UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL DE FINAL DE GRAU

GRAU EN ENGINYERIA FÍSICA

Divide and Conquer Networks

Author:
Alex Nowak Vila

Supervisor:
Joan Bruna Estrach

CILVR Lab
Center for Data Science, NYU

May 17, 2017

Universitat Politècnica de Catalunya

Abstract

Courant Intitute of Mathematical Sciences
Center for Data Science, NYU

Divide and Conquer Networks

by Alex Nowak Vila

This work considers the learning of algorithmic tasks by mere observation of input-output pairs. Rather than studying this as a black-box discrete regression problem with no assumption whatsoever on the input-output mapping, we concentrate on tasks that are amenable to the principle of *divide and conquer*, and study what are its implications in terms of learning.

This principle creates a powerful inductive bias that we exploit with neural architectures that are defined recursively and dynamically, by learning two scale-invariant atomic operations: how to *split* a given input into smaller sets, and how to *merge* two partially solved tasks into a larger partial solution. The resulting model is naturally divided in two phases: a discrete splitting phase that returns a binary tree over input elements, and a merge phase consisting in a differentiable routing mechanism. Our model can be trained in weakly supervised environments, namely by just observing input-output pairs, and in even weaker environments, using a non-differentiable reward signal. Moreover, thanks to the dynamic aspect of our architecture, we can incorporate the computational complexity as a regularization term that can be optimized by backpropagation.

We demonstrate the flexibility and efficiency of the Divide-and-Conquer Network on three combinatorial and geometric tasks: sorting, clustering and convex hulls. Thanks to the dynamic programming nature of our model, we show significant improvements in terms of generalization error and computational complexity.

Chapter 1

Introduction

Algorithmic tasks can be described as discrete input-output mappings defined over variable-sized inputs, but this “black-box" vision hides all the fundamental questions that explain how the task can be optimally solved and generalized to arbitrary inputs. A powerful framework that breaks into this vision is the principle that many tasks have some degree of scale invariance or self-similarity, meaning that the ability to solve the task for a certain input size is essentially all that is needed in order to solve it for larger sizes. This principle is the basis of recursive solutions and dynamic programming, and is ubiquitous in most areas of discrete mathematics, from geometry to graph theory. In the case of images and audio signals, invariance principles are also critical for success: CNNs 2.2.4 exploit both translation invariance and scale separation with multilayer, localized convolutional operators, which breaks the curse of dimensionality and brings the essential inductive bias explaining the success of CNNs. In our scenario of discrete algorithmic tasks, we build our model on the principle of *divide and conquer*, which provides us with a form of parameter sharing across scales.

While neural networks have been successful so far at providing flexible models for discrete regression and prediction tasks, mostly in Computer Vision (Krizhevsky, Sutskever, and Hinton, 2012), Natural Language Processing (Sutskever, Vinyals, and Le, 2014) and discrete Reinforcement Learning (Mnih et al., 2015), they are typically agnostic to complexity questions. Whereas some models are trained and tested at a fixed input/output scale (such as regression problems with generic fully connected neural networks), authors have explored ways to make training and testing less dependent of the input scale. The most prominent examples are Convolutional architectures, that exploit translation invariance to accept variable size inputs by averaging their predictions at the last layer; and Recurrent neural networks, that operate in an auto-regressive fashion to summarize any variable sized-input into a fixed-dimensional embedding. These two examples are paradigms of models whose complexity scales linearly with the input size.

Whereas CNN and RNN models define algorithms with linear complexity, attention mechanisms (Bahdanau, Cho, and Bengio, 2014) generally correspond to quadratic complexity, with notable exceptions (Andrychowicz and Kurach, 2016). This can result in a mismatch between the intrinsic complexity required to solve a given task and the complexity that is given to the neural network to solve it, which may impact its generalization performance. Our motivation is that learning cannot be ‘complete’ until these complexities match, and we start this quest by first focusing on problems for which the intrinsic complexity is well known and understood.

Following this lines, in this project we study these limitations and introduce a dynamic model that address the aforementioned issues.

The project is divided into three main blocks. The first one is a brief introduction to machine learning for those readers without background on this field. We explain the theoretical foundations of classical machine learning in the supervised setting 2.1 and introduce important concepts such as model generalization by providing the main popular techniques to address it. We also go through the recently popularized and exciting machine learning sub-field called *Deep Learning* . We provide examples of its main empirical successes and introduce the two more prominent architectures, namely, Convolutional Neural Networks (CNNs) (LeCun et al., 1989) and Recurrent Neural Networks (RNNs). Although CNNs are the main responsible architectures for the current deep learning hype among society, we focus on recurrent architectures because they are the ones that will come up later as a part of our developed model.

The next section introduces the designed Divide-and-Conquer Network (DCN) to the reader. In a nutshell, DCNs contain two modules: a *split* phase that is applied recursively and dynamically to the input in a coarse-to-fine way to create a hierarchical partition encoded as a binary tree; and a *merge* phase that traces back that binary tree in a fine-to-coarse way by progressively combining partial solutions 3.1. Each of these phases is parametrized by a single neural network, both variants of the sequence-to-sequence model introduced at the first section of this work. We also illustrate the versatility of the model by providing different ways of training it; differentiable standard training loss and non-differentiable rewards.

The last section is devoted to experimental results of the DCNs on algorithmic and geometric tasks with some degree of scale self-similarity: sorting, clustering and planar convex-hull. We show experimentally how the model is able to factorize problems in a recursive manner with minimal supervision and not only optimizing for the training accuracy but also for the overall runnning complexity.

Summary of Contributions:

We introduce a new dynamic architecture that incorporates the inductive bias from recursive tasks.

We show that it can be trained end-to-end with weak supervision, and whose average computational complexity can be optimized with gradient descent.

We provide empirical evidence that the dynamic programming principle can be efficiently learnt on tasks such as sorting, clustering and convex-hull.

Chapter 2

Introduction to Machine Learning

2.1 Machine Learning Background

Machine Learning is a broad subfield in computer science that consists in building algorithms that learn from data. Learning algorithms from data has become a fundamental tool to enhance nowadays technological systems.

Machine Learning models can be divided into two main categories:

Supervised Learning. The algorithm observes input-output pairs and its goal is to model the map from the inputs to the outputs.

Unsupervised Learning. The algorithm observes data and its goal is to find some structure and patterns.

In this work we will focus on the supervised setting because we will be addressing algorithmic tasks for which we have input-output pairs.

2.1.1 Supervised Learning Fundamentals

The goal of supervised learning is to determine a function $f : X \to Y$ from and input space X to an output space Y such that the value $f(x)$ is an accurate prediction of the corresponding output. That is, our goal is to predict a function $f \in F$ in some unknown function space that avoids memorization and instead generalizes concepts learned from a given set of examples.

Let's formalize the optimization problem we are now facing:

Definition 2.1.1 *Let $P \in$ Prob$(X \times Y)$ a probability distribution over the input-output space. Define the* **population risk** *as*

$$R(f) = \mathbb{E}_{(x,y)\sim P}[l(f(x), y)]$$

where $l(y, y')$ is a cost function that measures how close $y, y' \in Y$ are one from each other.

The **oracle problem** to solve is the following:

$$\min_{f \in F} R(f)$$

However, there are two main issues with the current formulation of the optimization problem.

First, we are facing a variational minimization problem in an unknown function space F. In order to make this problem amenable to gradient descent, we have to restrict ourselves to a differentiable parametric family of functions $F_\Theta = \{f_\theta : \theta \in \Theta\}$.

The second problem is the untractability of the population risk $R(f)$. This can happen because the distribution P is unknown and we only have a finite number of samples from P or because the integral is computationally intractable. On both cases $R(f)$ will be approximated by samples.

Definition 2.1.2 *Let* $(x_1; y_1); (x_2; y_2); : : : ; (x_n; y_n)$ 2 X Y *i.i.d samples from* P. *Define the* **empirical risk** *as*

$$R_n(f) = \frac{1}{n}\sum_{i=1}^{n} l(f(x_i); y_y)$$

Plugging the aforementioned changes in the original formulation, we get to the following optimization problem:

$$\min_{2} R_n(f\quad)$$

where the empirical risk $R_n(f)$ is computed using samples from an available dataset $D = f(x_i; y_i) : i = 1; : : : ; ng$ of size n.

2.1.2 Generalization gap

The family of function spapce F should be determined with three competing goals in mind.

1. F should contain prediction functions that contain enough **capacity** to achieve low empirical risk over the training set D to avoid underfitting the data.

2. The gap between $R(f)$ $R_n(f)$ between the empirical and population risk should be small for all f 2 F. Generally, one way to decrease this gap is to increase the number of samples n. However, this gap increases when one uses richer families of functions. This latter fact puts the second goal at odds with the first.

3. F should be chosen such that one can efficiently solve the optimization problem. This difficulty increases with the size of the dataset and the capacity of the function family.

The gap between population and empirical risk has been widely studied in the domain called Statistical Learning Theory ("Introduction to Statistical Learning Theory"). The following theoretical results are presented for risks with unitary range for simplicity.

A direct application of the Hoeffdings inequality (Hoeffding, 1963) guarantees that with probability at least 1 the following bound holds:

$$jR(f) \quad R_n(f)j \quad \sqrt{\frac{1}{2n}\log\left(\frac{2}{\ }\right)} \text{ for a given } f \ 2 \ F \tag{2.1}$$

The bound 2.1 gives the intuitive explanation that the gap decreases with the number of examples. Hoewever, the latter bound holds for a fixed function f, and in the machine learning setting we care about this gap for all functions in F.

Vapnik-Chervonenkis (VC) theory addresses this problem by bounding the supremum of empirical process indexed by a family of functions (Vapnik and Chervonenkis, 2015). The main concept is the so-called VC dimension $vc(F)$, which measures the representational power of the family by counting the cardinality of the

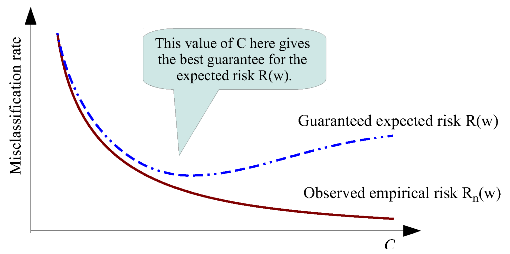


FIGURE 2.1: Illustration of structural risk minimization. Figure extracted from (Bottou, Curtis, and Nocedal, 2016)

largest set than cap be shattered by the functions of the family. VC dimension is closely related to the number of parameters describing every member of the family.

