite common approach shared by many of the presented works is to employ, more or less explicitly, some form of grammar to constrain the output structure to be closer to the actual target domain.

To the extent of our knowledge there are not yet results on Variational Autoencoders employed to perform tree-to-tree Neural Machine Translation tasks. Setting aside our interest in tree-structured data, only considering VAE application to Natural Language Processing tasks, there are still quite few results, particularly compared to the amount concerning images or speech applications. Possibly it is because it looks like to be more challenging to achieve state-of-the-art performance [64].
See [25] for a VAE based model for sentence compression directly employing as latent space a discrete language model; [65] introduces a VAE with convolutional encoder and deconvolutional decoder aiming at text generation tasks; [66] presents a VAE model able to learn an effective language model employing Dilated Convolutional Neural Network in the decoder; in [67] a model for seq-to-seq neural machine translation is presented; while a dialogical model is introduced in [68].

In light of the just cited results, and considering the potential benefits of employing all the available structural information, we now investigate a deep learning model for unconstrained tree transduction. Designing and implementing an ad hoc architecture.
Although we did not tackle seriously tree-to-tree Neural Machine Translation yet, it is the use case that drove the realization of our model; it allows us to reason in a really general setting.
Specifically we want a model capable of realizing unconstrained tree transductions. This involves both to generate and to embed tree-structured data with no requirements on their structural relationship. Moreover to gain more generality we consider the mapping to be probabilistic.
More formally we want to learn a generative process governed by some joint distribution $P(Y|X)$, where both X and Y are tree-structured data. In particular the transduction that we want to learn are those $X \rightsquigarrow Y$ induced by the joint distribution.

Among the possible model frameworks with which we might have tackled the task we looked for the one with the simpler and computationally lighter training procedure: Conditional Variational Autoencoders.
As we will show in the next sections dealing with trees it is challenging by it self, both in terms of model execution time and hyper-parameters selection. Therefore we tried to keep other aspects simpler. In particular we avoided all those

models which training procedures involve expensive Markov Chain Monte Carlo based algorithm [69], [70], opting for more efficient gradient-based training algorithms. The most prominent ones we could have choose among were VAEs and Generative Adversarial Networks, we preferred the formers since they have a much simpler training procedure, despite in image generation tasks they have shown a bit less accurate results.
This is just the first of many compromises we had to make in order to keep the obtained model practically employable. In Section 4 we better investigate the performance of our model, but first let us introduce in the next sections the general Conditional Variational Autoencoders framework and the original extension we made to make it capable of dealing with tree-structured data.

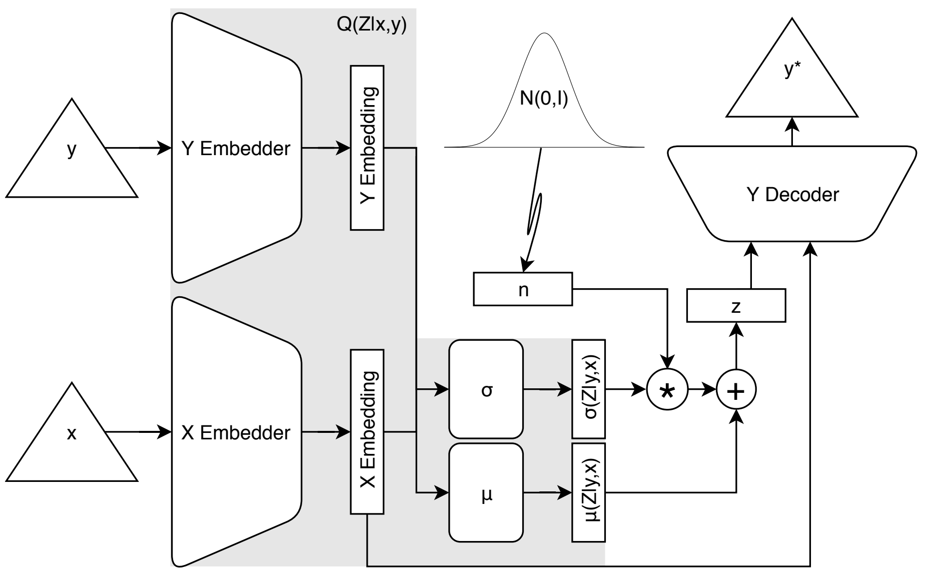

Figure 10: Overall architecture of our Conditional Variational Tree-Autoencoder. Rounded-corner shapes are neural network modules, triangles represent tree-structured data while rectangles are vectors, the curly arrow is the sampling operation.

3 Conditional Variational Tree-Autoencoder

The model we developed aims at learning a generative process realizing unconstrained tree transductions. In other words we want to learn to approximate a joint distribution $P(Y|X)$ from examples, where both X and Y are tree-structured random variables. Once modelled the distribution we additionally want a feasible procedure allowing us to sample from $P(Y|x)$ in order to be able to practically perform the probabilistic mapping $X \rightsquigarrow Y$ induced by the joint distribution $P(Y|X)$.

Our problem statement quite closely matches the capabilities of Conditional Variational Autoencoders, indeed from a high-level perspective our model does not differ much from the classical architecture as depicted in Figure 4. The ability to deal with tree-structured data is totally hidden into some encoding and decoding modules projecting data back and forth between structured and flat domains. Conceptually changing these modules is enough to be able to handle different kinds of data with the same overall architecture.

We set ourself in the usual scenario described and motivated in Section 2.1.1. As latent prior we employed $\mathcal{N}(Z; 0, I)$, an isotropic multivariate standard normal distribution and we modelled the approximated posterior $Q(Z|x, y)$ as an isotropic multivariate normal distribution $\mathcal{N}(\mu(Z|x, y), \sigma(Z|x, y) * I)$. Due to the difficulties in modelling probability distributions over tree-structured data we maximized the marginal data likelihood $P(Y|x, Z)$ stochastically on some sampled $z \sim Q(Z|x,y)$ rather than on the whole approximated posterior distribution.

The main building blocks of our Conditional Variational Tree-Autoencoder are depicted in Figure 10. Given an input pair $< x, y >$ sampled from the joint distribution $P(Y|X)$ the first processing step consists of embedding each of the tree-structured elements into more manageable flat vectors by means of two dedicated networks, the embedders. Their architecture is further investigated in Section 3.1.
These embeddings are used as inputs for the networks implementing $Q(Z|x,y)$, specifically in our scenario this is achieved by means of two separated networks computing respectively the mean vector $\mu(Z|x,y)$ and the covariance diagonal vector $\sigma(Z|x,y)$ of the multivariate isotropic normal distribution modelling the posterior. From this approximated posterior one or more codes z are then differentiably sampled and fed along with the x embedding to the output decoder, trying to reconstruct the original y.

Thanks to the reparametrization trick our model is differetiable end-to-end and thus can be trained with classical stochastic gradient descent techniques. In this setting our training procedure is the one described by [5] as Auto-Encoding Variational Bayes algorithm in the variants employing the Stochastic Gradient Variational Bayes estimator to Monte Carlo maximize the expectation term of the loss as obtained in Section 2.1.1:

$$\mathcal{D}[Q(Z|x,y)\|P(Z|y,x)] - \mathop{\mathbb{E}}_{z\sim Q(Z|x,y)}[\log P(y|z,x)]. \tag{26}$$

Rather than computing the expectation on the whole Q we compute it only for a finite number of sampled codes $\tilde{Q} = \{z_1, \ldots, z_L\}$, Moreover since we are not able to differentiably compute the likelihood, we employ a substitute to maximize it:

$$\mathcal{D}[Q(Z|x,y)\|P(Z|y,x)] + \mathop{\mathbb{E}}_{z\in\tilde{Q}}[\mathcal{L}(y,y^*)]. \tag{27}$$

At this point the loss function of Equation 27 can actually be computed: the Kullback-Leibler Divergence term in our settings has an analytical solution as showed in 2.1.1, the expectation is performed on a finite set $\tilde{Q}$, $\mathcal{L}$ is some arbitrary differentiable error function measuring how far is the reconstructed y^* from the original y. More detail on the latter are provided in Section 3.1.4, after we better characterize the kind of trees we are dealing with.
The final bit to get the loss function as we employed it is to add a scaling hyper-parameter $\delta \in \mathbb{R}^+$ and to extend the single-sample loss to a mini-batch version:

$$\mathop{\mathbb{E}}_{<x,y>\in M}\left[\delta\,\mathcal{D}[Q(Z|x,y)\|P(Z|y,x)] + \mathop{\mathbb{E}}_{z\in\tilde{Q}}[\mathcal{L}(y,y^*)]\right] \tag{28}$$

The scaling hyper-parameter δ is a regularization coefficient [19] balancing the objective components. Weighting too much the divergence term might lead to a model having an almost normally distributed $Q(Z|x,y)$ whose sampled encodings however brings very little useful information to the reconstruction process. Vice versa the encoding might leads to a good reconstruction but being the latent distribution $Q(Z|x,y)$ arbitrary far from being normally distributed we

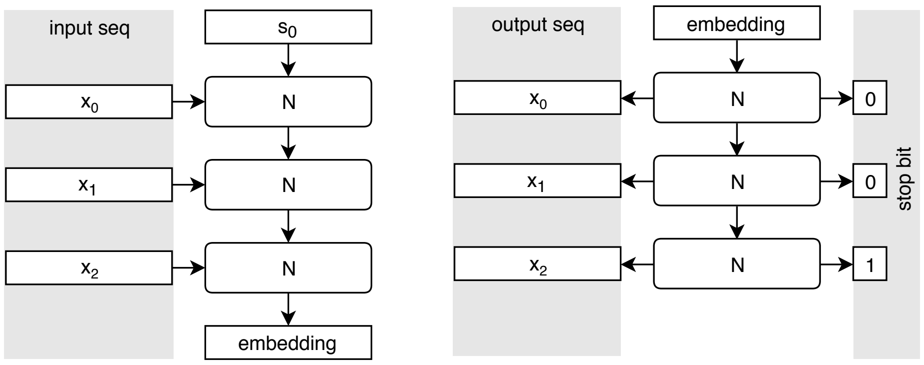

Figure 11: Example for a Recurrent Neural Network with sub-networks structured as the sequential sample at hand. Embedding case on the left, generation on the right.

might be unable to properly sample from it in order to generate plausible new samples.

An interesting aspect worth to be noticed is the nested stochastic approximation employed by Equation 28. At the outer level it is employed to make a big dataset manageable in the classical mini-batch approach, rather than minimize the target on the whole dataset at every iteration we employ only a random subset M. Inside this mini-batch stochastic approximation we find another stochastic approximation, more specifically related to the Variational Autoencoders: rather than computing an expectation on the whole approximated posterior $Q(Z|x, y)$ we approximate it employing a finite subset $\tilde{Q}$.

3.1 Tree Modules

Key components of our Conditional Variational Tree-Autoencoder are the modules dealing with the embedding and the generation of tree-structured data. They are designed as Recursive Neural Networks and architecturally are the most involved components of our model.

In general we would like to have some sub-networks that can be assembled together accordingly to the structure of the tree at hand. As for sequential models we obtain a sequence of sub-networks long as the sequence at hand, with trees we would like to obtain a computation graph shaped as a tree of sub-networks resembling the structure of the sample at hand. Figure 11 shows an example depicting the aforementioned phenomenon in the sequential case, while a simple example for tree-structured samples is shown in Figure 12. A closer look at the latter figure outlines some facets that with sequences might not be explicitly noticed due to their simplicity but should be directly faced when dealing with trees. Specifically we are referring to the higher structural variability and the compositionality of sub-networks.

When dealing with sequences the structural variability rises only from the se-

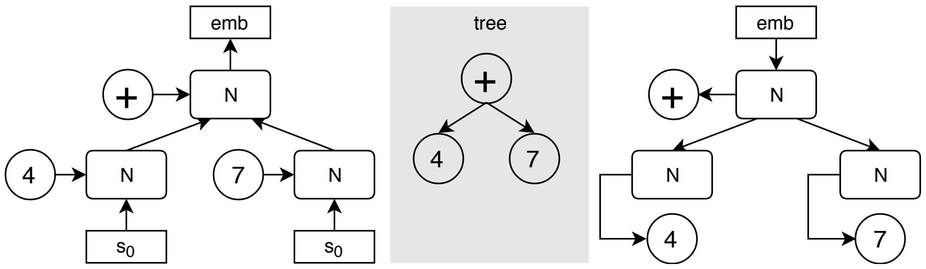

Figure 12: Example for a Recursive Neural Network with sub-networks structured as the tree sample at hand. Embedding case on the left, generation on the right. Note that we omitted all the parts concerning the structural choices performed during the generation since they depends from the employed model: we will better investigate it ahead in this section.

quence length. In the embedding case we can simply apply the recursive sub-network as long as there are elements of the sequence to process and then just stop. When generating it is enough to have an extra output telling us when to stop, see Figure 11 for an illustration.
Trees structure is more involved, it is affected by all the branches structure along with the values possibly associated with nodes. It is not straight forward to encode in the model a way to make choices about the generated sample, analogously to the sequential model stop bit. In fact in the example of Figure 12 we totally omitted that part. In order to properly design sub-networks able to make such structural choices we need some tool to reason about the possible structures a tree can have.

Another facet that gets more involved due to the higher structural variability regards compositionality.
In the sequential model we instantiate a copy of the sub-network, in a weight-sharing setting, for each element of the input sequence, see Figure 11. Every sub-network has an input for the state that can be the initial one s_0 or an output from the previous instantiation. In order to ensure compositionality the state shape a sub-network expects as input should be compatible[2] with the output shape a sub-network provides as next state and with the initial one s_0. In the sequential model this is achieved by simply having the state input shape and state output shape to match.
Looking at the tree embedding case of Figure 12 you can notice how the same sub-network has a single recursive input s_0 when employed in the leaves and has two recursive inputs when employed as the root, this breaks compositionality as defined for the sequential case.
We need to design a Recursive Neural Network whose sub-networks can be composed in all the ways required by the trees in the domain at hand, therefore with a variable number of children and possibly with a value associated.
Indeed we will not focus on the specific internal details of any sub-network, which might be application dependant, we are rather interested only in the

[2]In the usual case *compatible* means to have the same exact shape

sub-networks input and output shapes and in how to properly assemble them accordingly to all the possible samples. As before we need to better characterize trees structure.

3.1.1 Trees Characterization

In order to be able to manipulate trees we need to have some tool to reason about them, their structure and the value their nodes can have. Some knowledge about the domain of the trees we are dealing with is fundamental to properly instantiate the Recursive Neural Networks components. We need to ensure that we are able to dynamically assemble a proper network for each of the possible trees belonging to the domain.
Conceptually, it would be enough to know the domain of the values possibly associated to nodes. With this limited knowledge we would be able to design several strategies to deal with trees of arbitrary structure; nevertheless additional knowledge might be really helpful to foster the learning process.
On one side additional knowledge can be exploited to bound the tree domain employed by the model, the narrower and closer to the real data domain it is the easier it will be the learning process. A wider domain would allow to generate trees which do not belong to the real data domain.
Another interesting aspect to consider is related to node semantics. Additional knowledge may allow to closely match the nodes different semantics while building the corresponding network, having different networks specializing on different functions.