Mathematically, with the VC dimension measuring the capacity, one can establish one of the most important results in learning theory:

$$\sup_{f \in F} |R(f) - R_n(f)| \leq O\left(\sqrt{\frac{1}{2n}\log\left(\frac{2}{\delta}\right) + \frac{vc(F)}{n}\log\left(\frac{n}{vc(F)}\right)}\right) \quad (2.2)$$

This bound gives the intuitive explanation that the upper bound of the gap can be controlled both by the capacity of the class of functions (VC dimension) and the size of the dataset. If the class complexity increases the gap goes up, and if the number of samples increases the gap shrinks. The dependence is given by 2.2.

Statistical learning bounds as 2.2 are theoretically interesting but are rarely used in practice because the gap is normally estimated by *cross-validation* experiments. However, there are some techniques used in practice to shrink the gap. These techniques are normally referred as **regularization**.

2.1.3 Regularization

Structural Risk Minimization

One of the most classical common ways to choose a class of functions is using the so-called *structural risk minimization* (Vapnik, 1998). The idea is to define a *structure*, i.e, a collection of nested function families. Normally this structure is defined in terms of a preference function $R(f)$ and an *hyperparameter* C by $F_C = \{f \in F : R(f) \leq C\}$. The preference function $R(\cdot)$ is a scalar function defined over F such that restricting the optimization problem to F_C, improves the minimum of the empirical risk in the large C regime, but after some point it typically increases the gap between population and empirical risk; see 2.1.

The constraint minimization problem is normally viewed through the Lagrangian formulation as $R_n(f) + \lambda R(f)$.

One typical choice of $R(f_\theta)$ is the $\|\theta\|_2$ (L^2 norm of the parameters) when considering a parametric family of functions F_θ.

Optimization Algorithm Regularization

The choice of the optimization algorithm is one of the most important parts of a machine learning model both for efficiency and regularization. As explained in 2 we must consider parametric class of functions to be able to optimize it by gradient descent.

The **gradient descent** algorithm is a first order optimization method described in 2.3 that computes the gradient of the empirical risk and updates the parameters to the direction of steepest descent. The magnitude of the update is controlled by the **learning rate** $_t$ that may depend on the iteration. The optimal learning rate strongly depends on the optimization landscape and it is normally tunned by experimentation.

$$_{t+1} \qquad _t \qquad _t \, r R_n(\, _t) = \, _t \qquad _t \, r \frac{1}{n}\sum_{i=1}^{n} f(\, _{[i]}; \, _t) \tag{2.3}$$

The major problem with algorithm 2.3 is the efficiency of the gradient computation both in running complexity and memory. Both complexities are linear $O(n)$ in the size of the dataset, which is extremely costly given that the dataset sizes are usually extremely large. This is why the gradient descent method is never used in machine learning applications.

Instead, the main algorithm used is **stochastic gradient descent** (SGD) (Robbins and Monro, 1951). This gradient descent method is called stochastic because at every time step the gradient is computed using only one sample of the dataset. The update step reads

$$_{t+1} \qquad _t \qquad _t \, r f(\, _{[i_t]}; \, _t) \tag{2.4}$$

where the index i_t is chosen uniformly randomly from $f1; : : : ; ng$. In this case, both the running complexity and memory are constant $O(1)$.

SGD has two main advantadges from vanilla gradient descent (Hardt, Recht, and Singer, 2015):

Train faster.

Generalize better.

However, SGD also has some drawbacks in its original formulation.

1. It is sequential in nature, so this puts a limitation when brough to high performance computing because of the lack of parallelization.

2. Although the algorithm converges faster than other methods, the amount of noise in the gradient approximation makes the algorithm likely to be stuck in bassins as shown in (Bottou, Curtis, and Nocedal, 2016).

In order to overcome these issues, a bunch of SGD variants have appeared in the optimization literature. One kind of variants addresses both issues 1, 2 by estimating the gradient by a batch of samples instead of a single one. That is the so-called **batched stochastic gradient descent** (BSGD). The update then reads

$$^{bsgd}_{\;t+1} \qquad _t \qquad \frac{_t}{jS_tj} \sum_{i_t 2 S_t} r f(\, _{[i_t]}; \, _t) \tag{2.5}$$

where now the gradient is approximated by a random batch $S_t = f i_1; : : : ; i_{jS_tjg}$ of samples instead of one. The advantadges of **??** is that the variance is reduced by a factor proportional to the size of the batch, and the algorithm may be parallelized by the same factor.

Another algorithms use extra information from the previous iterates both to reduce the variance and to accelerate the algorithms (Tieleman and Hinton, 2012), (Duchi, Hazan, and Singer, 2011) .

2.2 Deep Learning

2.2.1 Learning representations

One of the most important aspects of a machine learning is how the data is represented. One trivial example is the following: imagine that you want to design a model that learns the XOR binary operation. The input of the model are two bits and the output the resulting XOR operation. Imagine that you are limited to only linear models, i.e, your model will search an hyperplane such that can solve the classification task. It is a simple exercise to show that this binary classification cannot be linearly separated in the natural input space of two dimensions because the elements of the same class always lie at the opposite corner of the unit square. However, if the input is embedded to a three dimensional space, then the problem will become linearly separable. A more illustrative example may be the use a simple logistic regression model to recommend cesarean delivery. In this case, the model does not examine the patient directly but uses relevant information from the doctor such as the presence or absence of a uterine scar. Each piece of information included in the representation of the patient is known as a *feature*. Logistic regression learns how each of these features of the patient correlates with various outcomes. However, it cannot influence the way that the features are defined in any way. If logistic regression was given an MRI scan of the patient, rather than the doctor's formalized report, it would not be able to make useful predictions because of the amount of variability on the raw MRI scan image that has nothing to do with the prediction task. Individual pixels in an MRI scan have negligible correlation with any complications that might occur during delivery. Bengio, Goodfellow, and Courville, 2015.

Classical machine learning methods use hand-crafted features from the data to feed to the learning model because otherwise the model wouldn't be able to disentangle important *factors of variation* on the data.

A major issue in many real-world machine learning applications is that many of the factors of variation influence every single piece of data we are able to observe. For instance, the shape of a car's silhouette depends on the viewing angle, and this concerns all pixels in an image. This factor disentanglement is extremely challenging or impossible for even the most advanced signal processing tools. How to proceed when the data representation problem is as hard as the prediction task at hand (in the previous example, the uterine scan prediction)?

Deep Learning (LeCun, Bengio, and Hinton, 2015) is the class of machine learning models that solves this problem by learning such representations together with the original classification task.

2.2.2 Brief history of deep learning

Although deep learning is thought as a new technology, its history goes back to the so-called *artificial neural networks* or simply *neural networks*. Deep learning models structure is inspired by the biological brain, however, they are generally not designed to be realistic models of biological function. One of the pioneer works was the invention of the *perceptron* (Rosenblatt, 1958), a simple mathematical model for a neuron. The perceptron and its continuous variant ADALINE (Widrow and Hoff, 1960) were the first models that could learn the weights defining the categories given examples of inputs from each category. A perceptron is basically a linear model followed by a non-linearity called *activation*:

$$P(\mathbf{x}) = \left(\sum_{i=1}^{n} w_i x_i\right) \tag{2.6}$$

where used to be a sigmoid or a step function. These linear models, however, are quite limited in terms of representative power as has been shown for the XOR classification problem. Motivated by the strucural arrangement of neurons on the brain and the need for more representative power, the *multilayer perceptron* (MLP) was designed, which consisted in several perceptrons stacked in a layered strucure. It has been shown that MLP are universal approximators for generic functions (Cybenko, 1989) even with a single layer. However, the downside is that you need exponentially large number of parameters.

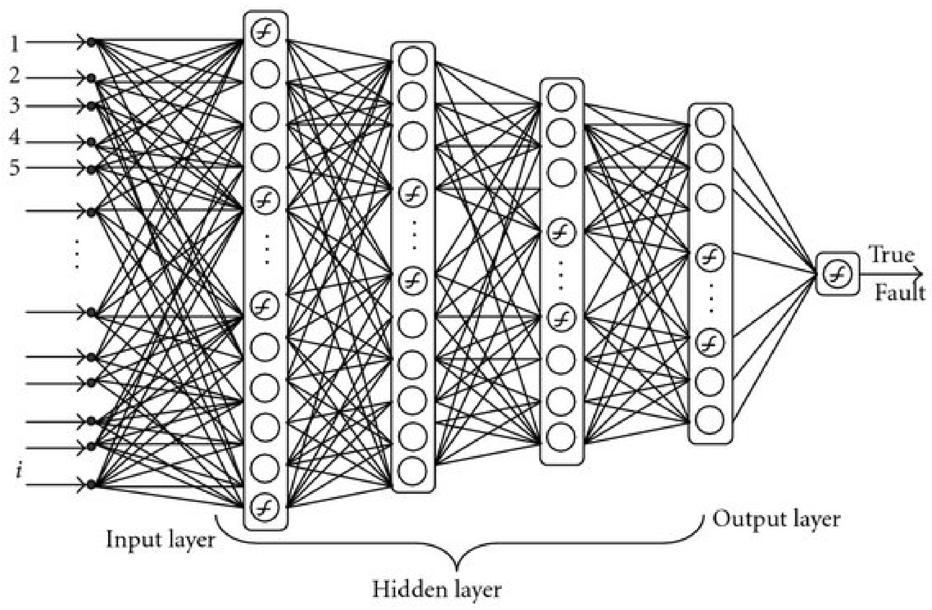


FIGURE 2.2: Multilayer perceptron of 2 layers for binary classification.

MLPs were a popular machine learning solution in the 1980s finding applications in diverse fields such as speech recognition, image recognition, and machine translation software, but have since the 1990s faced strong competition from the much simpler *support vector machines* (Cortes and Vapnik, 1995).