For example consider the domain of Arithmetic Expression Tree (AET) having internal nodes labelled with $+$ or $-$ and with leaves labelled by digit in $[0, 9]$. All is needed to naively design an encoder and a decoder is the unified domain of all values $V = [0, 9] \cup \{+, -\}$. Figure 13 shows the naive architecture of such an embedder and a decoder exploiting minimum knowledge from the real tree domain.
The encoder sub-network merges all the children of a node into an embedding, for each children it takes its value as a one hot encoding, its embedding and a recurrent state initially set to the zero vector. Symmetrically the decoder network generates all the children of a node, starting from the parent embedding at each step it generates an updated embedding to generate next children and the embedding and value of the current child, note that the generation will keep going accordingly to the output k.
Despite, the networks of Figure 13 can be assembled accordingly to every possible tree in the AET domain, they can also generate invalid trees such those having an operator in the leaves or those having more than two children on a subtraction node. Moreover, even if they intuitively look like to be computationally capable of representing AET trees ,consider that the same embedder network merges the children of a node whether it represents an addition or a subtraction, a specialized network per operation might better capture its semantic fostering the learning process.

Motivated by the aforementioned reasoning we decided to employ a tree characterization a bit more restricting than the minimal one requiring only the knowledge of a unified values domain. In general the kind of trees we are tar-

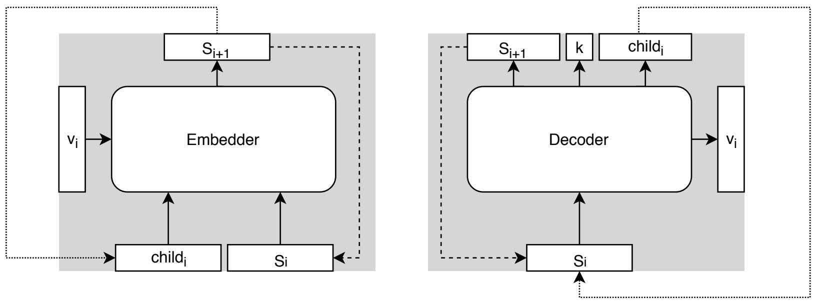

Figure 13: Naive architecture for encoding and decoding trees. The greyed area recurs over the children of a node keeping an intermediate state into S_i, at each step it embeds or generates one child value and its embedding. The decoder k output is a bit which determines when to stop generating children. The dotted lines outside of the greyed area are the recurrence occurring between children and parents, rather than children of the same parent as the dashed ones. As usual as initial recurrence state we employ a zero vector when encoding the first children of a node, while in the decoder we initialize the state with the parent embedding. Note that in this setting we need an artificial root node with no value and with a single child being the original root.

geting are ordered trees with an arbitrary number of children and whose nodes may have a value associated. Children position is not modelled, only order is relevant to us, thus children are always contiguous.

In such a setting we characterize some particular family of trees, e.g. AET, characterizing their nodes individually, employing the concept of node type we group similar nodes occurrences together. Every node has a type which restricts its behaviour, for instance in the number of children or in the associated value. On one hand it makes us able to explicitly give different identities to nodes that intuitively have a different semantic, in order to possibly employ specialized sub-networks, such as for the two operators $+$ and $-$. On the other hand it allows us to bound the domain of trees we are able to generate to be closer to the real one. For instance all the subtraction nodes can be forced to have exactly two children.
This characterization however is not enough restrictive to generate only proper trees. Describing nodes only individually we are not predicating on how different types of node can combine; there no constraints on parent-child node types therefore we can still generate for instance AET with an operator in a leaf.
Although we may expand the model expressiveness, in order to be able to describe narrower domains more closely matching the actual target domain, we decided to leave it as a possible future work. In many real world scenarios it is not easy to characterize how different node types can be composed; the rules might be arbitrary complex or even unknown. This consideration along with the considerable impact on the implementation is what made us simplify the tree characterization, neglecting for the moment parent-child node type constraints. The only kind of constraint we employed is concerning the root node type.

Ignoring parent-child type constraints, all we need to practically characterize the family of trees our model will actually deals with is a finite list of node types:

$$\text{TreeType} = \{\text{NodeType}_1, \dots, \text{NodeType}_k\}$$

Node Types Each NodeType_i is characterized by the following information:

> **Id** A unique string identifying the node type;
>
> **IsRoot** $\in \{\text{True}, \text{False}\}$ whether an instance of this node type can appear as root node.
>
> **ValueType** As introduced in the next paragraph, it characterizes the domain of values associated to the instances of this node type, a special value $\text{ValueType}_\perp$ denotes that no values can be associated to the instances of this node type.
>
> **Arity** Constrains the number of children of the instances of this node type, they can be a fixed number or varying in a range:
>
> > **FixedArity** a node have exactly n children, in the case $n = 0$ all the instances of this node type can appear only as leaves;
> >
> > **VariableArity** a node can have an arbitrary number of children in the range $[n, m]$, if $n = 0$ the node instances of this type may appear also as leaves. Even if it might makes sense we do not consider the case where $m = \infty$. As shown ahead in this section in some case we need a finite upper bound to the possible children to be able to build a proper recurrent neural network model.

Value Type Some node type can have a value associated, this have to be characterized as well in order to handle it properly. Different node types may have different value types while all the instances of a node type must share the same value type. A ValueType describes the domain of values and how to practically deal with them by means of the following information:

> **Id** A unique string identifying the value type;
>
> **Abstract Domain** The domain where abstract values belong to. Those values on which we would like to reason about but are possibly not well suited to be fed to a neural network. For instance an English word in some Natural Language Processing application;
>
> **Concrete Domain** The domain where concrete values belong to. Those values which we actually use during the computations and that are more well suited to be fed to a neural network. For instance a vectorial word embedding instead of an English word, or a one hot vector instead of a categorical value;
>
> **Mappings** Some function which allow us to map values from the abstract domain to the concrete one and vice versa.

Arithmetic Expression Trees Reconsider the Arithmetic Expression Tree example of Page 23, those having only addition and subtraction with possible leaves value ranging from 0 to 9. The characterization in our formalism is shown in Table 1.

Id	**IsRoot**	**ValueType**	**Arity**
AddNode	True	ValueType$_\perp$	Variable(2,10)
SubNode	True	ValueType$_\perp$	Fixed(2)
NumNode	True	NumValue	Fixed(0)

Id	**Abstract D.**	**Concrete D.**	**Mappings**
NumValue	$\{0, 1, \ldots, 9\}$	1 of K encoding	*

Table 1: The full characterization in our formalism of the Arithmetic Expression Tree of Page 23, employing only $+$ and $-$ on digits ranging in $[0, 9]$. * Mappings are the expected ones: given a digit its concrete representation is its one hot encoding, while for the vice versa the abstract value is obtained as the index of maximum value in the concrete value vector.

Addiction and subtraction are modelled as two different node types so we could constrain their arity independently. Subtraction nodes can and must have exactly two children while addiction nodes can vary between two and ten (note that this upper bound is arbitrary). Being the identities of the two operators distinguished we can specialize our network to differently deal with them, possibly having a dedicated sub-network for each one as shown next in Section 3.1.2. Modelling digits as a node type makes possible to explicitly constrains NumValue to be the only possible node type appearing as a leaf, it is the only node type having a zero in the arity definition.

3.1.2 Embedder Modules

Given a characterization of a family of trees in the way just described in the previous section we are now able to properly construct a Recursive Neural Network. Assembling sub-networks in a proper, compositional, way in order to map every possible tree in the domain into a fixed size vector.
This is the most popular setting when dealing with trees: we want to extract some information from the structured input and in order to do so we process the tree bottom-up, compressing the structured information we are interested in into a more manageable vector, i.e. the embedding. The computation graph is dynamically built assembling sub-networks accordingly to the sample structure, the embedding of a parent node is computed from the embeddings of its children and its value.

In order to obtain compositionality we employ the idea of an embedding space $\mathcal{E}$ containing all the embeddings of the trees in the domain at hand and also all the sub-trees composing them, single nodes as well.
This way we can think of every sub-network as realizing a projection of all its input into this space and we can build dedicated sub-networks to project specific leaf node with a certain ValueType, to merge the children embeddings for a particular NodeType into a single embedding or to augment a node embedding

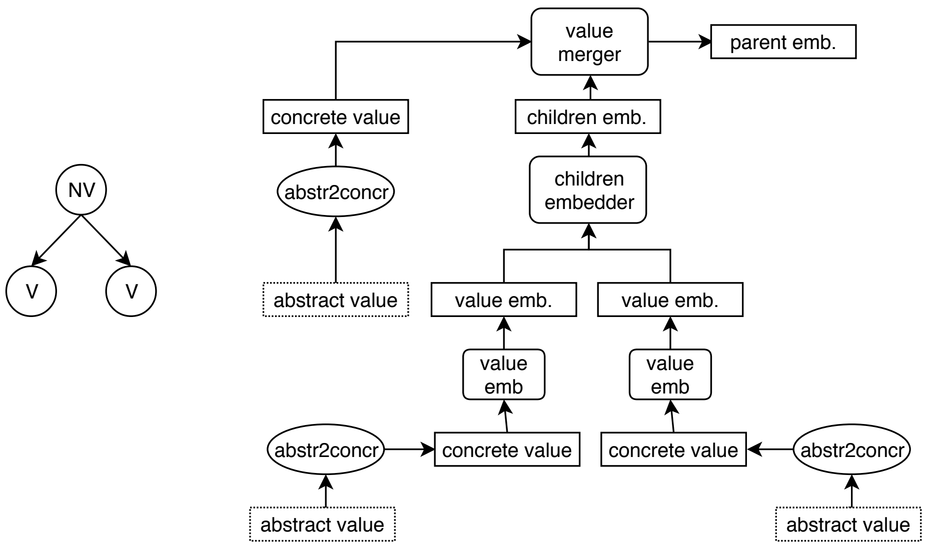

Figure 14: Overall architecture of the unfolded network for the embedding of an example tree.

for a given NodeType with its associated value.

All such kinds of sub-networks can be appreciated in the unfolded network of Figure 14. It realizes the embedding of a simple tree with only a root with two children. Proceeding bottom-up the computation starts from the leaves: accordingly to the mapping defined in the respective ValueType the abstract values are mapped to the concrete domain, than can then be fed to the value embedder sub-network in order to obtain a projection into the embedding space. All the children embeddings of the root node are then merged into a single embedding and then augmented with the associated value by means of the value merger.
In general we have two sub-network kinds: value embedders and node embedders. Respectively dealing with values and merging sub-trees embedding into a single embedding.

Value Embedders

Leaves Embedder The bottom-up computation starts projecting leaf values into the embedding space. For every pair <ValueType, NodeType> such that a node admits zero as a valid arity we build a sub-network projecting leaf concrete values into the embedding space $\mathbb{V}_i \rightarrow \mathcal{E}$. Note that we are assuming that leaves have always an associated value, a constant ValueType can be used in the case of leaves with no value.

Value Merger Whenever in our bottom-up traversal we encounter an internal node with an associated value, we first compute the node embedding and then augment it with the node value by means of a dedicated sub-network. For every NodeType having a ValueType associated we build a

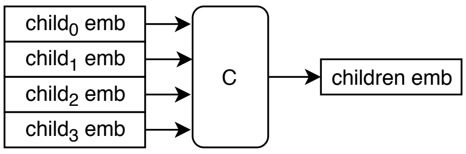

Figure 15: Embedder for a node whose NodeType has a fixed arity, having always the same input size a feed forward approach can be employed.

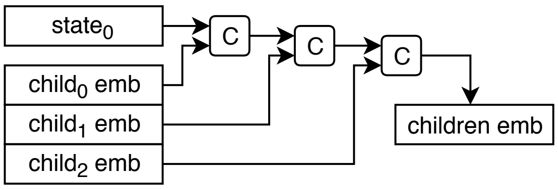

Figure 16: Recursive strategy to embed children into a single parent embedding. Note that the C sub-network is employed in a weight-sharing setting in all the instances.

value merger sub-network merging the node embedding and its value into a new embedding $\mathbb{V}_i \times \mathcal{E} \to \mathcal{E}$

Node Embedders The main kind of sub-networks involved in the embedding of a tree are the node embedders, they are employed to reduce a set of children embeddings into a single embedding representing their parent. Here is where some care is required to ensure compositionality for all the possible trees in the input domain.
For every NodeType a sub-network is created. In the case it has a fixed arity, i.e. all its instances have the same number n of children, no particular care is required, as depicted in Figure 15 every sub-network matching the signature $\mathcal{E}^n \to \mathcal{E}$ will work. For instance considering some proper feed forward network $\mathcal{C}$, we can compress all the children embeddings $c_0 \ldots c_{n-1}$ into a single embedding c simply as:

$$c = \mathcal{C}(c_1 \cdot ... \cdot c_{n-1}). \tag{29}$$

For NodeType with a variable arity it is a bit more involved since we do not have a fixed size input. Two different strategies have been developed:

Recurrent Strategy A quite natural strategy is to encode all the children with a sequential Recursive Neural Network, it easily adapts to different sequence length. However it has some drawbacks due to the fact that it creates a really deep computation graph. The deeper the network the more difficult might be to have the correct information bubbling up to the final embedding, or alternatively to have the gradient information reach the first layers. Secondly it slows down the computation introducing dependencies which might not be necessary, in fact the computation depth is linear in the number of children.
For instance consider the recurrent network $\mathcal{C}$ having as input the concatenation the actual input c_i and the recurrent state s_i, to embed all the children

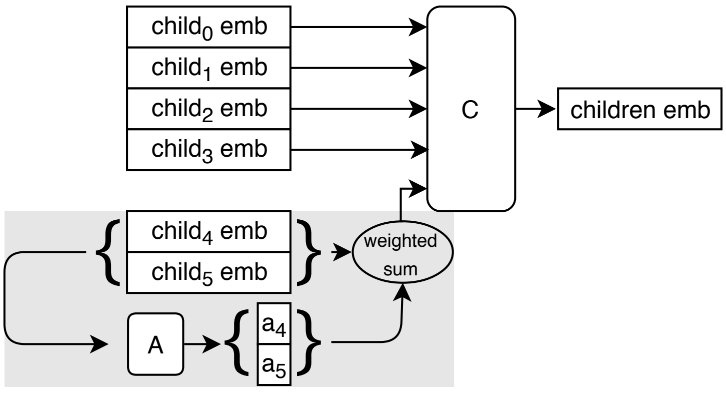

Figure 17: Flat strategy to embed children into a single parent embedding. Only the first children up to the *curtarity* are directly fed to the encoding sub-network, additional children are merged in a single embedding with an attention mechanism. When there are to few children missing embeddings are replaced by the all-zero vector.

embeddings in a single vector we need to unfold $\mathcal{C}$ among all of them:

$$c = \mathcal{C}(c_{n-1} \cdot s_{n-1}) = \mathcal{C}(c_{n-1} \cdot \mathcal{C}(c_{n-2} \cdot ... \cdot \mathcal{C}(c_0 \cdot s_0))). \tag{30}$$

In Figure 16 are depicted the schematics of such an unfolded sub-network for a concrete example. Note that the embedder C is used in a weight-sharing setting in all of its occurrences sharing the same parent NodeType.

Flat Strategy In the seek of a more computationally efficient approach we developed an alternative strategy which trades of some accuracy in the children information extraction process for a high parallelizable and optimizable computation. The rationale is that in many real world scenarios most nodes have few children and few nodes have a lot of children. Thus we choose a *curtarity* after which we approximate the computation for any other children, accepting to possibly loss some information.
As depicted in the example of Figure 17 only the first *curtarity* children are directly and in parallel fed to the embedder sub-network, additional children are lossy merged into a single embedding by means of an attention mechanism. A sub-network $\mathcal{A}$ estimates the relevance of every additional children, such coefficients are scaled to sum up to 1 and then used to merge all the additional children in a single embedding by means of a weighted sum. When instead children are less than *curtarity* threshold we employ the all-zero vector in place of missing embeddings in order to have an input consistent with the embedder sub-network signature.

More formally let us consider a node having n children and a flat strategy employing a cut arity of $k < n - 1$, then additional children embeddings are summarized by means of the attention network $\mathcal{A} : \mathcal{E} \to \mathbb{R}$. A single value expressing the relevance of every extra children is computed, then they are normalized with a softmax function as:

$$a_i = \frac{e^{\mathcal{A}(c_i)}}{\sum_{j=k+1}^{n-1} e^{\mathcal{A}(c_j)}} \qquad k < i < n \tag{31}$$

such coefficients are then employed to compute a lossy summarization of the additional children exceeding the *curtarity* as:

$$c^{(a)} = \sum_{j=k+1}^{n-1} c_j * a_j \tag{32}$$

then the final embedding compressing all children into a single vector can be computed as in the fixed arity case:

$$c = \mathcal{C}(c_0 \cdot ... \cdot c_k \cdot c^{(a)}). \tag{33}$$

Depending on the kind of trees we are dealing with the structural bias induced by this approach might turn out to be useful or harming. In the setting where children semantics change with respect to their position this approach might better capture different children meaning since it can specialize the sub-network parts corresponding to each input. With the recurrent strategy the same network has instead to learn to deal with all the semantically different children accordingly to their order in the sequence. Vice versa if all the children have an equal semantic, potentially they can be shuffled without changing the overall semantic of the tree, the flat strategy need to learn the same thing for all its sub-networks, while the recurrent one might benefit from its structural bias.

3.1.3 Generator Modules

The symmetric setting where we want to generate a tree starting from a single embedding vector encoding all the relevant information is a bit more involved than the embedding case. Proceeding top-down we generate the target tree from the root to the leaves. The tree structure however has to be chosen along with the values rather than being provided as input as in the embedding case. Exploiting the tree family characterization introduced in Section 3.1.1 we can construct a Recursive Neural Network whose sub-networks can be assembled in a proper, compositional, way and that not only generate the values associated to the target tree but also choose how to structure it.
Indeed the computation graph obtained while generating a tree is even more dynamic then in the embedding case. In fact it is dynamically built during the evaluation of the graph itself: the structure of a generated sub-tree depends on the evaluation of the computation graph associated to its parent. In the embedding case the computation graph may change with every sample as well, however once seen the input tree we know the whole structure of the computation graph without the need to evaluate any part of it.

In order to obtain compositionality we use the same approach employed in the embedder module. All the trees, sub-trees and leaves embeddings belong to

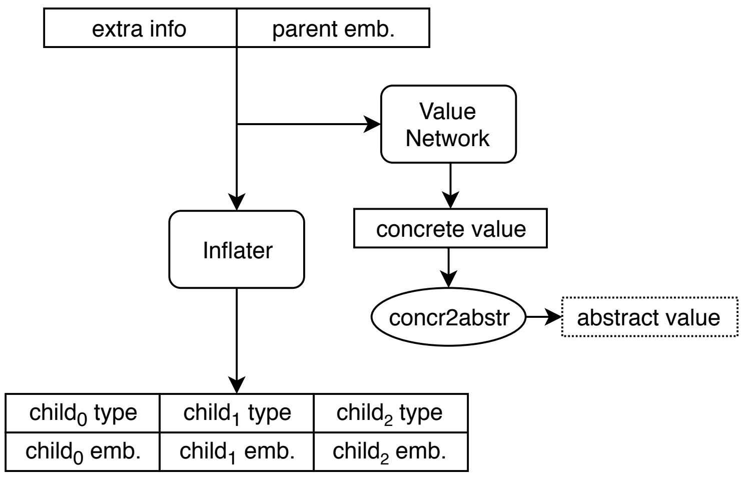

Figure 18: High level architecture of a single step of a generator. Starting from the parent node embedding along with some extra information, such as parent type and root embedding, it generates the associated value and the children embeddings and types.

the same embedding space $\mathcal{E}$. This way we can think of every sub-networks as realizing a projection from this space into the desired output space and we can build dedicated sub-networks in order to obtain for every ValueType a concrete value from the specific domain and for every NodeType possibly having children the embeddings and the types of its children.
In practice at every generation step we possibly expand the frontier of the generated tree employing as input, additionally to the parent embedding, some extra information such as parent node type, root embedding and conditioning embedding. Depending on the kind of application we are tackling we could even employ more additional information such as some tree-structured attention on the conditioning embedding tree. This does not break the compositionality as long as the same augmentation is used for all the embeddings.

Figure 18 depicts the overall architecture of a single generation step expanding a node in the generated tree frontier. As input it takes the parent node embedding and possibly some extra information as mentioned in the previous paragraph. If the parent node type has associated a value the value network corresponding to its ValueType it is employed to generate a value, first in a concrete form manageable by the neural network then mapped to the abstract desired form by means of the mapping provided by the ValueType definition. Possibly in parallel an inflater network generates the type and embeddings for an arbitrary number of children, possibly zero if it decides this has to be a leaf node. The specific architecture of this inflater is a bit involved due to the possibly varying sized output.

Node Inflater The main kind of sub-networks involved in the generation of a tree are the node inflaters. They are employed to expand the tree frontier possibly generating a list of children embeddings and types. Here is where some

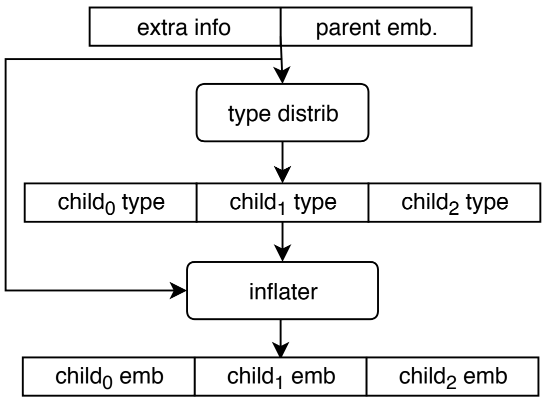

Figure 19: Simple inflater for node types with a fixed arity. Since input and output size are known and constant both types and embeddings can be generated altogether. Note that we split the generation in two step, we want to condition the embedding generation with the node chosen type.

care is required to ensure compositionality for all the possible trees in the output domain. The strategy employed to inflate from a parent node its children embeddings and types strongly depends from the parent NodeType characterization, indeed for each possible NodeType we create a dedicated sub-network. In the simplest scenario when the parent has a fixed arity, i.e. the output size is constant, even a simple feed forward neural network can be employed (no particular architecture is required as shown in Figure 19).
Note the two phases generation process: first a dedicated sub-network chooses the children types; then a second sub-network computes the embeddings also conditioned by the chosen types.

Given the parent embedding p and its associated extra information p' the sub-network $\mathcal{T}$ computes all the children types as:

$$t_0 \cdot ... \cdot t_{n-1} = \mathcal{T}(p' \cdot p) \tag{34}$$

then they are used along with the input to inflate all the children embeddings with the inflater sub-network $\mathcal{I}$:

$$c_0 \cdot ... \cdot c_{n-1} = \mathcal{I}(p' \cdot p \cdot t_0 \cdot ... \cdot t_{n-1}). \tag{35}$$

When the instances of a node type can have a variable number of children we need more involved architectures to deal with the variable sized output. As for the embedder module we developed two separate strategies:

Recurrent Strategy The usual approach to deal with a varying length sequence is to employ a Recurrent Neural Network. Figure 20 shows the folded architecture of such a recurrent strategy.
For each child a recurrence is performed. First a dedicate sub-network generates a distribution over types, augmented with an additional special type

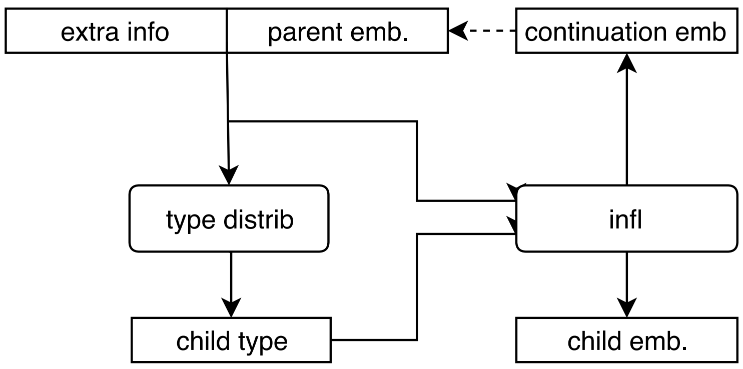

Figure 20: Inflater for node types with a variable arity employing a recurrent strategy. At every recursive iteration a child is generated, first its type and then its embedding. The process keep going as long as the special type *nochild* is not chosen. The parent embedding is used only in the first iteration, then a continuation embedding acts as the memory of the recurrent process

nochild used to stop the generation process. Then, if a child has to be generated, a dedicated sub-network is employed to generate the child embedding and a continuation embedding acting as a memory in the recurrent process and used in place of the parent embedding in the next iteration.

More formally, be p the parent embedding and p' the associated extra information, $\mathcal{T}$ is the network computing the child type distribution and $\mathcal{I}$ is the inflater network. We can then characterize the recurrent strategy with the following equations:

$$s_0 = p \tag{36}$$
$$t_i = \mathcal{T}(p' \cdot s_i) \tag{37}$$
$$s_{i+1} \cdot c_i = \mathcal{I}(p' \cdot s_i \cdot t_i). \tag{38}$$

The same reasoning we have done for the embedder recurrent strategy however holds also for this strategy, employing it we will end up building a sequential and deep computation graph with possibly not necessary dependencies. As for the embedder recurrent strategy this can lead to some drawbacks: deeper networks are harder to optimize and the introduced dependencies considerably limit the employable parallelism slowing down the computation.

Flat Strategy As for the encoder case we developed an alternative strategy seeking a computationally more efficient and possibly a more gradient friendly approach. The same key observation holds: most nodes have few children and few nodes have many children. With this in mind we designed a sub-network which up to a certain amount *cutarity* of children parallelizes the computation without losing information, while in the few cases where there are more than *cutarity* children it is still parallelizable but might loses some information.

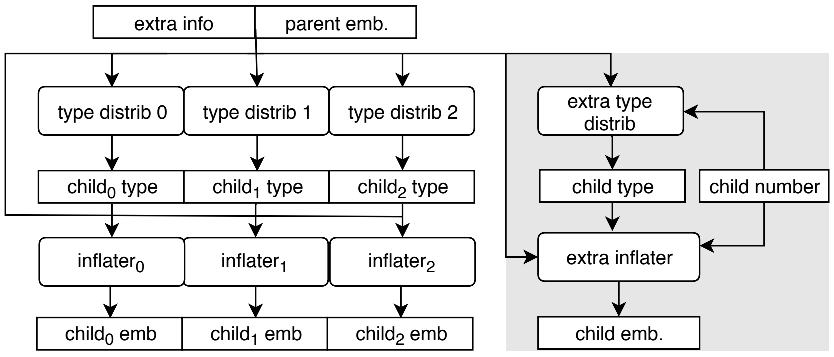

Figure 21: Inflater for node types with a variable arity employing a flat strategy, trading some accuracy for efficiency. Up to some number every child has two dedicated sub-networks computing type distribution and embedding. Extra children are instead computed all by the same two sub-networks, the greyed area, having in input a one hot encoding of the child number.

Figure 21 depicts the architecture of a sub-network employing such a flat strategy having a *cutarity* = 3. The first three children are computed in parallel by three different dedicated sub-networks, as before we first compute node type and then if the case we compute the embedding. All the remaining children are computed by the same sub-networks, children are differentiated augmenting the input with the child number encoded by means of a one hot encoding. As in the other strategy we use a special node type *nochild* to avoid to generate a child. Since in our model node children are ordered but not positional, we group together all the generated children collecting them left-to-right.

Be p the parent embedding and p' its associated extra information, be k a shorthand for the *cutarity* value, then the type of one of the children is computed as:

$$t_i = \begin{cases} \mathcal{T}_i(p' \cdot p) & \text{if } i < k \\ \mathcal{T}_k(p' \cdot p \cdot \text{onehot}(i)) & \text{if } i \geq k \end{cases} \tag{39}$$

for each of the possible children i a type is computed employing the proper sub-network: a dedicated one if $i < k$, otherwise the shared one $\mathcal{T}_k$. Indeed in this latter case it is required to feed to the network also the child number by means of its one hot encoding.
Notice however that the subscript i does not necessary match the i-th generated child, in fact we need a remapping to consider only children that have to be generated, i.e. those whose type distribution does not have the special type *nochild* as most probable value.
With $|\cdot|$ computing a set cardinality and abusing *argmax* to compute the most probable type over a distribution t, we can compute $\hat{i}$, the actual positional index leading to the generation of the i-th children, as:

$$\begin{aligned} \hat{i} = j \quad s.t. \quad & argmax(t_j) \neq nochild \\ & i = |\{i : \text{argmax}(t_i) \neq \mathit{nochild} \land i < j\}| \end{aligned} \tag{40}$$

Using $\hat{i}$ we can denote the proper sub-network and inputs to use to actually generate the i-th child embedding as :

$$c_i = \begin{cases} \mathcal{I}_{\hat{i}}(p' \cdot p \cdot t_{\hat{i}}) & \text{if } \hat{i} < k \\ \mathcal{I}_k(p' \cdot p \cdot t_{\hat{i}} \cdot \text{onehot}(\hat{i})) & \text{if } \hat{i} \geq k \end{cases} \tag{41}$$

As for the embedding flat strategy the structural bias induced by this approach might turn out to be useful or harming depending on the kind of trees we are dealing with. When children have different semantics depending on their position this approach might specialize its sub-network to capture it, at least up to *cutarity*. For children exceeding it the effect is similar to the recurrent strategy, the same sub-network is employed for all the semantically different children accordingly to their order in the sequence.
Vice versa if all the children have an equal semantic, potentially they can be shuffled without changing the overall semantic of the tree. The specialized sub-networks in the flat strategy might have to learn all the same thing while when employing the same sub-network this comes for free from the structural bias.

Value Inflater The network extracting a value from a node embedding, when the corresponding node type has a value associated, does not have any particular complication, the input and output shape are always the same. As shown in Figure 18 it takes as input the embedding of the node along with some extra information and generates a concrete value which can be mapped to the abstract value by means of the mapping provided along the ValueType definition.

3.1.4 Tree Reconstruction Accuracy

Employing an embedder and a decoder shaped as the one just introduced we are now able to build the whole Conditional Variation Tree-Autoencoder as presented in the beginning of this chapter. However to train the model we still miss a last bit: our optimization process is drove by the loss as in Equation 28 which require us to be able to compute some reconstruction error $\mathcal{L}(y, y^*)$ between the target tree y and the generated one y^*. Measuring how close two different trees are is not straight forward, as soon as they differ in their structure there is not a clear way to quantify how good is the reconstructed tree.

During our experimentations we employed some metrics to have an intuitive quantification of the performance goodness, all of them are based on the concept of Biggest Common Prefix Tree (BCPT). As exemplified by Figure 14 in order to compute the BCPT we visit the two trees in parallel starting from their roots and trying to match their nodes. Two nodes match if they have the same type and their parents match as well. If the roots do not match then the BCPT is empty, otherwise we can start the visit. At every step we try to match all the children of the current node in both trees, left-to-right we try to match the i-th child in the first tree with the i-th child in the second tree, on the matching children we continue visiting while all the sub-trees rooted in non-matching children are skipped. The BCPT is the sub-tree obtained only by the nodes we have been able to visit, the matching ones.