2.2.3 Backpropagation

Deep learning models are optimized through a method called **backpropagation** (Le Cun et al., 1990).

Backpropagation algorithm was motivated to train multi-layered neural networks by simply recursively applying the chain rule throughout the network.

First, the algorithm forwards the input storing all intermediate activations of the deep network for all nodes and computes the loss. Then, partial derivatives are recursively computed from the top node to the input applying the chain rule. The derivative of the loss function L with respect to the node $u^{(i)}$ is computed dynamically using the derivatives from its parent nodes and the derivative of the function connecting them. The recurrence reads:

$$\frac{\partial L}{\partial u^{(i)}} = \sum_{(i,j) \in Pa(u^{(i)})} \frac{\partial L}{\partial u^{(j)}} \frac{\partial u^{(j)}}{\partial u^{(i)}} \tag{2.7}$$

where $Pa(u^{(i)})$ denotes the set of parent nodes of the $u^{(i)}$. Once the derivatives are computed the weights are updated using the chosen optimization algorithm, usually stochastic gradient descent 2.4.

2.2.4 Convolutional Neural Networks (CNNs)

Definition

Convolutional neural networks (CNNs) (LeCun et al., 1989) are deep neural networks specially designed to handle natural images as inputs. CNNs have been tremendously successful in practical applications.

Convolutional networks structure imposes a prior of **spatial stationarity** and builds a **multiscale representation** by changing the matrix vector multiplication in MLPs by *convolutions*.

Convolutions are linear operators (hence, a special case of matrix vector multiplication in the discrete setting) with shared weights. CNNs replace hidden layers of perceptrons by convolutional maps, each one with its corresponding kernel, followed by an activation function.

This operation is cascaded in a multilayered structure and occasionally subsampled to reduce maps dimensionality. Due to the convolutional structure, CNNs deal with the input stationarity by sharing the network weights across space and building a multiscale representation of the input. Although the multiscale nature may not be apparent to the reader, the activations of the network at a given location are conditioned to bigger crops of the image when cascading on the depth dimension because of the compactness of the kernels support. This implicitly creates a multiscale structure. It is one of the key features of the network, allowing to build sparse representations similar to *wavelets* (Mallat, 1989) prior to the subsampling stage as has been shown in (Bruna and Mallat, 2013).

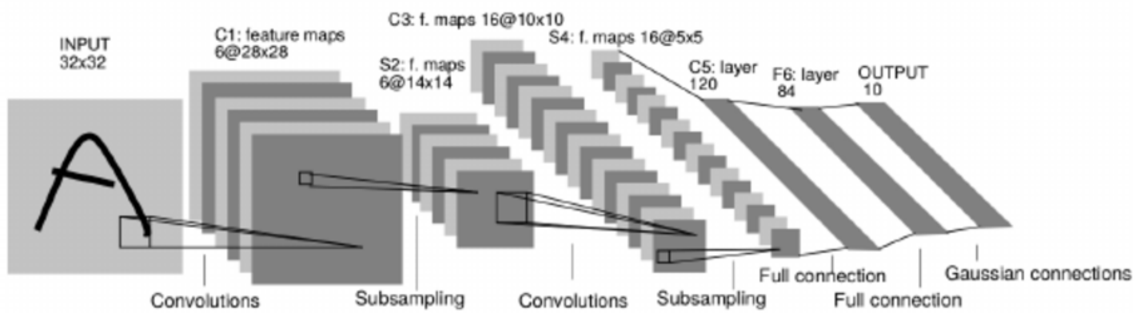


FIGURE 2.3: **LeNet-5**: First convolutional neural network designed by pioneer Professor Yann LeCun (Le Cun et al., 1990) for digit recognition.

Although convolutional neural networks were invented in the 80s (Le Cun et al., 1990) and were already successfully used for image recognition and check reading (Le Cun, Bottou, and Bengio, 1997), the inflection point for CNNs excitement started in 2012 when (Krizhevsky, Sutskever, and Hinton, 2012) won the ImageNet Challenge (Russakovsky et al., 2015) using deep learning by a large accuracy gap compared to other competitors.

The reason people lost interest for CNNs in the 90s and 00s was because of the insufficient computational resources available at that time to make deep learning algorithms work. Nowadays, there are speciallized deep learning libraries like Tensorflow (Abadi et al., 2016) (opensource tool used by Google AI research team) or Pytorch (recent library developed by the FAIR (Facebook Artificial Intelligence Research)) that are able to parallelize computation in GPUs.

Applications

CNNs are the main responsible for the majority of state-of-the-art applications of artificial intelligence for *computer vision*. They hold state-of-the-art benchmark results in *Image Recognition* and *Image Segmentation*.

2.2.5 Recurrent Neural Networks (RNNs)

Definition

A **Recurrent Neural Network** (RNNs) is an *autoregressive* neural network designed to deal with sequential input structures, i.e, *time-series*. Aurtoregressive here means that the model is conditioned on his own previous states. RNNs are the non-linear version of the famous linear $AR(\cdot)$ models used for linear prediction on time-series. One important example is *natural language*. That is, they model probability distributions that "easily" factorize in an autoregressive way :

$$P(X) = P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | x_{i-1}, x_{i-2}, \ldots, x_1) \tag{2.8}$$

The above relation holds for every probability distribution, however, in a time-series, the dependence decays with the distance $|i - j|$ in a similar way a natural image has a local dependence.

Given an input sequence $X = (x_1, \ldots, x_n)$, and an input *hidden state* or *hidden representation* h_0, the model produces an output sequence $Y = (y_1, \ldots, y_n)$ by applying:

$$\begin{cases} h_i = f(h_{i-1}, x_i) \\ y_i = g(h_i) \end{cases} \tag{2.9}$$

where $f(\cdot, \cdot)$ is a neural network that takes the previous hidden state and current input and produces the new hidden state. And $g(\cdot)$ is an output neural network that takes the current hidden state and outputs the new element in the output sequence; see Figurec 2.4.

Types of RNNs

The first model of RNN was developed by (Elman, 1990), (Jordan, 1997). The model was akin to the perceptron 2.6:

$$\begin{cases} h_i = \sigma_h(W_{hh} h_{i-1} + W_{hx} x_i + b_h) \\ y_i = \sigma_y(W_{yh} h_i + b_y) \end{cases} \tag{2.10}$$

where σ_h, σ_y are activation functions.

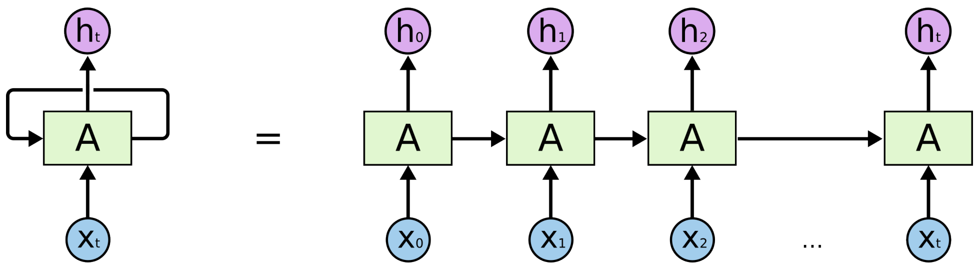


FIGURE 2.4: A Recurrent Neural Networks is a network that calls itself at every time-step. You can view it recursively as left, or unfolded in time as the right hand side of the image. Image courtesy of Chris Olah

The form of 2.10 has several limitations. The most important one is gradient norm explosion and information leakage from further states. The gradients coming from further states are likely to vanish or saturate due to the geometrical structure of the recurrence. Also, the model has difficulty to retrieve information from much previous states because at every iteration the model is overwritting the hidden state.

This problems are mainly solved by **LSTMs** (Hochreiter and Schmidhuber, 1997) and the simplest current version **GRUs** (Chung et al., 2014).

We will only introduce the GRU cell strucure for the sake of simplicity. The main idea is to introduce *gating*, which gives the neural network the possibility to decide whether to pass, store or overwrite information from the previous state to the new one.

$$\begin{cases} z_i = \sigma_g(W_{zh}h_{i-1} + W_{zx}x_i + b_z) \\ r_i = \sigma_g(W_{rh}h_{i-1} + W_{rx}x_i + b_r) \\ h_i = z_i \odot h_{i-1} + (1 - z_i) \odot \sigma_h(W_{hh}(r_i \odot h_{i-1}) + W_{hx}x_i + b_h) \end{cases} \tag{2.11}$$

where $\odot$ stands for an elementwise product. In this case, there are two gates z_i and r_i that eases gradient flow through the cell and allows all aforementioned procedures.

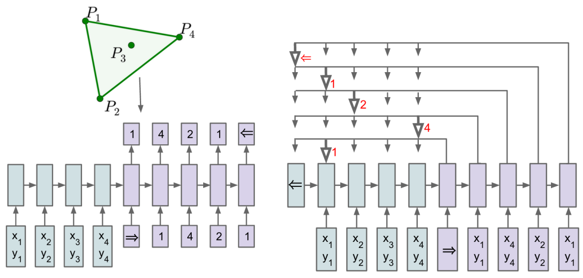
Sequence-to-sequence Models

A **sequence-to-sequence** model receives an input sequence $X = \{x_1, \ldots, x_n\}$ of length n and outputs a sequence $Y = \{y_1, \ldots, y_{m(n)}\}$ of variable length k. The input and output sequence elements belong to different dictionaries ($X \in \mathcal{X}^n$ and $Y \in \mathcal{Y}^{m(n)}$).