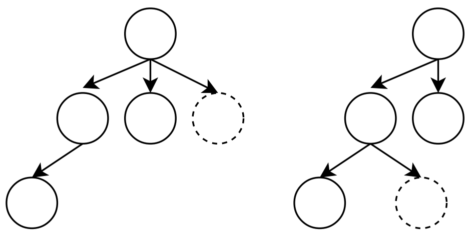

Figure 22: Example of how the biggest common prefix tree is computed. Given the two trees in figure, the BCPT is the one with solid nodes, dashed nodes does not have a matching node in the other tree. Note that we proceeded in left-to-right order and for readability we simplified the example omitting node types, in order to have two matched nodes their types should match as well.

Given the BCPT of the target tree y and reconstructed tree y^* we can compute some intuitive metrics measuring the reconstruction goodness, for instance:

- Be $|\cdot|$ the operator counting the nodes in a tree, then the amount of matched structure can be quantified as:

$$\text{acc}_s = \frac{2|BCPT(y, y^*)|}{|y| + |y^*|}$$

- To assess the reconstruction we can compare only those values on the matched nodes; the remaining of the two trees is not easily comparable, the different structures make hard to relate nodes. How to actually compute a value reconstruction accuracy depends on the value types. However consider for instance acc_{mv} to be some values reconstruction accuracy computed on the matched nodes, we can quantify the overall values reconstruction accuracy weighting it with the structural accuracy as $\text{acc}_v = \text{acc}_s * \text{acc}_{mv}$.

Although we have used such metrics to assess the performance goodness during the experiments we can not employ them in the optimization target, we want to use gradient based optimization techniques and the BCPT computation is not differentiable.

The alternative approach that we employed relies on the observation that during the training we are in a supervised setting, the structure that the generated sample has to have is known, it is the same as the target one. Therefore we could somehow cheat and generate a tree with the desired structure even if during the process the decoder would have chose a different one.
As long as we can measure how much of the structure is due to the cheat and how much of it has been actually learned this approach heavily simplifies the loss computation.

We now have two trees with the same structure and we can compute the loss regarding the structure almost independently from the loss regarding the values. Some care is needed only when we are combining them, a different scaling could weight differently their contribution. So we can split the loss in two components combined, for instance, by means of a convex combination:

$$\mathcal{L}(y, y^*) = \alpha \mathcal{L}_s(y, y^*) + (1 - \alpha)\mathcal{L}_v(y, y^*)$$

Structural Loss During the supervised generation process we cheat making always the structural choices that lead to a tree with the same structure as the ground truth. As for the Generator Module introduced in Section 3.1.3 we generate the output tree incrementally, starting from the root and possibly expanding its children recursively. At every step a new node of a certain type might be created accordingly to a generated type distribution d_i^* which includes also the special type *nochild*. In order to obtain a sample with the same ground truth structure we cheat and rather than making the structural choices based on our model generated distribution d_i^* we employ d_i, the ground truth distribution associated to the node at hand.

Comparing d_i and d_i^* at every step we are able to take into account the structural error of our model and employing the ground truth d_i we can keep generating a proper structure allowing us to compare also the deeper sub-trees independently from possibly wrong choices in their parents.
Note that during the process we collect also some distributions not directly associated to any actual node: those limiting variable arity node children generation with the special type *nochild*.

As for our tree characterization of Section 3.1.1 the set of possible node types is finite, therefore we can easily compare type distributions: they are probability vectors. In particular the ground truth distribution d_i is the zero vector with a one corresponding to the desired node type.
So once we choose a reasonable function to compare probability vectors we can obtain a practical loss to drive the optimization process toward a network generating properly structured trees. For instance with $D(y)$ being the set of all the desired distributions d_i associated to the target tree y and d_i^* be the corresponding distribution generated during the decoding process we can employ the Mean Squared Error function to obtain a practical structural loss:

$$\mathcal{L}_s(y, y^*) = \mathop{\mathbb{E}}_{d_i \in D(y)} \left[(d_i^* - d_i)^2 \right]$$

Value Loss Since in a supervised setting we can cheat during the generation and obtain a tree with the desired structure we have a one to one mapping between nodes and we can compare every node $n_i \in y$ with its corresponding $n_i^* \in y^*$. In particular we can also compare their associated values; the trick ensure us that nodes match structurally but also in their node type and value type. For each node $n_{t,i}$ having the ValueType vt_t and the concrete value v_i we can employ some proper function $\mathcal{L}_t^v$ to compare values in their concrete domain. As in the top level loss function when adding up different loss we may do it by means of a convex combination to properly balance them and avoiding to weight more the loss on some type of value only due to a different scaling.

$$\mathcal{L}_v(y, y^*) = \mathop{\mathbb{E}}_{n_{t,i} \in y} \left[\beta_t \mathcal{L}_t^v(v_i^*, v_i) \right]$$

4 Experiments

The characterization of our model as given in the previous section is still a bit abstract. In fact we provided a coarse grained architecture, detailed only as much as we needed to ensure that sub-networks can be properly assembled accordingly to every possible tree we might have to generate or to embed.
We treated all the sub-networks as black box building blocks without specifying any details of their internals. The only important aspect in order to be able to ensure their compositionality is their interfaces, the shape of the vectors they expect as input and they generate as output.
The sub-networks internal specifics have been left as implementation choices. Although they can undoubtedly affect the model performance they also might depends on the specific task at hand.

In order to assess our abstract model we addressed all the implementation details and faced the challenges involved in obtaining an efficient implementation, practically employable to experiment with tree-structured data.
We realized it using the popular deep learning framework TensorFlow. In order to exploit all the primitives it provides and to easily integrate our work in a bigger ecosystem. Indeed we made it publicly available open-sourcing it[3].

Exploiting such implementation we ran some exploratory experiments with a twofold objective. On one hand to assess among the possible architectural variants and hyper-parameters configurations which are the most promising ones in terms of performances, in particular evaluating the model capabilities of properly reconstructing trees. On the other to assess our implementation efficiency and scalability, both on CPUs and GPUs architectures, in order to see how practical is to employ it to tackle real world tasks having millions of samples.
We ran our tests on a synthetic dataset of arithmetic expression trees and on a popular Neural Machine Translation task, WMT'14 english-german as provided by [71].

4.1 Implementation

We chose to develop our implementation employing some pre-existing deep learning framework. Starting from scratch it would have been totally infeasible, there are too many different aspects involved.
Among the available frameworks we chose one of the most popular: TensorFlow [10]. It has the wider community, it is well documented and it provides a lot of fundamental building blocks for deep neural networks systems. Additionally it provides also several complementary tools to ease the experimentation process: such as training monitoring, model saving and resuming, and easy to use parallelization mechanisms to scale to distributed and heterogeneous architectures.
In some scenarios TensorFlow had been proved to be less efficient than other frameworks [72], however we opted anyway for such a framework since it is the only one providing a full fine grained control in all the model building process and in the training procedure. Moreover it is the one providing the most ready

[3]`https://github.com/m-colombo/conditional-variational-tree-autoencoder`

to use primitives allowing us to experiment our abstract model in several different variants without having to implement them from scratch.

In order to understand the implementation challenges involved by dynamical tree-structured computations we first need to get a general idea of the TensorFlow execution model [10].
All the computation structure and state it is represented in a single data flow graph. It is first created and then executed possibly multiple times for different inputs. Knowing the whole computation structure since the beginning makes several effective optimizations possible.
For instance when evaluating a certain node of the graph we can compute only the required sub-graph needed to get the desired value and we can reuse shared sub-graph computing them only once. Knowing the size of all the involved computations we can efficiently preallocate memory and possibly effectively allocate the computation on a heterogeneous architecture.
In fact the computation graph is usually built once at the beginning and used several times without changing it.

In our case however the computation graph is not static nor it is fully known a priori. For how we defined our model in Section 3 every sample might have a different structure inducing a different computation graph. Moreover when generating a tree the computation graph it is not even known a priori: it is built incrementally alternating the evaluation of the current node associated sub-graph and, accordingly to the results, the instantiation of the children associated sub-graph.
The inherent dynamical structure of the computation makes therefore impractical to classically employ TensorFlow, the main assumptions on which its execution and optimization model rely are broken. Luckily, we have been able to exploit the fine grained control offered by TensorFlow to made up for such limitations.

Initially we started experimenting with TensorFlow Fold [73], an extension purposely built to efficiently deal with dynamically structured-data. The underlying execution model had been extended to be able to build and optimize dynamic computation graphs.
However even if we succeed in implementing a tree embedder module we faced several difficulties when tackling the generator module implementation. Considering this, along with the fact that its development looks to be abandoned in an early alpha stage, we had to look for a different back-end to support the development of our model. Also because it works only with an old version of the framework (1.0) and we found anyway some difficulties in integrating it with the usual TensorFlow components.

Fortunately, at the time TensorFlow officially introduced a viable alternative: Eager Execution mode. When enabled it allows to skip the graph construction and directly evaluate expressions, i.e. eagerly.
In the classical Graph Execution mode the computation graph is built by means of a Python front-end API which communicates with a C++ back-end that actually handles the computation graph. When a node has to be evaluated the Python API can be used to trigger the evaluation in the C++ back-end and to

gather the results. In the eager execution mode all the Python API that were used to built the graph now directly evaluate to a value rather than to a pointer to some node in the computation graph.
This takes the advantage of a better control on the computation, it is directly related to the Python control flow and therefore it fits well with our situation requiring to arbitrary assemble computations on the fly.
However it has some drawbacks as well. TensorFlow back-end is no more aware of our intentions: we do not declare the whole computation plan, we rather ask time by time to compute part of it. This considerably limits the amount of optimizations that can be done, as they can only be local to single operations and they cannot consider the computation as a whole.
Moreover, despite the flexibility provided by employing the Python API to directly perform computations, it also might slows down the process even more. It introduces in fact a barrier between every operation, parallellisable operations evaluation might be linearised just because of the Python control flow evaluating them one by one.
Naively parallelize the Python API calls might not lead to the expected speed boost, understanding the interaction between the TensorFlow back-end and front-end is not always easy.

Despite the unpromising scenario depicted by all of such limitations and caveats, with some care, it is actually possible to achieve good performance also in Eager Mode. From our experimentations what emerged as crucial for efficiency is to minimize the number of control flow switches between front-end and back-end, in particular trying to execute as much as possible in a single call to the TensorFlow back-end. With this in mind we designed a mini-batch algorithm to process multiple trees in parallel.

4.1.1 Mini-Batch Tree Processing

Often when training a machine learning system a mini-batch procedure is employed: rather than proceeding one sample at a time we employ a batch of samples all at once. This on one side has nice properties regarding the learning process [74], on the other it makes possible to parallelize and considerably speed up the computation.

For instance let us consider the Multilayer Perceptron as introduced in Section 2.2 and in particular focusing on a single layer. It computes $y = \phi(\omega x)$ for some activation function ϕ, weight matrix $\omega \in \mathbb{R}^{n \times m}$ and vector-shaped input sample $x \in \mathbb{R}^m$.
Consider a mini-batch of b samples $x_1, \dots, x_b$. Applying our layer individually to every single sample is hugely slower than processing them altogether as

$$\begin{bmatrix} y_1 \\ \vdots \\ y_b \end{bmatrix} = \phi(\omega \begin{bmatrix} x_1 & \dots & x_b \end{bmatrix}). \tag{42}$$

A matrix-matrix multiplication is much more faster than b matrix-vector multiplications, for reasons both algorithmic and related to hardware optimization exploitation.

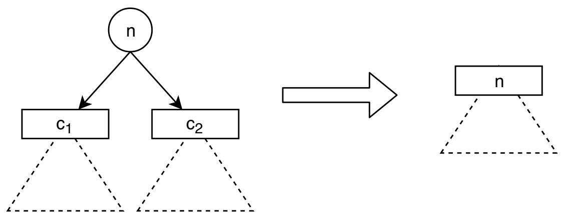

Figure 23: One step of reduction performed during the embedding of a tree. We start from a node n and the embeddings of its children c_1, c_2, each one representing all the sub-tree rooted in the corresponding child. Then we compute an embedding representing all the sub-tree rooted in n by means of the sub-network associated to its node type.

When dealing with dynamically structured data this kind of parallelization is less easy, every sample has a different structure and leads to a different computation. We are no more in a setting where for every sample we have to perform the same exact computation.

Let us consider first the simpler setting of sequences batching. A common approach is to uniform the structure of all the sequences in the mini-batch padding the shorter ones. This way the same computation can be executed, in parallel, for all of them and with some care the desired results for every sequence can be gathered at the right depth in the unfolded recurrent network. Although we are in fact performing more computations it is considerably faster since it can be heavily parallelized.

This approach however is not feasible for more general structures such as trees. Their structural variability is to high to be able to effectively uniform their structure. It is not feasible to reshape all of the trees in a batch in such a way that each one requires the same exact computation graph to be evaluated and still the desired result can be gathered.
Our approach, inspired by the TensorFlow Fold one [73], rather than trying to fully parallelize the computation at the samples level, breaks their computations in smaller steps and tries to aggregate and parallelize them.

Our model characterizes all the possible trees by means of a set of node types, as defined in Section 3.1.1. Then for each one of them some sub-networks are instantiated accordingly to the definition of Section 3.1.2 and Section 3.1.3. Therefore two computations for nodes of the same type have associated the same sub-network and can be aggregated together and computed in parallel, independently whether they belong to the same tree or not.
Based on the sub-network kind it might not be straightforward to aggregate the computations, for instance think about the node inflaters of the generator module in Section 3.1.3. They are composed of some interconnected sub-networks, thus some care is required when propagating the mini-batch computation.
Nonetheless we reduced the problem of batching tree-structured computations to batching simpler computations: all our sub-networks operates on fixed size vectors, usually from the embedding space $\mathcal{E}$, or in the recurrent strategies on se-

quences. For both this scenarios batched computation are commonly employed and indeed implemented by TensorFlow primitives.

Algorithm 1 Pseudo-code describing our algorithm for embedding or generating a batch of trees in parallel

```
for every reduction type $t$ do
    $R_t = \emptyset$
end for
// initialize the sets of computable reductions
if embedding then
    for $t \in$ batch, $l \in$ leaves($t$) do
        $R_{type(l)} \leftarrow R_{type(l)} \cup l$
    end for
else // generating
    for $t \in$ batch do
        $R_{type(root(t))} \leftarrow R_{type(root(t))} \cup root(t)$
    end for
end if

while $\exists t.\ R_t \neq \emptyset$ do
    $\bar{t} = argmax_t\{|R_t|\}$ // the most requested reduction kind
    $\bar{R} = R_{\bar{t}}$ // all the reductions to perform
    $R_{\bar{t}} = \emptyset$
    execute(aggregate($\bar{R}$)) // perform all the reductions in parallel
    for $r \in \bar{R}$ do // update the sets of computable reductions
        for $d \in$ dependent($r$) do // check all the reductions that depends on $r$
            if $d$ is computable then
                $R_{type(d)} = R_{type(d)} \cup d$
            end if
        end for
    end for
end while
```

Let us consider our mini-batch algorithm employed to embed a batch of trees; the generation case is a bit more involved but analogue, it starts from the roots rather then from the leaves and has also to choose the sample structure.
To embed a tree we proceed bottom-up: we start from the leaves and reduce incrementally bigger sub-trees into a single embedding. Given a node n we employ the sub-network associated to its type to merge all its children embeddings into a new embedding representing the whole sub-tree rooted in n. Such a reduction step, as depicted in Figure 23, is iteratively performed from the leaves up to when we obtain a single embedding representing the whole tree.

Our batch computation is therefore realized as an iterative process aggregating similar reduction steps and performing them in parallel, a pseudo-code characterization is provided in Algorithm 1.
At every iteration we have a set of computable reduction steps grouped by their associated node type. Each one has all of its dependencies cleared, i.e. all the input it requires have been already computed.

In general a reduction step will consist in executing a sub-network associated to some node type, although it might also be just a part of it. For instance with recurrent strategies we might choose to consider as a reduction step a single application of the recurrent sub-network, rather than all of its unfolding.
Among all the kinds of reduction step that we can perform we pick the one having the biggest amount of computable reductions, aggregate and execute all of them in a single batch in parallel. Then we update the set of computable reduction steps accordingly to the dependency cleared by the ones just performed.
This way we try to maximize the amount of computations required by every single call to the TensorFlow back-end. Compared to many unforeseen subsequent calls, a bigger unique computation might let the back-end employ better optimization and parallelization strategies.
Although the policy with which we choose the next reduction step is greedy and possibly not optimal, it led us to good results, as shown in Section 4.2.3. A possible future work can be to investigate less greedy policy, possibly finding the best execution schedule considering all the structural information we have available; rather then only the frontier of our computation.

Let set ourself in a simplified setting to clearly depict our mini-batch algorithm. We want to embed in parallel the two trees of Figure 24. Node types are represented by letters, the first number is the first iteration at which the associated reduction can be performed, while the second number is the iteration at which our algorithm actually execute it.
At the first iteration the computable reductions are only those mapping leaves into some embedding. Among the A and L types we perform the L reductions since are the most requested.
This clears all the dependencies of the internal A and B nodes, which can now be aggregated with the corresponding A and B leaves in the other tree. At this point we can perform the reductions associated to A and B in any order and then finally embed the two roots in parallel.

Notice how this procedure allow us to aggregate also A and B nodes, which might have been computed in different iterations with a different execution policy. It is just a lucky case and several counterexamples in which our policy is suboptimal can be provided, for an example see Figure 25. However our intuition hints at an average good performance. We always greedily execute the reduction step involving more nodes than all the others. This implies that we always execute the most parallelizable operation and it likely be the one clearing more dependencies than all the others. Therefore it likely leads to much more new computable reduction steps and thus it likely increases the parallelizability of all the other operations, more than any other reduction steps.

4.1.2 Sub-Networks

For how we implemented the mini-batch algorithm the parallelization degree is inversely proportional to the number of different node types. Clearly this depends on the specific task we are tackling, however for the same task there are often many possible ways of modelling its tree domain.
In general all the design choices have to be made carefully balancing the accuracy in the modelling and the expected computational performance. Although

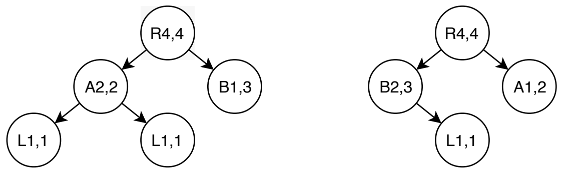

Figure 24: A simplified example of our mini-batch tree processing algorithm applied to embed in parallel two trees differing in their structure. The letter in the nodes represents their type, in our model two nodes with the same type have to be processed by the same sub-network and possibly this computations can be aggregated and performed at once in parallel. The numbers represent respectively the first iteration at which the node can be computed and the actual iteration at which it is computed. Starting from the leaves we aggregate all the computable nodes and compute the operation with most nodes.

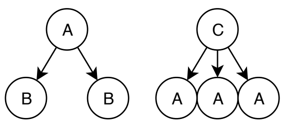

Figure 25: A counter example where our batch algorithm greedy policy is not optimal. Our algorithm would execute in the order $3A, 2B, 1A, 1C$ while the optimal execution policy would choose $2B, 4A, 1C$.

our implementation made computationally feasible to deal with trees, it still really expensive. Some good choices in the design and some compromises can spare us hours or days of computation.
In all the design and implementation process we always tried to stay in the sweet spot with the right balance between efficiency and efficacy.

Another aspect directly related to the sub-networks number and complexity is the amount of hyper-parameters. Every kind of sub-network has possibly many of them, depending on its internal architecture which could be an hyper-parameter by itself.
In general when performing model selection different hyper-parameter spaces are combined by means of a cartesian product. It means that every kind of sub-network expand the hyper-parameters search space by a multiplicative factor proportional to its search space dimension. Few sub-network kinds might be enough to make the search space dimension explode.

For instance consider one of the simplest sub-networks involved in the tree generation process, the fixed arity inflater of Figure 21 and re-proposed here in Figure 26. Every rounded corner shape it is a sub-network by itself with the only imposed constraints regarding the input and output vectors size. Many possible internal implementations are feasible. Even if during model selection for every sub-network only few variants have to be assessed, when combining all of thepossible variants of all the possible sub-networks the problem easily become intractable.

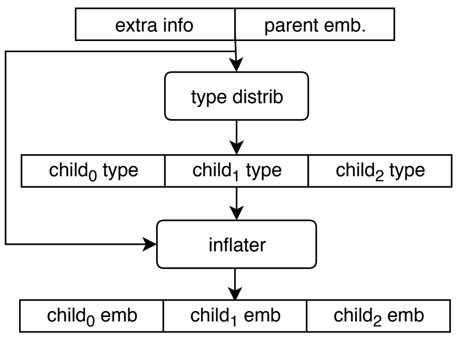

Figure 26: One of the simplest sub-networks involved in the tree generator module. It has two different components each one only constrained in its input and output vectors size. Therefore many different internal implementations are possible.

Dimensioning In order to practically deal with the huge hyper-parameters search space we did some initial simplifications. Leaving as a future work the problem of how to properly perform a model selection and possibly understand how different sub-networks hyper-parameters choices interact.
We imposed the same architecture to all the sub-networks of the same category, e.g. all the node embedders independently from their associated type. In particular we considered all the core components to be Multilayer Perceptron with the same activation function.
As an additional simplification we do not want to select hyper-parameters specifically for every individual component, as this might be computationally impractical. We want instead to be able to tune them at a more global scale, for the whole tree module.
Therefore we dimensioned all the MLP layers in function of only two hyper-parameters: the embedding size and the so called hidden layers coefficient $h \in (0, \infty)$. The former directly governs the size of the inputs and the outputs vectors. The latter abstractly expresses how big should the hidden layers be. For every category of sub-network we dimension its layers proportionally to this coefficient h and its input and output expected size.
For instance consider an MLP with one hidden layer, input size s_i and output size s_o. Then, for example, we might compute its hidden layer size s_h as

$$s_h = \lceil (s_i + s_o) * h \rceil, \tag{43}$$

the actual formula employed might change accordingly to the category of sub-network. However the only hyper-parameters employed are the hidden coefficient h, and the input and output size, which strictly depend on the embedding size.
This way we are clearly oversimplifying relating all the sub-networks together, however it makes it possible to practically test some different variants at model selection time. This is one of the choices we made as a trade-off between accuracy and efficiency.

Flat Strategies Sub-Networks Another choice motivated by efficiency led us to the flat strategies employed for variable arity node types. On one side in

this way we can process all the children in parallel, without losing any information up to *cutarity* children and loosely approximating the additional ones.
The rationale is that in many real life scenarios most nodes have few children and few nodes have many children. Thus we trade off the accuracy on few nodes for the efficiency on many others. Following the same consideration we can see that many times we might deal with less than *cutarity* children but still we need to somehow evaluate sub-networks on all of them due to their input and output size requirements. This basically makes us perform useless computations.
With some care we implemented custom TensorFlow neural network components minimizing the computations to the required ones.

Zeroable Dense Network When embedding a variable arity node we summarize children exceeding the *cutarity* in a single embedding and then process all of them with a single sub-network. In our specific implementation a two layer MLP.
Be $k =$*cutarity* and be c_k the embedding summarizing exceeding children, then the parent embedding p is obtained as:

$$x = c_0 \cdot ... \cdot c_k \tag{44}$$
$$p = \phi_o(\omega_o(\phi_h(\omega_h x))) \tag{45}$$

however, since we encode missing children with the all-zero vector, in many case the last $k - z$ children are zero. Hence we can avoid some useless computations:

$$\omega_h x = \left[\omega_h^{(0)} \cdots \omega_h^{(k)}\right] \left[c_0 \cdots c_k\right] \tag{46}$$
$$c_z = c_{z+1} = \cdots = c_k = 0 \tag{47}$$
$$p = \phi_o(\omega_o(\phi_h(\left[\omega_h^{(0)} \cdots \omega_h^{(z-1)}\right] \left[c_0 \cdots c_{z-1}\right]))) \tag{48}$$

Where $\omega_h^{(i)}$ denotes the portion of ω_h that during the matrix-vector multiplication $\omega_h x$ is used along with the values of c_i.
To perform a batch computation we have to consider z for the node with the maximum amount of children. In such a way we can perform a parallel computation for all the nodes in the batch. As often happens it is more efficient to compute the same thing many times in parallel than computing slightly different things one by one.

Parallel Dense Network When generating the children embeddings of a variable arity node we directly generates the first $k =$*cutarity*-1 of them. Evaluating each time a sub-network with the exact same architecture but with different weight parameters. To efficiently implement this we exploited TensorFlow primitives computing stacked matrix multiplications all at once. It efficiently computes a list of matrix multiplications $A_1B_1, \ldots, A_nB_n$ as long as all the multiplication are equal in size, formally $A_i \in \mathcal{R}^{m_1 \times m_2}, B_i \in \mathcal{R}^{m_2 \times m_3} \quad \forall i \in [1, n]$.
As for the embedding case we implemented a custom neural network components realizing only the necessary computations. It gathers only the needed matrices and stacks them together in order to compute only the desired children embeddings with one TensorFlow back-end call. We omit the formal characterization since it is similar to the one before, it is just a bit more tedious.

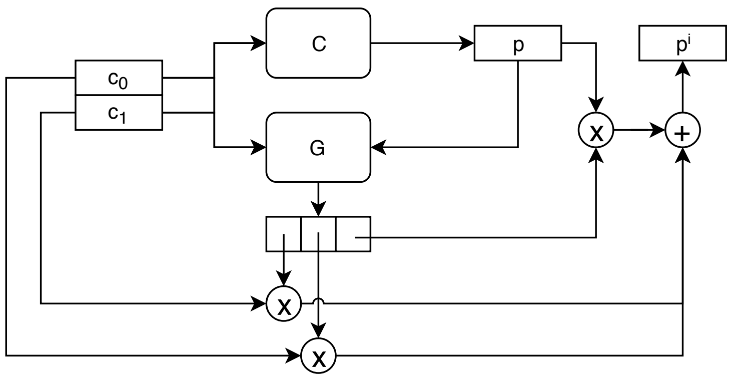

Figure 27: Schematics of the output gate we employed on a fixed arity node embedder. Given the children embedding c_0 and c_1 the network $\mathcal{C}$ merge them in their parent embedding p. The gate network $\mathcal{G}$ computes a relevance coefficient for each children and for the parent embedding, although not illustrated for readability they are softmax normalized to sum up to one. Then the final embedding p' is obtained as a convex combination of all the children and the parent embedding, by means of the X and $+$ nodes, respectively the multiplication and addition operation.

Gates One choice we made trading efficiency for efficacy is concerning gating techniques. Inspired by their effectiveness in deep networks, see for instance the Highway Networks [75], we implemented a multiplicative output gate in the node embedders and inflaters. Ideally it makes the information directly flows from and to the right sub-trees.
Figure 27 depicts the schematics of the employed gating technique for a fixed arity encoder with two children. More generally let us consider the case for k children: we merge their embeddings c_i into the parent embedding p by means of the sub-network $\mathcal{C}$

$$p = \mathcal{C}(c_1 \cdot ... \cdot c_k) \tag{49}$$

then the gate network $\mathcal{G}$ evaluates for each children and for the parent a coefficient representing how much each one should flow through the network

$$g = \mathcal{G}(p \cdot c_1 \cdot ... \cdot c_k) \tag{50}$$

this coefficients are then normalized by means of a softmax function and used to compute the actual embedding as a convex combination of the ones of the computed parent and all of its children

$$\hat{g}_i = \frac{e^{g_i}}{\sum_{j=0}^{k} e^{g_j}} \tag{51}$$

$$p' = p\hat{g}_0 + \sum_{i=1}^{k} \hat{g}_i c_i. \tag{52}$$

Analogous gating mechanisms have been implemented also for the other encoder and decoder sub-networks.

	x			y		
	min	$\mu(\sigma)$	max	min	$\mu(\sigma)$	max
Depth	2	3.8(0.6)	4	1	3.6(0.9)	4
Nodes	3	14.7(7.5)	50	1	12.3(7.5)	48
Arity	2	2.4(0.3)	4	2	2.4(0.4)	4
Leaves	2	9.2(5.0)	33	1	7.8(4.8)	32

Table 2: Main characteristics of the generated Arithmetic Expression Tree pairs $< x, y >$ we employed to perform our exploratory experiments. We considered the tree depth, the number of nodes, their arity and the number of leaves.

4.2 Benchmarks

4.2.1 Arithmetic Expression Tree

During the development process we employed a simple synthetic dataset to assess our implementation choices and also to investigate how different approaches and hyper-parameters choices might interact and affect our model capabilities. Specifically we used the Arithmetic Expression Tree as the one characterized in Section 3.1.1 to model a tree transduction task performing the left-most possible derivation step in the expression tree evaluation.
For instance, given the expression $(5 - 3) + (6 + 6)$ in the form of its parse tree, the tree that we want to obtain is the one corresponding to the expression $2 + (6 + 6)$.
We tested some architectural variants and hyper-parameters values in a fixed and simple settings. Note however that this does not want to be an exhaustive analysis, it is rather an exploratory analysis aimed at getting an idea of how our model behaves.

Experimental Settings

Dataset Rather than employing a static dataset we dynamically generate trees and compute their derivations on the fly. We recursively build one of them sampling uniformly the type of each node, the number of children they have in the case of variable arity, and leaves values. When we first sample the root type we do not choose a value node, in order to avoid having one third of the dataset to be single node trees.
Being the dataset dynamic we do not employ a static training and validation set, every mini-batch is composed of different trees. In Table 2 are summarized the main characteristics of the generated trees.

Model In all the tests we employed the same model, apart from the assessed hyper-parameters. Specifically we used an embedding size of 100, flat strategies with *cutarity* $= 4$, output gates in the encoder sub-networks and fixed sub-network dimensioning coefficient $h = 0.3$, Leaky Rectified Linear Unit [76] as activation functions, Kullback-Leibler rescaling coefficient $\delta = 0.1$. In order to stochastically maximize $P(y|z, x)$ we sample two different values from $z \sim Q(Z|x, y)$ and thus reconstruct two output trees for each input.

Learning We tested all the model training them with Adam Optimizer [77] with learning rate $0.001, \beta_1 = 0.9, \beta_2 = 0.999, eps = 10^{-8}$, for 4000 iterations and employing mini-batch of 64 tree pairs.

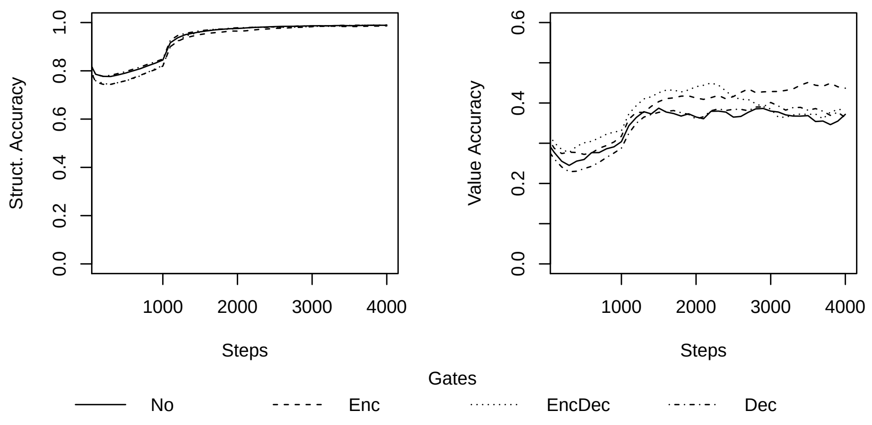

Figure 28: Performance plots showing the results of employing output gates. In the four combinations: no gates, only in the encoder module, in both the encoder and the decoder module, only in the decoder module.