This is done in two steps. The **encoder** builds an element-wise representation $\mathbf{e} = (e_1, \ldots, e_n)$ of X using an autoregressive model by passing a RNN cell $f_{\text{enc}}(\cdot, \cdot)$. The model reads:

$$e_i = f_{\text{enc}}(e_{i-1}, x_i) \quad \forall i \in (1, \ldots, n) \tag{2.12}$$

The vector e_n is a **global representation** of the input sequence built by the autoregressive model. The **decoder** uses e_n to decode an output sequence in the output dictionary $\mathcal{Y}$ with another autoregressive model using another RNN cell $f_{\text{dec}}(\cdot, \cdot)$. The model reads:



P_1
P_4
P_3
P_2
1 4 2 1 ⇐
x_1 y_1 x_2 y_2 x_3 y_3 x_4 y_4 ⇒ 1 4 2 1
⇐ 1 2 4 1
⇐
x_1 y_1 x_2 y_2 x_3 y_3 x_4 y_4 ⇒ x_1 y_1 x_4 y_4 x_2 y_2 x_1 y_1

FIGURE 2.5: An encoding RNN converts the input sequence to a code (blue) that is fed to the generating network (purple). At each step, the generating network produces a vector that modulates a content-based attention mechanism over inputs ([5, 2]). The output of the attention mechanism is a softmax distribution with dictionary size equal to the length of the input.

$$\begin{cases} d_0 = e_n; \quad y_0 = \text{start} \\ d_i = f_{\text{dec}}(d_i \;\, 1; y_i \;\, 1) \quad i \; 2 \; (1; :::; n) \\ p^i = \text{softmax}(g(d_i)) \quad i \; 2 \; (1; :::; n) \\ y_i = \text{argmax}(p^i) \quad i \; 2 \; (1; :::; n) \end{cases} \tag{2.13}$$

where $g(\ )$ sends decoder hidden states d_i to vectors of dimensionality $jYj + 2$, where extra tokens have been added to define both start and end of sequence. This model has linear running complexity $O(n)$.

This model only fetches information from the input sequence by the global representation vector e_n, and then this information may be partially lost when running the decoder cell. One useful and powerful idea is to increase the complexity of the model to $O(m(n)n) \quad O(n^2)$ by letting the decoder to look at the input element-wise representations e_j at any time during the decoder process.

The **content based attention** model computes pairwise interactions between the current decoder state d_i and encoder states e_j and use it to propagate extra information to the new decoder state that will be fed to next step. The attention reads:

$$\begin{cases} u^i_j = e_j^T d_i \quad j \; 2 \; (0; :::; n) \\ a^i_j = \text{softmax}(u^i_j) \quad j \; 2 \; (0; :::; n) \\ d^0_i = \sum_{j=1}^{n} a^i_j e_j \\ d_i = W_c[d_i; d^0_i] \end{cases} \tag{2.14}$$

where e_0 is a vector of learnable parameters of the model specifying state for the end token This model works for fixed size dictionaries Y, however, in some problems the output dictionary depends on **x**. In this case, both vanilla sequence-to-sequence models and content based attention variant cannot be used because g is a parametrized functional to R^{jYj} so the output dictionary must be fixed.

Pointer networks address this issue when $Y = fx_i g_i$, i.e, the output dictionary is the set of input sequence elements; see 2.5. This is solved by using the attention vectors $\mathbf{a}^i = (a^i_1; :::; a^i_n)$ as pointers to the input insted of being used to feed a weighted input sum to the next decoder state. The content based attention mechanism is replaced by:

$$\begin{cases} u_j^i = e_j^T d_i & j \in (0, \ldots, n) \\ p^i = a^i = \mathrm{softmax}(u^i) \end{cases} \tag{2.15}$$

and there is no output g function in this case, so the model is completely dictionary input independent and only decides over input indexes.

The output of the pointer network is an attention matrix $A \in \mathbb{R}^{n \times n}$ where the columns are the p^i indices probabilities.

$$A_k = \begin{bmatrix} \vdots & \vdots & & \vdots \\ p_1 & p_2 & & p_n \\ \vdots & \vdots & & \vdots \end{bmatrix} \in \mathbb{R}^{n \times n} \tag{2.16}$$

The output indexes will be computed by taking the **argmax** along columns. The target matrix $T \in \mathbb{R}^{n \times n}$ will be a binary matrix where at the $i-th$ column there will be a 1 at the index of y_i in the input and zero otherwise. The 'end' tokens correspond to the first row.

The loss function will be the sum over columns of cross-entropies between A and T columns.

Chapter 3

Divide and Conquer Networks

Following the motivations stated at 1, we design the so-called Divide-and-Conquer Networks.

Divide-and-Conquer Networks contain two modules: a *split* phase that is applied recursively and dynamically to the input in a coarse-to-fine way to create a hierarchical partition encoded as a binary tree; and a *merge* phase that traces back that binary tree in a fine-to-coarse way by progressively combining partial solutions; see Figure 3.1. Each of these phases is parametrized by a single neural network that is applied recursively at each node of the tree, enabling parameter sharing across different scales and leading to good sample complexity and generalisation. Since the split phase operates with sets, we use the so-called *set2set* models (**vinyals_set2set**) for the split, and since the merge phase routes intermediate outputs from its inputs, we use the pointer net attention mechanism (**ptrnet**).

In this work, we attempt to incorporate the scale-invariance prior with the desiderata to only require weak supervision. In particular, we consider two setups: learning from input-output pairs, and learning from a non-differentiable reward signal. Since our split block is inherently discrete, we resort to policy gradient estimators to train the split parameters, while using standard backpropagation for the merge phase; see Section 3.4. An important benefit of our framework is that the architecture is dynamically determined, which suggests using the computational complexity as a regularization term. As shown in the experiments, computational complexity is good proxy for generalisation error in the context of discrete algorithmic tasks.

We demonstrate our model on algorithmic and geometric tasks with some degree of scale self-similarity: sorting, clustering and planar convex-hull (see the experiments section 4 if not familiar with any of these tasks). Our numerical results on these tasks reaffirm the fact that whenever the problem has scale invariance, then exploiting it leads to improved generalization and computational complexity over previous approaches.

3.1 Related Work

Using neural networks to solve algorithmic tasks is an active area of current research, but its models can be traced back to context free grammars (Fanty, 1994). In particular, dynamic learning appears in works such as (Pollack, 1991) and (Tabor, 2000).

The current research in the area is dominated by Recurrent Neural Networks (Joulin and Mikolov, 2015; Grefenstette et al., 2015), LSTMs (Hochreiter and Schmidhuber, 1997), sequence-to-sequence neural models (Sutskever, Vinyals, and Le, 2014; Zaremba and Sutskever, 2014), attention mechanisms (Vinyals, Fortunato, and Jaitly,

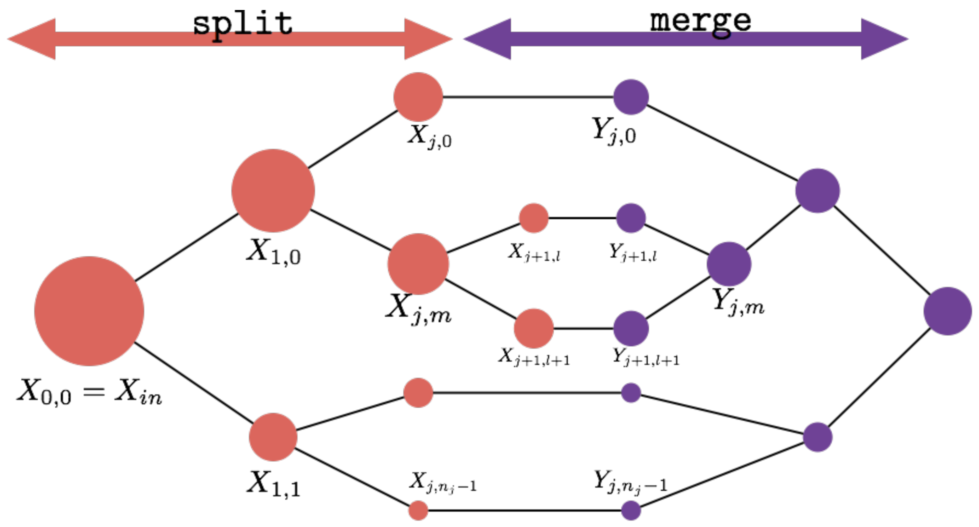


FIGURE 3.1: Divide and Conquer Network. The split phase is determined by a dynamic neural network S that splits each incoming set into two disjoint sets: $f X_{j+1;l}; X_{j+1;l+1} g = S(X_{j;m})$, with $X_{j;m} = X_{j+1;l} \; t \; X_{j+1;l+1}$. The merge phase is carried out by another neural network M that combines two partial solutions into a solution of the coarser scale: $Y_{j;m} = M(Y_{j+1;l}; Y_{j+1;l+1})$; see Section **??** for more details.

2015; Andrychowicz and Kurach, 2016) and explicit external memory models (Weston, Chopra, and Bordes, 2014; Sukhbaatar et al., 2015; Graves, Wayne, and Danihelka, 2014; Zaremba and Sutskever, 2015). We refer the reader to (Joulin and Mikolov, 2015) and references therein for a more exhaustive and detailed account of related work.

Amongst these works, we highlight some that are particularly relevant to us. Neural GPU (Kaiser and Sutskever, 2015) defines a neural architecture that acts convolutionally with respect to the input and is applied iteratively $o(n)$ times, where n is the input size. It leads to fixed computational machines with total (n^2) complexity. Neural Programmer-Interpreters (Reed and Freitas, 2015) introduce a compositional model based on a LSTM that can learn generic programs. It is trained with full supervision using execution traces. Directly related, (Cai, Shin, and Dawn, 2017) incorporates recursion into the Neural Programming Interpreter to enhance its capacity and provide learning certificates in the setup where recursive execution traces are available for supervision. Hierarchical attention mechanisms have been explored in (Andrychowicz and Kurach, 2016). They improve the complexity of the model from $o(n^2)$ of traditional attention to $o(n \log n)$, similarly as our models, but they are trained very differently, using REINFORCE. Finally, Pointer Networks (Vinyals, Fortunato, and Jaitly, 2015; Vinyals, Bengio, and Kudlur, 2015) modify classic attention mechanisms to make them amenable to adapt to variable input-dependent outputs, and illustrate the resulting models on geometric algorithmic tasks. It belongs to the (n^2) category class.

3.2 Problem Setup

3.2.1 Scale Invariant Tasks

We consider tasks consisting in a mapping T between a variable-sized input set $X = \{x_1, \dots, x_n\}, x_j \in \mathcal{X}$ into an ordered set $Y = \{y_1, \dots, y_{m(n)}\}, y_j \in \mathcal{Y}$. This setup includes problems where the output size $m(n)$ differs from the input size n, and also problems where Y is a labeling of input elements. In particular, we will study in detail the case where $\mathcal{Y} \subseteq \mathcal{X}$ (and in particular $Y \subset X$).