Evaluation Metrics To assess the performance we employed the two metrics defined in Section 3.1.4. The structural accuracy acc_s quantifying how similar is the generated tree structure compared to the ground truth. The value accuracy acc_v quantifying how similar are the generated tree values compared to the ground truth. Both of them range in $[0, 1]$, where 1 means the trees are identical and 0 means they are totally different.
All the performance plots you will see ahead report such metrics averaged on a freshly generated mini-batch and in an unsupervised settings. We sample the encoding from the prior $\mathcal{N}(Z; 0, I)$ rather than from our estimated posterior $Q(Z|x, y)$. For readability all the plots have been smoothed by means of a moving average.

Gates The gates efficacy test results are shown in Figure 28. We tested the four combinations raising from employing or not employing gate mechanisms in the decoder and in the encoder modules. The structural accuracy does not looks to be affected, in few iterations all the models achieve perfect structural matching. While for value accuracy we can observe that none of the models achieved perfect performance, this is quite a usual scenario in our test, values looks to be more difficult to be learnt. Although at the end the learning is not really stable, we can see that the gate mechanisms do improve value accuracy, especially when employed in the encoder.

Variable Arity Strategy We compared the efficacy of the two strategies we developed for variable arity nodes. In Figure 29 we can see the results. Flat strategy shows better performance achieving perfect structural accuracy while recurrent strategy is unable to reach it. Flat strategy shows a slightly unstable behaviour on the value accuracy in last part of the experiment. Compared to it the recurrent strategy is more stable although it achieve a lower performance.

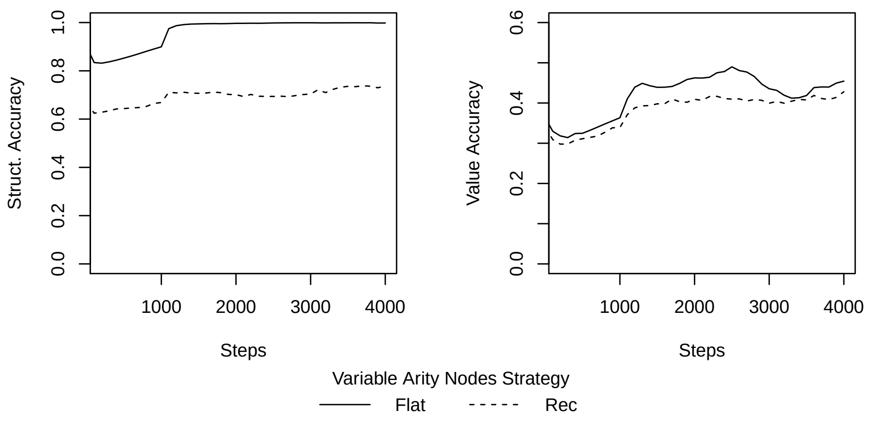

Figure 29: Performance plots showing the results of employing the different strategies we developed for variable arity nodes. The flat one and the recurrent one.

Kullback-Leibler Regularization In Equation 28 we introduced the regularization coefficient δ weighting the contribution of the Kullback-Leibler divergence term in the loss. In order to get an initial idea of how it affects the model capabilities we ran our test with some different values of δ.
From the results of Figure 30 we can observe that structural accuracy is inversely proportional to δ, while for value accuracy some more peculiar behaviours can be appreciated. For instance consider the test with $\delta = 100$, its structural accuracy does not reach the maximum and it is outperformed by all the other models apart from the one having $\delta = 1000$. However its value accuracy almost outperform all the other models.

Bigger Trees In light of this few experiments we wanted to see how our model behaves with deeper trees. We employed flat strategies, regularization coefficient $\delta = 1$ and gates both in the encoder and in the decoder. Then we tested them on our synthetic Arithmetic Expression Tree transduction task employing trees of maximum depth: 5, 7 and 9.
Figure 31 shows the results. We can observe that the deeper is the model the more hard is to learn its structure. While for the value accuracy we unexpectedly observe the inverse. In Figure 32 you can observe two examples of transductions performed by the model trained on trees of maximum depth 5. Such examples show that our model has actually learnt to perform arithmetic computations, however struggles to remember leaf values. Indeed in the second example the generated tree has the right structure and the right computation has been performed, however one of the not involved leaves changed its value.

In all our exploratory experiments we observed a peculiar behaviour on the value accuracy, an initial fast convergence and then a degradation. Although we leave as future work the better investigation of this phenomena, our initial guess is that this is the effect of the interaction of the different components in

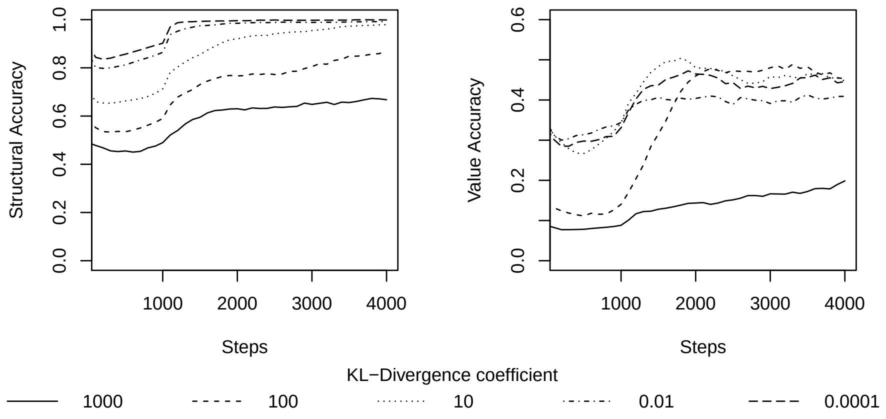

Figure 30: Performance plots showing the results of employing different regularization coefficient for the Kullback-Leibler divergence. The δ coefficient as introduced in Equation 28.

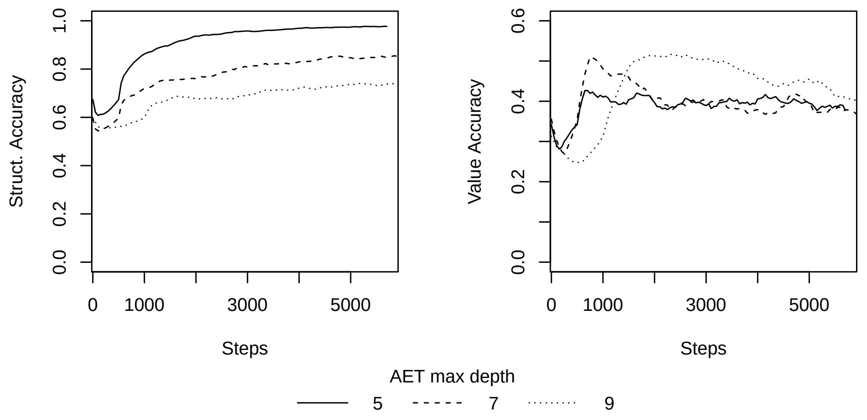

Figure 31: Performance plots showing the results when tackling trees of different maximum depths. The same model have been tested on a dataset having trees of maximum depth: 5, 7 and 9.

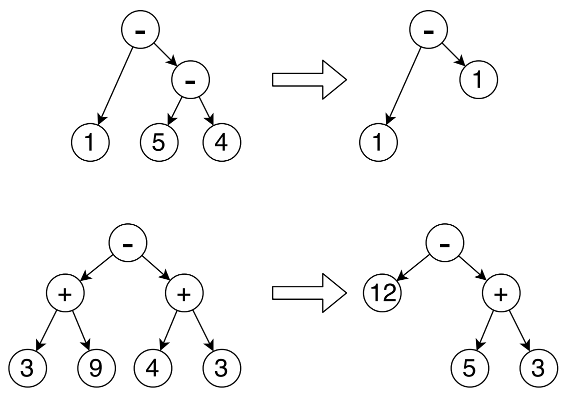

Figure 32: Example of two transductions performed by our model trained on trees having maximum depth 5. Notice that the second transduction is wrong.

our loss function.
In particular of the regularizing divergence component gradually pushing the code distribution $Q(Z|x, y)$ to be closer to $\mathcal{N}(0, I)$ and less informative to the reconstruction process. For deeper tree or with less powerful model it is more hard to learn how to map the posterior Q to be close to $\mathcal{N}$, therefore we can observe an initial better performance. As for instance showed in Figure 31. A better regularization coefficient or possibly some dynamic schedule of its value might mitigate this phenomena.

4.2.2 Tree-to-Tree Neural Machine Translation

Neural Machine Translation is the perfect use case for which we think this kind of models and which drove the realization of our. However tackling such tasks effectively it is challenging. The text preprocessing phase should be designed with care, how we normalize and represent words heavily affects the final performance. Models obtaining competitive results are usually quite big and involved, therefore their training procedure requires a considerably amount of time and computational resources. Moreover they usually employ sophisticated learning boosting mechanisms, such for instance attention based ones. So complex models usually have a wide variety of hyper-parameters and thus their model selection phase might be quite expensive.
For these reasons we did not seriously tackle a translation task yet. Rather than trying to obtain competitive translation performance we just ran some exploratory experiments and set the basis for doing it as a future work.

Experimental Settings

Dataset We employ the popular WMT english-german dataset containing 4.5 millions of sentence pairs [71]. We parsed each sentence in its constituency parse tree by mean of the Stanford Parser [78]. Then we replaced each leaf word with its corresponding word embedding obtained by means of a pre-trained fastText model [79]. In Figure 33 can be appreciated an example of the trees associated to a sentences pair of the dataset, while in Table 3 are shown the obtained trees main characteristics.

Train	English			German		
	min	$\mu(\sigma)$	max	min	$\mu(\sigma)$	max
Depth	4	14 (4.7)	67	5	9.3 (2.1)	36
Nodes	3	74 (41)	576	4	60 (33)	566
Arity	1	1.5 (0.9)	92	1	1.7 (1.3)	85
Leaves	1	25 (14)	92	1	24 (14)	222

Valid	English			German		
	min	$\mu(\sigma)$	max	min	$\mu(\sigma)$	max
Depth	4	13 (4.5)	52	5	9.0 (2.1)	24
Nodes	3	66 (38)	376	4	56 (33)	366
Arity	1	1.5 (0.9)	23	1	1.6 (1.2)	23
Leaves	1	23 (13)	137	1	23 (14)	145

Table 3: Main characteristics of the constituency trees generated from the WMT'14 English-German dataset. We considered the tree depth, the number of nodes, their arity and the number of leaves. Training is composed of 4.5 million sentence pairs, while validation is composed of 6000 pairs, obtained as the concatenation of the newstest 2012 and newstest 2013.

Tree Grammar We characterize the tree domain in our formalisms by means of few node types, rather then employing a node type for each possible syntactic category. There are 147 different categories [48]. Modelling each one with a dedicated node type is hard, moreover having so much node types makes us having as much different sub-networks. Considerably lowering the parallelizability of our batching algorithm and also increasing a lot the number of parameters to optimize.
So we modelled all constituency trees only with three node types: a generic node having as associated value the syntactic category, a pre-leaf node having as associated value the syntactic category of its associated leaf and a leaf node having as associated value the embedding corresponding to its word.

Model In light of the AET experiments and of some manual hyper-parameters assessments we selected three different model configurations to test, only differing in the embedding size being 100, 200 or 300. We employed flat strategies with *cutarity*= 20, output gates both in the encoder and decoder modules and regularization coefficient $\delta = 0.1$.. All the sub-networks have been instantiated with Leaky Rectified Linear Unit activation function [76] and dimensioned proportionally to the same hidden coefficient $h = 0.3$. In order to stochastically maximize $P(y|z, x)$ we sample two different values from $z \sim Q(Z|x, y)$ and thus reconstruct two output trees for each input.
To get an idea of the size of the models consider that the number of parameters to optimize are, in order from the smallest the the biggest: 4.3M, 13M and 27M.

Learning In the AET experiments we observed how our model struggles to capture values deeper in the trees, therefore we did not aim at learning the word embedding in the leaves of our sentence trees. We rather want to learn the constituency structure and its associated syntactic categories. If we are able to succeed then we can employ some attention based mechanism with

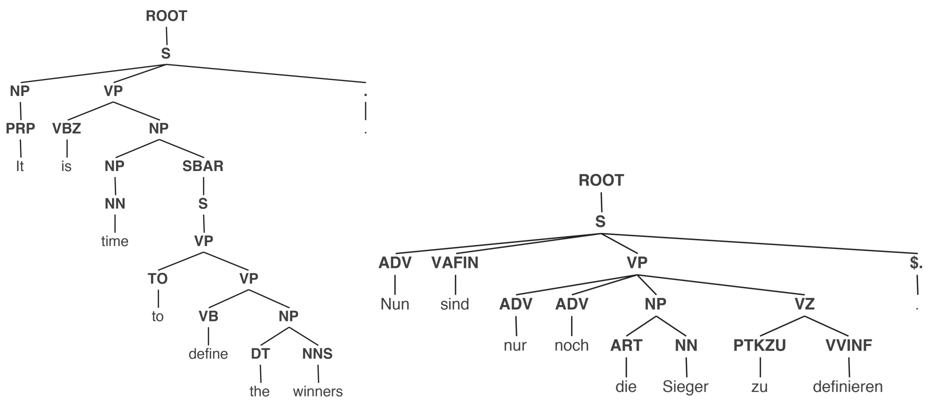

Figure 33: Example of the constituency trees of a pair of sentences in the dataset.

the generated leaves embeddings along with the input sentence to perform a translation exploiting the learnt structure. Intuitively, knowing the structure of the target sentence should boost the translation performance.
We trained the selected models as long as we had an average performance improvements, specifically let i be the current iteration, l_{old} be the average loss computed on the iterations $[i - 10^4, i - 5 * 10^3]$ and l_{new} the average loss computed on the iterations $[i - 5 * 10^3, i]$. We kept the training process going as long as $\frac{l_{new}}{l_{old}} < 0.9999$.
We employed an Adam Optimizer [77] with learning rate $0.001, \beta_1 = 0.9, \beta_2 = 0.999, eps = 10^{-8}$. In order to have a stabler learning process, at every iteration we clipped the gradient by its global norm [80] to be lower than 0.1. We used mini-batch of 256 tree pairs, avoiding the biggest trees, specifically those whose depth exceed 25 or having more than 120 nodes.
To train each of such models we spent 2-3 days on an Nvidia Tesla V100-SXM2-16GB. Approximatively processing 10M sentences.

Evaluation Metrics As for AETs to assess the performance we employed the two metrics defined in Section 3.1.4. The structural accuracy acc_s quantifying how similar is the generated tree structure compared to the ground truth. The value accuracy acc_v quantifying how similar are the generated tree values compared to the ground truth. Both of them range in $[0, 1]$, where 1 means the trees are identical and 0 means they are totally different. Performance are assessed on the validation set in un unsupervised setting. We sample the encoding from the prior $\mathcal{N}(Z; 0, I)$ rather than from our estimated posterior $Q(Z|x, y)$.