As discussed earlier, we are interested in tasks that are self-similar across scales, meaning that T can be decomposed as

$$\begin{aligned} \forall n, \forall X, |X| = n, \; T(X) &= M(T(S_1(X)), \dots, T(S_s(X))), \\ |S_j(X)| &< n, \; \cup_{j \leq s} S_j(X) = X, \end{aligned} \tag{3.1}$$

where both M and $S = (S_1, \dots, S_s)$ are *independent of* n. Under this assumption, the task T can thus be solved by first splitting the input into s strictly smaller subsets $S_j(X)$, solving T on each of these subsets, and finally merging the corresponding outputs together. In order words, T can be solved by recursion. A particularly simple and illustrative case is the binary setup with $s = 2$ and $S_1(X) \cap S_2(X) = \emptyset$, that we will adopt in the following for simplicity.

Many algorithmic and geometric tasks admit the decomposition (3.1). It corresponds to the well-known principle of divide and conquer. Besides providing the road-map to understand the asymptotic behavior of the task as the size of the input grows, such decompositions typically lead to optimal computation complexity, in the sense that they provide the most efficient reuse of operations.

If T is presumed to satisfy (3.1), we can thus attempt to learn T by learning S and M, respectively the split and merge steps. Since S and M are independent of n, one may hope for superior generalization performance than a model that is agnostic to the scale-invariance property.

3.2.2 Weakly Supervised Recursion

Our first goal is to learn how to perform T for any size n, by observing only input-output example pairs $(X^l, Y^l), l = 1 \dots L$. Throughout this work, we will make the simplifying assumption of binary splitting ($s = 2$), although our framework extends naturally to more general versions. Given an input set X associated with output Y, we first define a split phase that breaks X into a disjoint partition tree $\mathcal{P}(X)$:

$$\mathcal{P}(X) = \{X_{j,k}, 0 \leq j < J, 0 \leq k < n_j\}, \text{ with } X_{j,k} = X_{j+1,2k} \sqcup X_{j+1,2k+1}, \tag{3.2}$$

and $X = X_{1,0} \sqcup X_{1,1}$. This partition tree is obtained by recursively applying a trainable binary split module S_θ:

$$\begin{aligned} \{X_{1,0}, X_{1,1}\} &= S_\theta(X), \text{ with } X = X_{1,0} \sqcup X_{1,1}, \\ \{X_{j+1,2k}, X_{j+1,2k+1}\} &= S_\theta(X_{j,k}), \text{ with } X_{j,k} = X_{j+1,2k} \sqcup X_{j+1,2k+1}, \; (j < J, k \leq 2^j). \end{aligned} \tag{3.3}$$

Here, J indicates the number of *scales* or depth of recursion that our model applies for a given input X, and S_θ is a neural network that takes a set as input and produces a binary, disjoint partition as output. Eq. (3.3) thus defines a hierarchical partitioning of the input that can be visualized as a binary tree; see Figure 3.1. This binary tree is data-dependent and will therefore vary for each input example, dictated by the current choice of parameters for S_θ. The scale J can be either be a hyperparameter that is set constant for all input examples, or determined dynamically for each input based on the size of the leaves, such that $\forall k, |X_{J,k}| \leq C$.

The second phase of the model takes as input the binary tree partition $P(X)$ determined by the split phase and produces an estimate $\hat{Y}$. We follow the dynamic computation graph determined by the tree using a second trainable block, the merge module M_ϕ:

$$
\begin{aligned}
Y_{J,k} &= \bar{M}_\phi(X_{J,k})\ , (1 \leq k \leq 2^J)\ , \\
Y_{j,k} &= M_\phi(Y_{j+1,2k}, Y_{j+1,2k+1})\ , (1 \leq k \leq 2^j, j < J)\ , \\
\hat{Y} &= M_\phi(Y_{1,0}, Y_{1,1})\ .
\end{aligned} \tag{3.4}
$$

Here we have denoted by $\bar{M}_\phi$ the atomic block that transforms inputs of size $\leq C$ at the leaves of the split tree, and M_ϕ is a neural network that takes as input two (possibly ordered) inputs and merges them into another (possibly ordered) output. In the setup where $Y \subseteq X$, we further impose that

$$
Y_{j,k} \subseteq Y_{j+1,2k} \cup Y_{j+1,2k+1}\ , \tag{3.5}
$$

to guarantee that the computation load does not diverge with J.

3.2.3 Learning from non-differentiable Rewards

Another setup we can address with (3.1) consists in problems where one can assign a cost (or reward) to a given partitioning of an input set. In that case, Y encode the labels assigned to each input element. We also assume that the reward function has some form of self-similarity, in the sense that one can relate the reward associated to subsets of the input to the total reward.

In that case, (3.3) is used to map an input X to a partition $P(X)$, determined by the leaves of the tree $\{X_{J,k}\}_k$, that is evaluated by an external black-box evaluator returning a cost $L(P(X))$. For instance, one may wish to perform graph coloring satisfying a number of constraints. In that case, the cost function would assign

$$
L(P(X)) = \begin{cases} 0 & \text{if } P(X) \text{ satisfies constraints} \\ |X| & \text{otherwise} \end{cases} .
$$

In its basic form, since $P(X)$ belongs to a discrete space of set partitions of size super-exponential in $|X|$ and the cost is non-differentiable, optimizing $L(P(X))$ over the partitions of X is in general intractable. However, for tasks with some degree of self-similarity, one can expect that the combinatorial explosion can be avoided. Indeed, if the cost function L is submodular (**bach_submodularity**), i.e.,

$$
L(P(X)) \leq L(P(X_{1,0})) + L(P(X_{1,1}))\ , \text{with} P(X) = P(X_{1,0}) \sqcup P(X_{1,1})\ ,
$$

then the hierarchical splitting from (3.3) can be used as an efficient greedy strategy, since the right hand side acts as a surrogate upper bound that depends only on smaller sets.

In our case, since the split phase is determined by a single block S_θ that is recursively applied, this setup can be cast as a simple fixed-horizon (J steps) Markov Decision Process, that can be trained with standard policy gradient methods; see Section 3.4.2.

3.2.4 Computational Complexity as Regularization

Besides the prospect of better generalization, the recursion (3.1) also enables the notion of computational complexity regularization. Indeed, in tasks that are scale invariant the decomposition in terms of M and S is not unique in general. For example, in the sorting task with n input elements, one may select the largest element of the array and query the sorting task on the remaining $n \quad 1$ elements, but one can also attempt to break the input set into two subsets of similar size using a pivot, and query the sorting on each of the two subsets. Both cases reveal the scale invariance of the problem, but the latter leads to optimal computational complexity ($(n \log n)$) whereas the former does not ((n^2)). Therefore, in a trainable divide-and-conquer architecture, one can regularize the search for split and merge parameters by minimizing computational complexity; see section 3.4.

We describe in the next two sections the neural network design of S and M and our training procedure for the two regimes described above, namely weakly supervised with input-output pairs and with non-differentiable rewards.

3.3 Neural Models for S and M

This section describes the neural network architectures we use as split and merge blocks.

3.3.1 Split

The split block S receives as input a variable-sized set $X = (x_1; : : : ; x_M)$ and produces a binary partition $X = X_0 \; t \; X_1$. We encode such partition with binary labels $z_1 : : : z_M$, $z_m \; 2 \; f0; 1g$. These labels are sampled from probabilities

$$p \; (z_m = 1 \, j \, X) \tag{3.6}$$

that we now describe how to parametrize. Since the model is defined over sets, we use an architecture that certifies that (3.6) are invariant by permutation of the input elements. The *Set2set* model (**set2setvinyals**) constructs a nonlinear set representation by cascading R layers of

$$\begin{aligned}
h_m^{(1)} &= \quad (B_{1;0} x_m + B_{2;0} \; (X)) \\
h_m^{(r+1)} &= \quad \left(B_{1;r} h_m^{(r)} + M \quad {}^1 B_{2;r} \sum_{m^0 \quad M} h_{m^0}^{(r)} \right) \; ; \; m \quad M \; ; r \quad R \; ; \; h_m^{(r)} \; 2 \; \mathbb{R}^d \; ; \\
p \; (z_m = 1 \, j \, X) &= \frac{e^{b^T h_m^{(R)}}}{1 + e^{b^T h_m^{(R)}}}
\end{aligned} \tag{3.7}$$

The parameters of S are thus $\quad = f B_0; B_{1;r}; B_{2;r}; bg$. In order to avoid covariate shifts given by varying input set distributions and sizes, we consider a normalization of the input that standardizes the input variables x_j and feeds the mean and variance $(X) = (\; _0; \quad)$ to the first layer.

Finally, the binary partition tree $P(X)$ is constructed recursively by first computing $p \; (z \, j \, X)$, then sampling from the corresponding distributions to obtain $X = X_0 \, t \, X_1$, and then applying S recursively on X_0 and X_1 until the partition tree leaves have size smaller than a predetermined constant, or the number of scales reaches a maximum value J. We denote the resulting distribution over tree partitions by $P(X) \quad \mathrm{S} \ (X)$.

3.3.2 Merge

Single Merge with Pointer Network

The merge block $M \quad$ takes as input a pair of sequences Y_0, Y_1 and produces an output sequence O. We describe first the architecture for this module, and then comment on how it is modified to perform the finest scale computation $\tilde{M} \quad$.

We modify a Pointer Network (PtrNet) (Vinyals, Fortunato, and Jaitly, 2015) to our input-output interface as our merge block $M \quad$. A PtrNet is an auto-regressive model for tasks where the output sequence is a permutation of a subsequence of the input. This model is an instance of a sequence-to-sequence model with attention, where the attention is used as a pointer to the input instead of being used to compute a weighted sum of the input sequence embeddings.

The model encodes each input sequence $Y_q = (x_{1,q}, \ldots, x_{n_q,q})$, $q = 0, 1$, into a global representation $e_q := e_{q,n_q}$, $q = 0, 1$, by sequentially computing $e_{1,q}, \ldots, e_{n_q,q}$ with an RNN. Then, another RNN decodes the output sequence with initial state $d_0 = \quad (A_0 e_0 + A_1 e_1)$, as described in detail next. The trainable parameters regroup to the RNN encoder and decoder parameters.

Suppose first one has a target sequence $T = (t_1 \ldots t_S)$ for the output of the merge. In that case, we use a conditional autoregressive model of the form

$$\begin{cases} e_{q,i} = f_{\text{enc}}(e_{q,i \ 1}, y_{q,i}) & i = 1, \ldots, n_q \, , q = 0, 1 \, , \\ d_s = f_{\text{dec}}(d_{s \ 1}, t_{s \ 1}) & s = 1, \ldots, S \end{cases} \tag{3.8}$$

The conditional probability of the target given the input is computed by performing attention over the input embeddings $e_{q,i}$ and interpreting the attention as a probability distribution over the input indexes:

$$\begin{cases} u^s_{q,i} = e^T_{q,i} d_s & s = 0, \ldots S \, , q = 0, 1 \, , i \quad n_q \, , \\ p_s = \text{softmax}(u^s) \, . \end{cases} \, , \tag{3.9}$$

leading to

$$P = M \quad (Y_0, Y_1) \, . \tag{3.10}$$

However, since we are interested in weakly supervised tasks, the target output only exists at the coarsest scale of the partition tree. We thus also consider a generative version of the merge block that uses its own predictions in order to sample an output sequence. Indeed, in that case, we replace equation (3.8) by

$$\begin{cases} e_{q,i} = f_{\text{enc}}(e_{q,i \ 1}, y_{q,i}) & i = 1, \ldots, n_q \, , q = 0, 1 \, , \\ d_s = f_{\text{dec}}(d_{s \ 1}, y_{o_{s \ 1}}) & s = 1, \ldots, S \end{cases} \tag{3.11}$$

where o_s is computed as $o_s = x_{\arg\max p_s}$, $s \quad S$, yielding

$$O = M^g(Y_0, Y_1) \, . \tag{3.12}$$

The initial merge operation at the finest scale $\tilde{M}$ is defined as the previous merge module applied to the input $(X_{J,k}, \,)$. We describe next how the successive merge blocks are connected so that the whole system can be evaluated and run.

Recursive Merge over Partition Tree

Given a partition tree $P(X) = fX_{j;k}g_{j;k}$, we perform a merge operation at each node $(j;k)$.

The merge operation traverses the tree in a fine-to-coarse fashion. At the leaves of the tree, the sets $X_{J;k}$ are transformed into $Y_{J;k}$ as $Y_{J;k} = M^{g}(X_{J;k};\ ;)$, and, while $j > 0$, these outputs are recursively transformed along the binary tree as

$$Y_{j;k} = M^{g}(Y_{j+1;2k}; Y_{j+1;2k+1})\ ;\ 0 < j < J\ ; \tag{3.13}$$

using the auto-regressive version, until we reach the scale with available targets:

$$P(Y\ j\,P(X)) = M\ \ (Y_{1;0}; Y_{1;1})\ : \tag{3.14}$$

For simplicity, we write $P(Y jP(X)) = \mathbf{M}\ \ (P(X))$. At test-time, in absence of ground-truth outputs, we replace the last M by its generative version M^{g}.

In practice, during training it is convenient to view the whole merge phase as a structured attention mechanism over the input partition tree, as explained next.

Bootstrapping the Merge Partition Tree

The recursive merge defined by (3.14) can be viewed as a factorized attention mechanism over the input partition. Indeed, the pointer network outputs (3.9) include a stochastic matrix

$$= [p_1; : : : ; p_S]^T\ 2\ \mathbb{R}_+^{S\ \ (n_0+n_1)}\ ; \tag{3.15}$$

where the rows are the p_j probability distributions over the indexes. The number of rows of this matrix is the length of the output sequence and the number of columns is the length of the input sequence. The output O is expressed in terms of by binarizing its entries and multiplying it by the input:

$$_{s;i} = \begin{cases} 1 & \text{if } i = \arg\max_{i^0} p_s(i^0)\ : \\ 0 & \text{otherwise.} \end{cases}\ ;\ O = \begin{pmatrix} Y_0 \\ Y_1 \end{pmatrix}\ : \tag{3.16}$$

Since the merge blocks are cascaded by connecting each others outputs as inputs to the next block, the resulting mapping can be written as

$$p(Y\ j\,P(X)) = \left(\prod_{j=0}^{J} \tilde{\ }_j\right) \begin{pmatrix} Y_{J;0} \\ \vdots \\ Y_{J;n_J} \end{pmatrix}\ ; \text{with} \tag{3.17}$$

$$\tilde{\ }_0 = \ _{0;0}\ ;\ \tilde{\ }_1 = \begin{pmatrix} _{1;0} & 0 \\ 0 & _{1;1} \end{pmatrix} ; : : : ; \tilde{\ }_J = \begin{pmatrix} _{J;0} & 0 & & 0 \\ 0 & _{J;1} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & & 0 & _{J;n_J} \end{pmatrix}\ : \tag{3.18}$$

It results that the recursive merge over the binary tree is a specific reparametrization of PtrNet. The difference now is that the permutation matrix has been decomposed into a product of permutations dictated by the binary tree, indicating our belief that many routing decisions are done locally within the original set. More specifically, the practical implementation of (3.14) will consist in a standard PtrNet in which the permutation matrix (3.15) will be replaced by $\prod_{j=0}^{J} \tilde{\ }_j$. Lastly, since we will train the model with maximum likelihood, in order to avoid singularities we need to enforce that $\log p_{s;t_s}$ is well-defined and therefore that $p_{s;t_s} > 0$. We thus regularize the quantization step (3.16) by replacing $0;1$ with $\;^{1=J};1 \quad n\;^{1=J}$ respectively. We also found that the algorithm converges faster if one starts applying the quantizations only after the first epoch of training.

3.4 Training

This section describes how the model parameters $f \;\; ; \;\; g$ are estimated under two different learning paradigms.

3.4.1 Weak supervision with input-output pairs

Given a training set of pairs $f(X^l;Y^l)g_l \quad _L$, we consider the loss

$$L(\;\; ; \;\;) = \frac{1}{L}\sum_{l \quad L} \mathrm{E}_{P(X) \quad \mathrm{S} \;(X)} \log p \;\; (Y^l \, j \, P(X^l)) \; ; \text{with} \, p \;\; (Y \, j \, P(X)) = \mathrm{M} \;\; (P(X)) \, : \tag{3.19}$$

Section 3.3.2 explained how the merge phase M is akin to a structured attention mechanism. Equations (3.17) and (3.18) show that, thanks to the parameter sharing and despite the quantizations affecting the finest leaves of the tree, the gradient

$$r \quad \log p \; _{;} \;\; (Y \; j \; X) = \mathrm{E}_{P(X) \quad \mathrm{S} \;(X)} \, r \quad \log \mathrm{M} \;\; (P(X)) \tag{3.20}$$

is well-defined and non-zero almost everywhere.

However, since the split parameters are separated from the targets through a series of discrete sampling steps, the same is not true for $r \quad \log p \; _{;} \;\; (Y \; j \; X)$. We therefore resort to the identity used extensively in policy gradient methods. For arbitrary F defined over partitions, and denoting by $f \;\; (X)$ the probability density of the random partition $\mathrm{S} \;\; (X)$, we have

$$\begin{aligned} r \quad \mathrm{E}_{P(X) \quad \mathrm{S} \;(X)} F(P(X)) &= \sum_{X} F(X) \, r \quad f \;\; (X) \\ &= \sum_{X} F(X) f \;\; (X) \, r \quad \log f \;\; (X) \\ &= \mathrm{E}_{X \quad \mathrm{S} \;(X)} F(X) \log f \;\; (X) \\ & \quad \frac{1}{S} \sum_{X^{(s)} \quad \mathrm{S} \;(X)} F(X^{(s)}) \log f \;\; (X^{(s)}) \, : \end{aligned} \tag{3.21}$$

Since the split variables at each node of the tree are conditionally independent given its parent, we can compute $\log f \;\; (P(X))$ as

$$\log f \;\; (P(X)) = \sum_{j=1}^{J} \sum_{k \quad n_j} \sum_{m \quad jX_{j;k}j} \log p \;\; (z_{m;j;k} \, j \, X_j \quad _{1;k=2}) \, :$$

By plugging $F(P(X)) = \log p \;\; (Y \; j \; P(X))$ we thus obtain an efficient estimation of $r \quad \mathrm{E}_{P(X) \quad \mathrm{S} \;(X)} \log p \;\; (Y \; j \; P(X))$. We improve the bias-variance tradeoff of the gradient estimation by substracting a baseline split “policy" that performs random binary partitions over the input set.

3.4.2 Learning from non-differentiable rewards

From Subsection 3.4.1, it is straightforward to train our model in a regime where a given partition $P(X)$ of an input set is evaluated by a black-box system producing a reward $R(P(X))$. Indeed, in that case, the loss becomes

$$L(\ ) = \frac{1}{L}\sum_{l\ \ L} \mathrm{E}_{P(X)\ \ \mathrm{s}\ (X)} R(P(X))\,; \tag{3.22}$$

which can be minimized using (3.21) with $F(P(X)) = R(P(X))$.

Although (3.22) formally allows to train a split model under arbitrary reward functions, the curse of dimensionality will affect the quality of the gradient estimator unless we restrict our attention to structured rewards setting such as the ones described in Subsection 3.2.3; see Section **??** for an illustration of such setup to hierarchical clustering.

3.4.3 Regularization with Computational Complexity

As discussed previously, an added benefit of considering dynamic computation graphs is that one can consider computational complexity as a regularization criteria. We describe how computational complexity can be controlled in both split and merge modules by considering regularization.