In Figure 34 are showed the performance of our model tested in the tree different configurations just described. As for the AET tests we can observe an initial fast convergence in the structural and value accuracy, and then the latter continues to slowly increase. Interestingly the bigger models behaved worst, possibly a better hyper-parameterization is needed.
Although the average performance is low, we observed that the best model

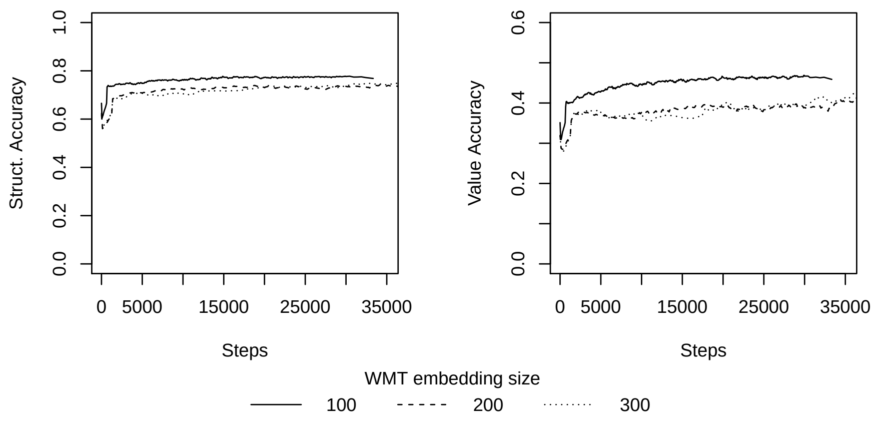

Figure 34: Performance plot showing the results of models employing different embedding size when tackling the WMT english-german translation task. Note that the value accuracy is computed on the syntactic category labels rather than on the words.

achieved both on structural and values accuracy a perfect match on 3% of the sentences. This makes us suppose that our model is indeed capable of dealing with tree-to-tree translation. We just need a better hyper-parameters tuning and possibly more powerful learning mechanisms.

4.2.3 Computational Efficiency

As we just seen in the previous section such tree models can be huge and can take days to be trained. This makes crucial to have an efficient implementation. During all the development process we ran several benchmark to assess the computational efficiency of our model. This had been a really delicate facet of our work, a wrong choice made in the abstract model specification or in the implementation can make our model impractical to use. For this reason we tried to find the right balance between model efficacy and computational efficiency.

Recurrent Strategy A first assessment have been made to evaluate the recurrent strategies practicability for variable arity nodes. We employed the synthetic AET transduction task of Section 4.2.1 and with all the same hyper-parameters we compared the running time of our model for flat and recurrent variable arity node strategies on some different kind of trees. As showed in Table 4 recurrent strategies are in general slower and scale worse to deeper and wider trees. Indeed we did not employ them in our experiments, always preferring flat strategies.

Hardware Architectures We also ran some benchmarks on different hardware architectures and with different execution environment settings. On one side to compare our implementation efficiency on different architectures, on the other to possibly find the most effective tuning to use on each one of them.

Max depth	Max arity	Depth $\mu(\sigma)$	Nodes $\mu(\sigma)$	Arity $\mu(\sigma)$	Flat node/s	Rec node/s	Speed up
4	5	3.7 (0.7)	16 (10)	2.7 (1.0)	3198	1623	2.0
	10	3.8 (0.6)	44 (39)	4.0 (2.7)	4833	1691	2.8
	15	3.9 (0.5)	93 (82)	5.3 (4.3)	6303	1782	3.5
6	5	5.6 (1.2)	69 (47)	2.8 (1.1)	5366	2163	2.5
	10	5.6 (1.2)	299 (303)	4.0 (2.7)	6481	2139	3.0
	15	5.5 (1.3)	1083 (1188)	5.2 (4.3)	7819	2130	3.7

Table 4: Efficiency test results comparing efficiency of recurrent strategies and flat strategies. Recurrent strategies does not scale to deeper and wider trees.

Dataset	Batch Size	Emb. Size	Depths	Nodes	Arities
AET1	128	100	3.7 (0.7)	13.3 (7.4)	2.5 (0.7)
AET2			6.4 (1.6)	77.2 (55.3)	2.5 (0.7)
AET3			8.1 (2.3)	222.4 (172.6)	2.5 (0.7)
WMT1	128	100	10.7 (3.6)	54.8 (24.4)	1.5 (1.1)
WMT3		200			
WMT2	256	100			
WMT4		200			

Table 5: Datasets employed for benchmarking performances on different architectures.

To perform this comparison we used seven different benchmarks based on the two experiments of the previous sections, Arithmetic Expression Tree and Machine Translation. Table 5 characterizes the datasets we employed as test beds.

We compared five different settings executing on CPUs, each time compiling TensorFlow from sources in order to exploit all the available hardware dependent optimizations.
We tested an Intel Core i7-4870HQ of a mid-2014 MacBook Pro and an Intel Xeon Gold 6132 in four different configurations raising from the combinations of two options: whether to employ the optimizations provided by Intel Math Kernel Library (MKL) [81]–[83] and whether to try to parallelize limiting the number of available cores to $\frac{1}{4}$ by mean of Docker tools. In Table 6 are reported the results.

In general we can observe that on the Xeon Gold employing all the cpus with MKL optimizations performs better than all the other configurations. Although it is interesting noticing how the MKL optimized setting performs when restricted to a subset of the cpus. It shows worse speed-up when compared to the two configurations employing all the cpus.

We also tested performance on two different GPU architectures: a Nvidia Pascal P100-PCIE-16GB and a Nvidia Tesla V100-SXM2-16GB. They have CUDA

Dataset	i7	$\frac{1}{4}$G - MKL	$\frac{1}{4}$G	G - MKL	G
	node/s - tree/s				
AET1	5030 - 186	3755 - 138	4759 - 178	**5035 -187**	4573 - 167
AET2	6088 - 39	5691 - 38	6153 - 40	**7678 - 50**	5765 - 38
AET3	5487 - 12	6253 - 15	6986 - 16	**8819 - 21**	6600 - 16
WMT1	1952 - 18	2561 - 23	2705 - 25	**3670 - 33**	2520 - 23
WMT2	1844 - 17	2523 - 23	3245 - 29	**4365 - 39**	3135 - 28
WMT3	1378 - 13	2561 - 23	2157 - 20	**3014 - 27**	2030 - 19
WMT4	995 - 9	2523 - 23	2633 - 25	**3740 - 34**	2511 - 23

Table 6: Benchmark results comparing some CPUs configurations. We tested the processor of a MacBook Pro mid-2014, an Intel Core i7-4870HQ . And an Intel Xeon Gold 6132 in four different configurations out of the combinations of two settings: whether we use Intel Math Kernel Library (MKL) [81]–[83] optimizations and whether we employ only $\frac{1}{4}$ of the CPUs, i.e. 14 logic units, limiting their use by mean of Docker tools. Each columns shows the performance achieved on each dataset in the format $n - t$, respectively the nodes and the trees processed per second.

Dataset	node/s - tree/s	
	P100	V100
AET1	2778 - 103	**4292 - 152**
AET2	5177 - 34	**7434 - 49**
AET3	7285 - 17	**9212 - 21**
WMT1	3103 - 26	**4675 - 40**
WMT2	3581 - 31	**5554 - 48**
WMT3	3118 - 26	**4597 - 40**
WMT4	3858 - 33	**5090 - 44**

Table 7: Benchmark results for GPUs. Specifically for a Nvidia Pascal P100-PCIE-16GB and a Nvidia Tesla V100-SXM2-16GB

capabilities of, respectively, 6.0 and 7.0. In Table 7 are reported the results. Clearly the Tesla V100 outperform the less powerful Pascal P100, however the interesting comparison is the one with the CPUs.

On the smaller AET datasets CPUs and GPUs best performances are close, while on the bigger WMT tasks they are still comparable; there is less than a factor two. This clearly depends from our design and implementation choices, nevertheless we could motivate this phenomena as an effect of the computation dynamical nature and its branched structure. In general CPUs deal better with small computations with possibly a complicated structure, while GPUs perform better with big computation with simpler structures.

An additional complication rising from the the computation dynamical nature concerns memory occupation. In general it is not easy to estimate how much memory is required to perform one training step for a certain mini-batch of trees. In particular when generating trees without a ground truth guiding the generation process.

When performing computation on GPUs the shortage of memory is one of the main limitations. In a static setting you simply tune the size of your application to fit into all the available GPU memory. With dynamic computations it is not that easy, every iteration has a different memory requirement.

You could dimension the model accordingly to some empirical approximation of how much memory is required in the worst case. However, depending on how variable is the memory requirement of every iteration, this might lead you to use for most of the time really little of the available resources. Moreover, if only a single mini-batch exceed the memory requirement approximation, you will end up anyway with an Out of Memory error.
In the best case such an error leads to simply fail the current iteration, in the worst case it might ruin the GPU session moving it into some undefined state and making necessary to restart it, possibly losing what was in memory. Therefore it is not easy to allocate dynamical computation, such the one performed by our model, on GPUs.

In our implementation we adjust the mini-batch size accordingly to the amount of Out of Memory errors, but we accept that some errors may occur for a few mini-batches, those having a particular high number of trees which induce a bigger computation.
During the learning process we often save the model status, so when an error occur we skip the current mini-batch and try to recover from the GPU in-memory model. If this does not work we recover it from the most recent checkpoint. When we end in the worst case, where our GPU session is no more usable, we recreate it from scratch before recovering the model from the latest checkpoint. This technique allowed us to exploit at most the GPU memory, making possible to process up to 512 WMT samples per mini-batch.

When performing computation on CPUs the amount of memory it not as crucial as on GPUs. In fact in our CPU experiments we did not had any memory issues, the most complicate aspect have been tuning the MKL optimizations.
As showed in Table 6 such optimizations led to the best CPU performance, in some case also better then those obtained employing GPUs.
However when executing a program with such optimizations there are a lot of possible configurations, several settings have to be tuned to get the best performance. In our experiments we employed the settings suggested by the TensorFlow community and we assessed few similar variants. The performance where significantly different, understanding which setting to employ is not straightforward.

Another interesting aspect concerns parallelizability. As shown by results of Table 6 the MKL performances are much better when using all the available processing units. When employing a subset of $\frac{1}{4}$ of them the performances are worse with MKL than without. This possibly can be fixed with right MKL configuration, however it makes hard to efficiently run several concurrent tasks on the same machine. On GPUs this kind of parallelism is simpler, as long as there is enough memory.

5 Conclusion

In this thesis we tackled the problem of building a neural network based system capable of learning from examples some unconstrained tree transduction function. We set our self in a probabilistic setting, modelling the data as generated by an underlying generative stochastic process governed by some joint distribution $P(Y|X)$; where both X and Y are tree-structured random variables. The probabilistic mapping we aim to learn is $X \rightsquigarrow Y$, the one induced by the joint distribution.

We designed our Conditional Variational Tree-Autoencoder as an original extension of the popular Conditional Variational Autoencoder framework, making it capable of dealing with tree-structure data.
Specifically we designed abstract architectures for the encoder and decoder module as Recursive Neural Networks that can be adapted to every possible domain of trees. We introduced a simple formalism with which we can characterize tree domains. We then presented a methodology to instantiate, accordingly to such tree characterizations, all the needed sub-networks in order to be able to dynamically assemble, for every possible tree, a proper network capable of embedding or generating it.

We faced all the implementation details and realized a practical implementation of our model, in particular implementing an efficient mini-batch algorithm for tree-structured computations. In the spirit of possibly contributing to the community we realized our implementation integrating it in one of the most popular deep learning framework, TensorFlow, and we made it publicly available disclosing it source code[4].

The execution efficiency of such implementation have been tested on some different CPU and GPU architectures. Results suggest that it is practically employable to tackle real life tasks.
We also exploit the implementation to test our abstract model with some initial exploratory assessments of its learning capabilities. Results suggest that our model is indeed able to learn to perform tree transduction, however finding the right hyper-parameters is challenging and possibly some more powerful mechanisms are needed to achieve a competitive performances.

In fact we think that to achieve good learning performances on real life tasks some more work is needed. With this thesis we provided an abstract framework and a practical implementation of all the core components.
Rather than tackling some actual tree transduction task, we focused on obtaining a working implementation both in terms of computational feasibility and learning capabilities.
Indeed we often had to compromise leaving something for future works in order to focus on the core components needed to get a working implementation where to start from. Now from it we can further experiment and start building effective models.

[4]`https://github.com/m-colombo/conditional-variational-tree-autoencoder`

5.1 Future works

Our model still has to be endowed with most of those mechanisms which usually make a huge different in performances.
Attention mechanisms are at the core of many successful models, whether sequential [35] or over trees [43], thus introducing them in our model might boost learning performance.
In particular rather than providing to the decoder network the conditioning x as a flat embedding we might employ a tree structured attention mechanism on the whole tree of intermediate embeddings of x. At every decoding step a proper embedding is summarized from the encoding tree. Clearly this is challenging in terms of computational efficiency.
Without the attention mechanism during training we perform 3 tree shaped computations per sample, two encoding for x and y, and the decoding for y^*. With a the tree-attention mechanism we have to compute a tree shaped computation for approximatively every node of y. In the spirit of trading off efficacy for efficiency would be interesting to investigate the possibility of sharing the same summarized embedding for multiple nodes in a sub-tree, therefore computing it less times.
Another powerful mechanism is gating. In our implementation almost all of the sub-networks are realized as vanilla Multilayer Percepton and then endowed with an output gate. Employing more complex gated sub-network such Long Short Term Memories adapted for tree structures might foster the information flow [46], [49].

As we have seen a crucial aspect for this kind of models is their computational efficiency. They require a lot of memory, computation capabilities and are not trivially parallelizable. It is usual that the training of a single model requires several days, hence a small speed up in some core components might lead to a saving of hours of computation.
Therefore we may better investigate our training procedure from an algorithmic point of view. In particular the mini-batch algorithm, we may find more effective policy to schedule aggregated execution of sub-networks, rather than the current greedy one.
Concretely considering the implementation details we may try to better exploit the TensorFlow primitives, for instance hybridly employing Eager Execution mode for the high level orchestrations and Graph Execution mode to evaluate sub-networks all at once.
Last but not least we might try to optimize the execution accordingly to the underlying hardware. For instance explicitly considering jobs allocation based on the number and on the kinds of available GPUs and CPUs. Possibly considering a heterogeneous allocation based on the kind of computation.

The tree characterization framework as introduced in Section 3.1.1 can be extended in order to model more closely the actual target domain, for instance considering parent-child type constraints. Consequently the implementation should be adapted. Although in general it is not always straight forward to precisely characterize the tree domain we are willing to use, in some scenarios performances might benefit.

Due to the model complexity there are many hyper-parameters which make the model selection process really expensive, up to be infeasible in many scenarios. Thus a useful future work might be an investigation aiming at finding feasible and effective hyper-parameters selection procedures.
First a broader experimentation on hyper-parameters might gives us some hints on their effects and their interactions. Secondly we might find better ways to bound hyper-parameters together, as we did with the hidden coefficient h, and reduce the search space. Moreover we might try to employ smarter hyper-parameter search algorithm [84], rather than the simple grid search, to see if some one behaves better in this scenario.
Particularly interesting hyper-parameters to investigate are those coefficient weighting the contribution of losses of different kinds. Think of how we combined the Kullback-Leibler divergence, the structural loss and the value losses in Section 3.1.4. How all these different losses interact and affect the learning process is something that we still need to investigate. For instance for some consideration about the divergence term see [85].