Split: We verify from Subsection 3.3.1 that the cost of running each split block S is linear on the input size. It results that the average case complexity $C_S(n)$ of the whole split phase on an input of size n satisfies the following recursion:

$$\mathrm{E}C_S(n) = \mathrm{E}f C_S(\ \ _s n) + C_S((1\ \ \ _s)n) g +\ \ (n)\,; \tag{3.23}$$

where $\ _s$ are the fraction of input elements that are respectively sent to each output. Since this fraction is input-dependent, the average case is obtained by taking expectations with respect to the underlying input distribution. Assuming without loss of generality that $\mathrm{E}(\ _s)\ \ 0{:}5$, we verify that the resulting complexity is of the order of

$$\mathrm{E}C_S(n)\ \ '\ \ \frac{n \log n}{\log \mathrm{E}\ \ _s{}^1}\,; \tag{3.24}$$

which confirms the intuition that balanced partition trees ($\ _s = 0{:}5$) will lead to improved computational complexity. We can enforce $\ _s$ to be close to $0{:}5$ by maximizing the variance of the split probabilities $p\ (z\,j\,X)$ computed by $S\ $ (Eq 3.7):

$$R(S) = \left[M\ \ ^1 \sum_{m\ \ M} p\ (z\,j\,X)^2\ \ \ M\ \ ^2 \left(\sum_m p\ (z\,j\,X) \right)^2 \right] : \tag{3.25}$$

Merge: Each merge block uses an attention mechanism, whose complexity is quadratic in the size of the input to each node. Although the overall complexity of the recursive merge phase can be $o(n^2)$ if on average each merge block reduces the size of its input, the general formulation of (3.17) writes a $n\ \ n$ routing matrix as a product of $J\ n\ \ n$ (block diagonal) matrices. In order to avoid such computational overhead, we can regularize equation (3.9) to enforce merge operations with small number of transpositions. Indeed, for each $q = f0;$ and $s\ \ S$, we denote by $\ _q$ the most recent index of Y_q that has been chosen. We modify the merge output probabilities as

$$
\begin{aligned}
\alpha^s_{q;i} &= e^T_{q;i} d_s + \quad \max(0; \quad ji \quad {}_q j)\;;\; s = 0; : : : S\;; q = 0; 1\;; i \quad n_q\;; \\
p_s &= \mathrm{softmax}(\alpha^s)\;:
\end{aligned}
$$

We thus penalize pointer accesses that deviate further from diagonals away from the identity, since the computational complexity of a model that has no such accesses is $(n \quad)$.

Chapter 4

Experiments

In this section we show results of the Divide and Conquer Network where the atomic dynamic operations are designed as described in 3.3.1 and 3.3.2.

We also test the two different types of training procedures; the weak supervision setting 3.4.1 for the sorting and convex hull problem, and the non-differentiable reward setting 3.4.2 for the k-means.

We provide empirical evidence that introducing recursion to a neural network model outperforms the baseline when the corresponding task has some degree of self-similarity. This recursion is automatically learnt by the model through examples using minimal supervision as described in 3.4. In the case of the convex hull, we also illustrate how the running complexity of the model can also be optimized by using the regularization techniques in 3.4.3.

The model has been implemented using the Pytorch framework (Erickson et al., 2017) in Python and all experiments have been launched in an high performance computing cluster with a total of 4 TitanX GPUs.

4.1 Architecture of atomic blocks and Hyperparameters

No extensive search of hyperparameters has been done in both *split* and *merge* blocks. We have used the same architecture and training hyperparameters for all empirical results, this way the main message of the paper can be better transmitted.

We use 15 hidden units and 5 layers for the set2set split block. The capacity of the corresponding neural network is quite limited, however, the oracle split operations both for sorting and convex hull are relatively simple compared to the merge. The merge block is a PtrNet with a GRU of 128 hidden units both for the encoder and decoder.

The split parameters are updated with the RMSProp algorithm with a learning rate of 0.01 and the merge parameters with SGD with a learning rate of 0.1.

4.2 Sorting

Sorting is a well-known algorithmic task for which there exist various divide and conquer strategies that reduce the algorithm complexity from (n^2) to $(n \log n)$. The most important ones being *mergesort* and *quicksort*. The first one, uses a trivial split operation where the input sequence is split into two subsets independently of the input. The task then is reduced to merge two sorted sequences which can be done in linear time, resulting in an overall complexity of $(n \log n)$. The second one, which achieves the same optimal complexity, uses a trivial merge operation which consists of concatenating both input sequences. The task is then reduced to split the input sequence into two subsets in a way that all elements of one are larger than the

	Baseline	Mergesort	Quicksort	Joint Training
n=8	84%	90%	100%	100%
n=16	30%	67%	100%	100%

TABLE 4.1: We show the accuracy results for the task of sorting with all our architectures of DCNs compared to the baseline. It is surprising to see such a big difference compared to the baseline for small values of length n (where the self-similarity of the task across scales can't be extensively exploited).

ones of the other. This can also done in linear time by finding the median (to ensure equal cardinality) and comparing each to element to it.

These two dynamic algorithms make the sorting task interesting and illustrative to show the versatility of our model to learn split and merge operation with minimal supervision.

The baseline model that we are going to compare our results with is a simple Pointer Network as described in (**vinyals2015pointe**).

We will perform three kind of experiments:

Quicksort. We fix the merge block to be the oracle one for the quicksort, i.e, the sequence must be already sorted at the leaves of the dynamic binary tree. The reward used to train the split block will be the accuracy of the sequence at the leaves. We are in the setting 3.4.2.

Mergesort. We fix the split block to be the oracle one for the mergesort, i.e, the binary tree will be always a balanced tree of depth $\log n$. The merge block must learn how to sort the elements at every node recursively. We are in the setting 3.4.1 but without learning split parameters.

Joint Learning. We learn the split and merge parameters jointly. The reward of the split block is the minus loss of the merge block. This fits in the more general setting 3.4.1.

All models are trained with a dataset created with n samples of real scalar values lying in the unit interval. It is interesting to track the training dynamics of the DCNs in the fully weakly supervised setting (*Joint Training*). We observe three difference phases during training. The first one, the merge is far from convergence and so the split doesn't have access to any information about the training process. In this regime, we observe that the merge parameters start to converge and the model is performing mergesort (i.e, random split and all the sorting is done in the merge phase). However, once this stage has converged, the split block starts receiving more reliable rewards from the merge block and so it will start to split the input in suitable ways. It eventually figures out that the best way to do it is to split the input by large and small values (this way, the sequence will be already ordered at the leaves and it will lower the merge loss). Even though the model has converged to *quicksort*, it is interesting to see that some errors produced by the split block are later corrected by the merge.

4.3 Convex Hull

The planar convex hull task consists of finding the convex hull of a set of points in the plane, i.e, the extremal points of the polytope that contains all the points.

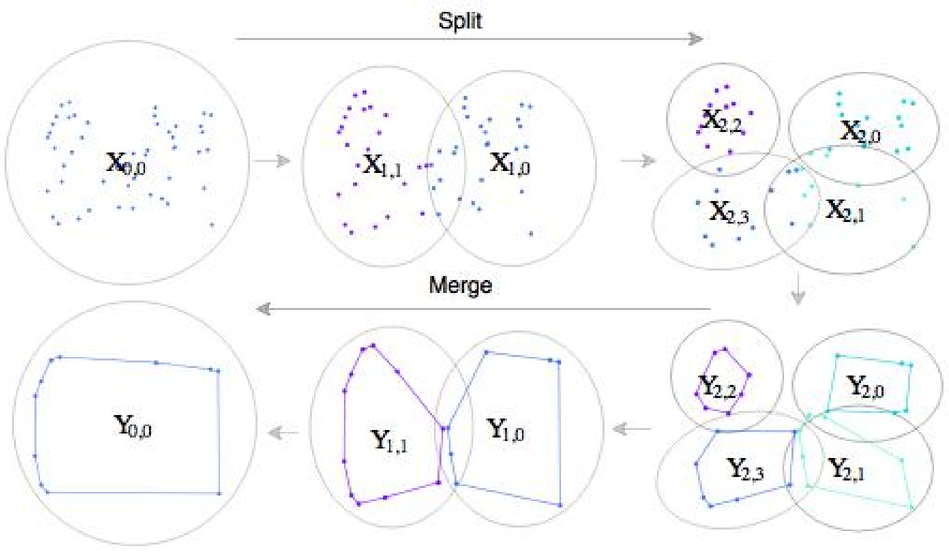


FIGURE 4.1: Representation of the execution trace of the Divide-and-Conquer Network for the Convex Hull problem.

	n=25	n=50	n=100	n=200
Baseline (**vinyals2015pointe**)	75%	69.6%	50.3%	22.1%
DCN	78%	74%	62%	42%

TABLE 4.2: We show the accuracies both of the baseline and de DCN. Both models have been trained for sets of points of cardinality ranging from 8 to 48 and tested at different ranges as can be observed from the table columns. We show how our model easily outperforms the baseline.

The divide and conquer strategy for this task consists of splitting the set of points recursively with hyperplanes $V = fx : h(x) = 0g$ such that the resulting subsets are of equal or similar cardinality $S(X) = fX_{fx:h(x) \quad 0g}; X_{fx:h(x)<0g}g$. This atomic operation can be done in linear time. The merge block computes the union convex hull of both resulting disjoint convex hulls coming from each dynamic branch. This can also be done in linear time in the number of points of both convex hulls. The overall complexity of the recursion is $(n \log n)$. For clearity on the deterministic dynamic convex hull algorithm; see 4.1.

Analogously to the task of sorting, the baseline will be the original pointer network. In this case, however, we use exactly use the model of (**vinyals2015pointe**) as this was one of the three tasks presented in the paper. In order to be comparable to them, we use the same number of parameters as them, in this case 512 hidden units (note that the total number of parameters scales quadratically with the number of hidden units). We train the DCNs in the setting 3.4.1, that is, we do not impose any assumption on the input-output mapping whatsoever and the DCNs must figure out how to factorize the problem recursively to achieve better performance. Both baseline and DCN are trained with sets of points of cardinality between 5 and 50 sampled randomly at the unit square, and we test the model for even larger unseen examples. We show empirically that the network learns this factorization with minimal supervision and outperforms the already powerful baseline; see the results in table 4.2.