Once we are able to have a proper hyper-parameterization and our model is endowed with the right mechanisms we expect it will be able to overfit. Therefore we might want to better investigate regularization methods. We are already employing a form of regularization by mean of the Kullback-Leibler divergence as required by Conditional VAEs, however we might benefit from investigating how it behaves and how it interacts with other forms of regularization such as the usuals Dropout[86] and Weight Decay [87].

Our implementation have been mostly driven by the need of obtaining a proof of concept and a framework to experiment with. It is efficient, reasonably general and publicly available. However it still not easy to be used from other users. Therefore a really interesting future work would be to refactor our implementation and properly document it in order to make it easily usable by the community.

Despite we faced some synthetic tasks on Arithmetic Expression Tree transduction and a real life task on Neural Machine Translation we employ them mostly as a test bench to compare different approaches. We did not really focus on solving such tasks achieving the best possible performance.
Hence, now that we have a working implementation, we can seriously tackle a real life task. This entails the need of employing much effort also in aspect not directly related to our model. For instance consider machine translation, the goodness of the trained model strongly depends on the goodness of the text preprocessing, in particular from the kind of words representation employed.

With our Conditional Variational Tree-Autoencoder we tried to model the whole joint distribution $P(Y|X)$. An alternative approach might be to model only $P(Y|x)$, basically obtaining a distribution of codes z only relative to y, rather than to the pair $< x, y >$. This might lead to considerably different results and should definitely be investigated.

References

[1] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, DOI: 10.1207/s15516709cog1402_1. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog1402_1. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1402_1.

[2] M. I. Jordan, "Chapter 25 - serial order: A parallel distributed processing approach," in *Neural-Network Models of Cognition*, ser. Advances in Psychology, J. W. Donahoe and V. P. Dorsel, Eds., vol. 121, North-Holland, 1997, pp. 471–495. DOI: https://doi.org/10.1016/S0166-4115(97)80111-2. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166411597801112.

[3] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://doi.org/10.1162/neco.1997.9.8.1735. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735.

[4] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: encoder-decoder approaches," *ArXiv e-prints*, Sep. 2014. arXiv: 1409.1259 [cs.CL].

[5] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *ArXiv preprint arXiv:1312.6114*, 2013.

[6] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[7] X. Huang, M.-Y. Liu, S. Belongie, and J. Kautz, "Multimodal unsupervised image-to-image translation," *ArXiv e-prints*, Apr. 2018. arXiv: 1804.04732 [cs.CV].

[8] A. Deshpande, J. Lu, M.-C. Yeh, M. J. Chong, and D. Forsyth, "Learning diverse image colorization," *ArXiv e-prints*, Dec. 2016. arXiv: 1612.01958 [cs.CV].

[9] X. Yan, J. Yang, K. Sohn, and H. Lee, "Attribute2image: conditional image generation from visual attributes," *ArXiv e-prints*, Dec. 2015. arXiv: 1512.00570.

[10] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *Tensorflow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[11] S. Kullback and R. A. Leibler, "On information and sufficiency," *Ann. Math. Statist.*, vol. 22, no. 1, pp. 79–86, Mar. 1951. DOI: 10.1214/aoms/1177729694. [Online]. Available: https://doi.org/10.1214/aoms/1177729694.

[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[13] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th international conference on Machine learning*, ACM, 2008, pp. 1096–1103.

[14] S. Rifai, G. Mesnil, P. Vincent, *et al.*, "Higher order contractive auto-encoder," in *Machine Learning and Knowledge Discovery in Databases*, D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 645–660, ISBN: 978-3-642-23783-6.

[15] A. Makhzani and B. J. Frey, "K-sparse autoencoders," *CoRR*, vol. abs/1312.5663, 2013. arXiv: `1312.5663`. [Online]. Available: `http://arxiv.org/abs/1312.5663`.

[16] J. Deng, Z. Zhang, E. Marchi, and B. Schuller, "Sparse autoencoder-based feature transfer learning for speech emotion recognition," in *2013 Humaine Association Conference on Affective Computing and Intelligent Interaction(ACII)*, vol. 00, Sep. 2014, pp. 511–516. DOI: `10.1109/ACII.2013.90`. [Online]. Available: `doi.ieeecomputersociety.org/10.1109/ACII.2013.90`.

[17] Q. V. Le, "Building high-level features using large scale unsupervised learning," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 8595–8598. DOI: `10.1109/ICASSP.2013.6639343`.

[18] Y. Bengio, L. Yao, G. Alain, and P. Vincent, "Generalized denoising auto-encoders as generative models," in *Advances in Neural Information Processing Systems*, 2013, pp. 899–907.

[19] C. Doersch, "Tutorial on variational autoencoders," *ArXiv preprint arXiv:1606.05908*, 2016.

[20] A. Roberts, J. Engel, and D. Eck, "Hierarchical variational autoencoders for music," in *NIPS Workshop on Machine Learning for Creativity and Design*, 2017.

[21] J. Walker, C. Doersch, A. Gupta, and M. Hebert, "An uncertain future: forecasting from static images using variational autoencoders," *ArXiv e-prints*, Jun. 2016. arXiv: `1606.07873 [cs.CV]`.

[22] R. Yeh, Z. Liu, D. B. Goldman, and A. Agarwala, "Semantic facial expression editing using autoencoded flow," *ArXiv e-prints*, Nov. 2016. arXiv: `1611.09961 [cs.CV]`.

[23] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: `http://yann.lecun.com/exdb/mnist/`.

[24] L. Devroye, "Sample-based non-uniform random variate generation," in *Proceedings of the 18th conference on Winter simulation*, ACM, 1986, pp. 260–265.

[25] Y. Miao and P. Blunsom, "Language as a latent variable: Discrete generative models for sentence compression," *ArXiv preprint arXiv:1609.07317*, 2016.

[26] T. D. Kulkarni, W. F. Whitney, P. Kohli, and J. Tenenbaum, "Deep convolutional inverse graphics network," in *Advances in neural information processing systems*, 2015, pp. 2539–2547.

[27] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," *ArXiv preprint arXiv:1611.01144*, 2016.

[28] C. J. Maddison, A. Mnih, and Y. W. Teh, "The concrete distribution: A continuous relaxation of discrete random variables," *ArXiv preprint arXiv:1611.00712*, 2016.

[29] S. S. Haykin, *Neural networks and learning machines*, 3rd. Pearson Upper Saddle River, 2009, ch. 4.

[30] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1, ch. 10.

[31] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural Networks*, vol. 18, no. 5, pp. 602–610, 2005, IJCNN 2005, ISSN: 0893-6080. DOI: `https://doi.org/10.1016/j.neunet.2005.06.042`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0893608005001206`.

[32] H. Jaeger, "Echo state network," *Scholarpedia*, vol. 2, no. 9, p. 2330, 2007.

[33] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[34] Y. Wu, M. Schuster, Z. Chen, *et al.*, "Google's neural machine translation system: bridging the gap between human and machine translation," *ArXiv e-prints*, Sep. 2016. arXiv: `1609.08144 [cs.CL]`.

[35] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *ArXiv preprint arXiv:1409.0473*, 2014.

[36] J. B. Pollack, "Recursive distributed representations," *Artificial Intelligence*, vol. 46, no. 1-2, pp. 77–105, 1990.

[37] P. Frasconi, M. Gori, and A. Sperduti, "A general framework for adaptive processing of data structures," *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 768–786, Sep. 1998, ISSN: 1045-9227. DOI: `10.1109/72.712151`.

[38] L. Bottou, "From machine learning to machine reasoning," *Machine Learning*, vol. 94, no. 2, pp. 133–149, Feb. 1, 2014, ISSN: 1573-0565. DOI: `10.1007/s10994-013-5335-x`. [Online]. Available: `https://doi.org/10.1007/s10994-013-5335-x`.

[39] M.-C. De Marneffe and C. D. Manning, "Stanford typed dependencies manual," Technical report, Stanford University, Tech. Rep., 2008.

[40] R. Socher, A. Perelygin, J. Wu, *et al.*, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 conference on empirical methods in natural language processing*, 2013, pp. 1631–1642.

[41] M. Iyyer, J. Boyd-Graber, L. Claudino, R. Socher, and H. Daumé III, "A neural network for factoid question answering over paragraphs," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 633–644.

[42] R. Socher, B. Huval, C. D. Manning, and A. Y. Ng, "Semantic compositionality through recursive matrix-vector spaces," in *Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning*, Association for Computational Linguistics, 2012, pp. 1201–1211.

[43] Y. Kim, C. Denton, L. Hoang, and A. M. Rush, "Structured attention networks," *ArXiv preprint arXiv:1702.00887*, 2017.

[44] A. Eriguchi, K. Hashimoto, and Y. Tsuruoka, "Tree-to-sequence attentional neural machine translation," *ArXiv preprint arXiv:1603.06075*, 2016.

[45] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *CoRR*, vol. abs/1503.00075, 2015. arXiv: `1503.00075`. [Online]. Available: `http://arxiv.org/abs/1503.00075`.

[46] X. Zhu, P. Sobihani, and H. Guo, "Long short-term memory over recursive structures," in *International Conference on Machine Learning*, 2015, pp. 1604–1612.

[47] A. Bies, M. Ferguson, K. Katz, *et al.*, "Bracketing guidelines for treebank ii style penn treebank project," *University of Pennsylvania*, vol. 97, p. 100, 1995.

[48] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The penn treebank," *Computational linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[49] X. Zhang, L. Lu, and M. Lapata, "Top-down tree long short-term memory networks," *ArXiv preprint arXiv:1511.00060*, 2015.

[50] C. Dyer, A. Kuncoro, M. Ballesteros, and N. A. Smith, "Recurrent neural network grammars," *ArXiv preprint arXiv:1602.07776*, 2016.

[51] L. Dong and M. Lapata, "Language to logical form with neural attention," *ArXiv preprint arXiv:1601.01280*, 2016.

[52] D. Alvarez-Melis and T. S. Jaakkola, "Tree-structured decoding with doubly-recurrent neural networks," 2016.

[53] R. Aharoni and Y. Goldberg, "Towards string-to-tree neural machine translation," *ArXiv preprint arXiv:1704.04743*, 2017.

[54] J. Bradbury and R. Socher, "Towards neural machine translation with latent tree attention," *CoRR*, vol. abs/1709.01915, 2017. arXiv: `1709.01915`. [Online]. Available: `http://arxiv.org/abs/1709.01915`.

[55] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *ArXiv preprint arXiv:1704.01696*, 2017.

[56] E. Parisotto, A. Mohamed, R. Singh, *et al.*, "Neuro-symbolic program synthesis," *CoRR*, vol. abs/1611.01855, 2016. arXiv: `1611.01855`. [Online]. Available: `http://arxiv.org/abs/1611.01855`.

[57] X. Chen, C. Liu, and D. Song, "Towards synthesizing complex programs from input-output examples," *ArXiv preprint arXiv:1706.01284*, 2017.

[58] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," *ArXiv preprint arXiv:1704.07535*, 2017.

[59] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *CoRR*, vol. abs/1802.03691, 2018. arXiv: `1802.03691`. [Online]. Available: `http://arxiv.org/abs/1802.03691`.

[60] R. Shin, A. A. Alemi, G. Irving, and O. Vinyals, "Tree-structured variational autoencoder," 2016.

[61] M. J. Kusner, B. Paige, and J. M. Hernández-Lobato, "Grammar variational autoencoder," *ArXiv preprint arXiv:1703.01925*, 2017.

[62] W. Jin, R. Barzilay, and T. Jaakkola, "Junction tree variational autoencoder for molecular graph generation," *ArXiv e-prints*, Feb. 2018. arXiv: `1802.04364`.

[63] H. Dai, Y. Tian, B. Dai, S. Skiena, and L. Song, "Syntax-directed variational autoencoder for molecule generation," in *International Conference on Machine Learning*, 2018.

[64] S. R. Bowman, L. Vilnis, O. Vinyals, *et al.*, "Generating sentences from a continuous space," *ArXiv preprint arXiv:1511.06349*, 2015.

[65] S. Semeniuta, A. Severyn, and E. Barth, "A hybrid convolutional variational autoencoder for text generation," *ArXiv e-prints*, Feb. 2017. arXiv: `1702.02390 [cs.CL]`.

[66] Z. Yang, Z. Hu, R. Salakhutdinov, and T. Berg-Kirkpatrick, "Improved variational autoencoders for text modeling using dilated convolutions," *ArXiv preprint arXiv:1702.08139*, 2017.

[67] B. Zhang, D. Xiong, J. Su, H. Duan, and M. Zhang, "Variational neural machine translation," *ArXiv preprint arXiv:1605.07869*, 2016.

[68] I. V. Serban, A. Sordoni, R. Lowe, *et al.*, "A hierarchical latent variable encoder-decoder model for generating dialogues.," in *AAAI*, 2017, pp. 3295–3301.

[69] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[70] R. Salakhutdinov and H. Larochelle, "Efficient learning of deep boltzmann machines," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 693–700.

[71] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Empirical Methods in Natural Language Processing (EMNLP)*, Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 1412–1421. [Online]. Available: `http://aclweb.org/anthology/D15-1166`.

[72] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, Nov. 2016, pp. 99–104. DOI: `10.1109/CCBD.2016.029`.

[73] M. Looks, M. Herreshoff, D. Hutchins, and P. Norvig, "Deep learning with dynamic computation graphs," *ArXiv e-prints*, Feb. 2017. arXiv: 1702.02181.

[74] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning.* MIT press Cambridge, 2016, vol. 1, ch. 8.

[75] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," *ArXiv e-prints*, May 2015. arXiv: 1505.00387.

[76] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, 2013, p. 3.

[77] D. P. Kingma and J. Ba, "Adam: a method for stochastic optimization," *ArXiv e-prints*, Dec. 2014. arXiv: 1412.6980.

[78] C. D. Manning, M. Surdeanu, J. Bauer, *et al.*, "The Stanford CoreNLP natural language processing toolkit," in *Association for Computational Linguistics (ACL) System Demonstrations*, 2014, pp. 55–60. [Online]. Available: http://www.aclweb.org/anthology/P/P14/P14-5010.

[79] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *ArXiv preprint arXiv:1607.04606*, 2016.

[80] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," *ArXiv e-prints*, Nov. 2012. arXiv: 1211.5063.

[81] E. Wang, Q. Zhang, B. Shen, *et al.*, "Intel math kernel library," in *High-Performance Computing on the Intel Xeon Phi: How to Fully Exploit MIC Architectures.* Cham: Springer International Publishing, 2014, pp. 167–188, ISBN: 978-3-319-06486-4. DOI: 10.1007/978-3-319-06486-4_7. [Online]. Available: https://doi.org/10.1007/978-3-319-06486-4_7.

[82] V. Pirogov and F. Gennady. (2016). Introducing dnn primitives in Intel Math Kernel Library, [Online]. Available: https://software.intel.com/en-us/articles/introducing-dnn-primitives-in-intelr-mkl (visited on 11/19/2018).

[83] O. Elmoustapha. (2017). Tensorflow optimizations on modern intel architecture, [Online]. Available: https://software.intel.com/en-us/articles/tensorflow-optimizations-on-modern-intel-architecture (visited on 11/19/2018).

[84] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 2951–2959. [Online]. Available: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf.

[85] I. Higgins, L. Matthey, A. Pal, *et al.*, "Beta-vae: Learning basic visual concepts with a constrained variational framework," 2016.

[86] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[87] A. Krogh and J. A. Hertz, “A simple weight decay can improve generalization,” in *Advances in neural information processing systems*, 1992, pp. 950–957.