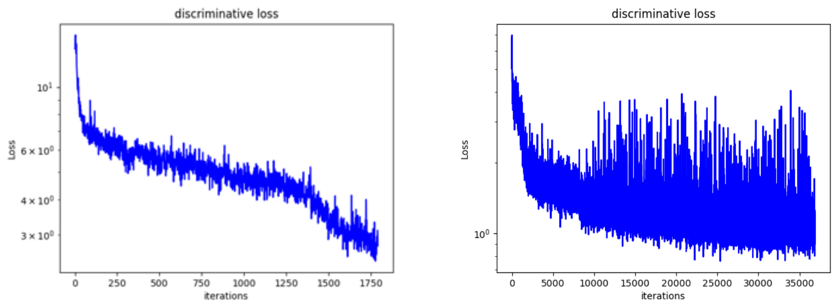


FIGURE 4.2: Tracking of training dynamics at two different iteration scopes. Left: Beginning of training. The cliff appearing at iteration 1300 corresponds to the moment where the split block converges. Right: The other significant change in dynamics occurs at iteration 8192 (the dataset has already been seen once). At that point, we decide to sample the points to connect them between pointer networks instead of modulating with the output matrices at every node 3.3.2. Look that although reaching a lower loss, the sampling adds noise to the problem and unstabilizes it.

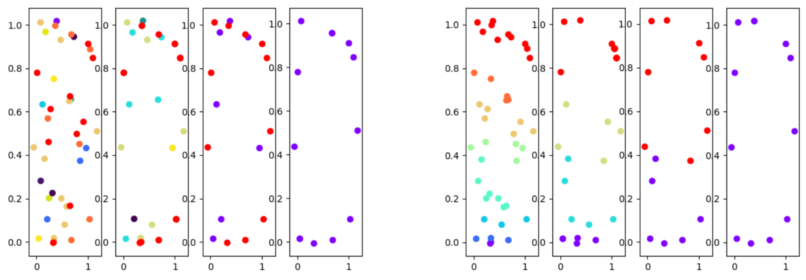


FIGURE 4.3: Examples created by the model. Left: Example with a random split at the beginning of the DCNs training. Observe that the dynamic step operation is not that easy because sometimes it has to alternate between points from one convex hull to the other as can be seen at the last node of the tree. Right: Example with a converged split. You can see how the split block converges to the oracle illustrated in 4.1.

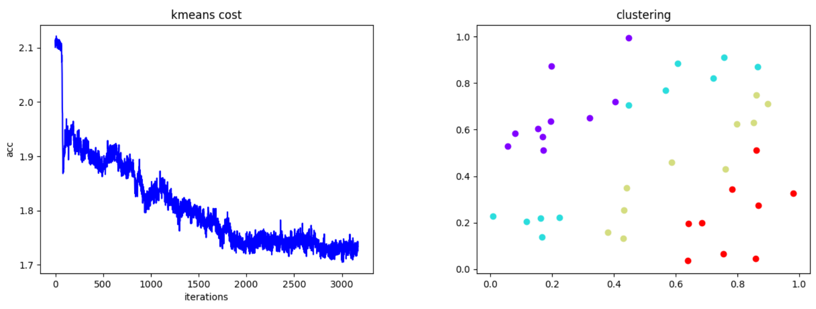


FIGURE 4.4: Left: The cost function $L(P)$ optimized with policy gradient. Right: One example of the DCN at inference time. You can easily observe that the prediction is quite wrong. We relate this problem not with the optimization algorithm but with the lack of capacity of the split block.

4.4 Clustering

The last task we are going to train DCNs on is clustering or k-means. This is a typical task example of the setting with non-differentiable rewards 3.4.2.

K-means is known to be an NP-Hard problem untractable in general. However, many greedy and iterative approximations exist and it is currently still an active line of research.

The problem is the following: Given n points in R^d, find a k-labelling of the points such that the cost

$$L(P) = \sum_{k=1}^{K} \sum_{i2C_k} kx_i \qquad {}_k k^2 \tag{4.1}$$

is minimized. In other words, find the k-clusters on the data such that the sum of the distances of all the points to its corresponding cluster center is minimized.

A lot of existing algorithms that tackle this problem do it in a hierarchical way. The rationale underlying this idea is to try to define coarse-to-fine partitions to the input exploiting some input self-similarity, this is usually called hierarchical clustering. We are going to make DCNs learn it.

The cost function 4.1 is defined on k-partitions of the input. We can just take the non differentiable cost 4.1 and apply the training procedure 3.4.2 with policy gradient.

Experiments are performed by unfolding a binary tree until depth 2 and performing 4-clustering, that is, we get a final partition of the input defined by the clusters at the leaves.

We observe that although our model can be trained consistently with policy gradient, the split block model is way too simple to model a binary clustering and tends to infer unexisting patterns as can be seen in 4.4.

We conjecture that this problem could be solved in future work by increasing the capacity of S .

Chapter 5

Conclusions

The main contribution of this project is the Divide and Conquer Network (DCN) presented in Section 3. We have deeply analyzed its variants and optimization procedures and provided motivating empirical results that show the importance of choosing a prior in machine learning; in our case, the self-similarity across scales that we assume implicitly by sharing the same network parameters across all scales.

We outperform the baselines for the sorting and convex hull and show as in (Cai, Shin, and Song, 2017) that recursion is key to make neural architectures generalize.

Machine learning performance boils down to make learning models generalize on unseen data; in the case of algorithmic problems, unseen inputs with larger dimensionality. We cannot say a task has been successfully learned if it fails for large lengths, and this is still an open problem and an active are of research.

Assuming priors on the data is also key to achieve good results. If no prior is assumed on the input distribution, the problem becomes totally ill-posed (*No Free-Lunch Theorem*), and no reasonably good algorithm can be found.

We think that self-similarity is an obiquitous property and we show that recursion is the natural way to address it.

As future work, we aim at better understanding DCN behavior and applying it successfully to other problems such as shortest path or even NP-hard problems with some degree of self-similarity that can be exploited with greedy strategies like is the case of the famous Travelling Salesman Problem or Graph Coloring.

Bibliography

Abadi, Martín et al. (2016). "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467*.

Andrychowicz, Marcin and Karol Kurach (2016). "Learning Efficient Algorithms with Hierarchical Attentive Memory". In: *arXiv preprint arXiv:1602.03218*.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473*.

Bengio, Yoshua, Ian J Goodfellow, and Aaron Courville (2015). "Deep learning". In: *Nature* 521, pp. 436–444.

Bottou, Léon, Frank E Curtis, and Jorge Nocedal (2016). "Optimization methods for large-scale machine learning". In: *arXiv preprint arXiv:1606.04838*.

Bruna, Joan and Stéphane Mallat (2013). "Invariant scattering convolution networks". In: *IEEE transactions on pattern analysis and machine intelligence* 35.8, pp. 1872–1886.

Cai, Jonathon, Richard Shin, and Song Dawn (2017). "Making Neural Programming Architectures Generalize via Recursion". In: *ICLR*.

Cai, Jonathon, Richard Shin, and Dawn Song (2017). "Making Neural Programming Architectures Generalize via Recursion". In: *arXiv preprint arXiv:1704.06611*.

Chung, Junyoung et al. (2014). "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *arXiv preprint arXiv:1412.3555*.

Cortes, Corinna and Vladimir Vapnik (1995). "Support-vector networks". In: *Machine learning* 20.3, pp. 273–297.

Cybenko, George (1989). "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems (MCSS)* 2.4, pp. 303–314.

Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.Jul, pp. 2121–2159.

Elman, Jeffrey L (1990). "Finding structure in time". In: *Cognitive science* 14.2, pp. 179–211.

Erickson, Bradley J et al. (2017). "Toolkits and Libraries for Deep Learning". In: *Journal of Digital Imaging*, pp. 1–6.

Fanty, Mark (1994). "Context-free parsing in connectionist networks". In: *Parallel natural language processing*, pp. 211–237.

Graves, Alex, Greg Wayne, and Ivo Danihelka (2014). "Neural turing machines". In: *arXiv preprint arXiv:1410.5401*.

Grefenstette, Edward et al. (2015). "Learning to transduce with unbounded memory". In: *Advances in Neural Information Processing Systems*, pp. 1828–1836.

Hardt, Moritz, Benjamin Recht, and Yoram Singer (2015). "Train faster, generalize better: Stability of stochastic gradient descent". In: *arXiv preprint arXiv:1509.01240*.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780.

Hoeffding, W. (1963). "Probability Inequalities for Sums of Bounded Random Variables," in: *Journal of the American Statistical Association* 58, pp. 13–30.

Jordan, Michael I (1997). "Serial order: A parallel distributed processing approach". In: *Advances in psychology* 121, pp. 471–495.

Joulin, Armand and Tomas Mikolov (2015). "Inferring algorithmic patterns with stack-augmented recurrent nets". In: *Advances in Neural Information Processing Systems*, pp. 190–198.

Kaiser, Łukasz and Ilya Sutskever (2015). "Neural gpus learn algorithms". In: *arXiv preprint arXiv:1511.08228*.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*, pp. 1097–1105.

Le Cun, B Boser et al. (1990). "Handwritten Digit Recognition with a Back-Propagation Network". In: *Advances in Neural Information Processing Systems*. Citeseer.

Le Cun, Yann, Leon Bottou, and Yoshua Bengio (1997). "Reading checks with multilayer graph transformer networks". In: *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*. Vol. 1. IEEE, pp. 151–154.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521.7553, pp. 436–444.

LeCun, Yann et al. (1989). "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4, pp. 541–551.

Mallat, Stephane G (1989). "A theory for multiresolution signal decomposition: the wavelet representation". In: *IEEE transactions on pattern analysis and machine intelligence* 11.7, pp. 674–693.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Olivier Bousquet, Stéphane Boucheron and Gábor Lugosi'. "Introduction to Statistical Learning Theory". In: URL: http://84.89.132.1/~lugosi/mlss_slt.pdf.

Pollack, Jordan B (1991). "The induction of dynamical recognizers". In: *Machine Learning* 7.2-3, pp. 227–252.

Reed, Scott and Nando de Freitas (2015). "Neural programmer-interpreters". In: *arXiv preprint arXiv:1511.06279*.

Robbins, Herbert and Sutton Monro (1951). "A stochastic approximation method". In: *The annals of mathematical statistics*, pp. 400–407.

Rosenblatt, Frank (1958). "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6, p. 386.

Russakovsky, Olga et al. (2015). "Imagenet large scale visual recognition challenge". In: *International Journal of Computer Vision* 115.3, pp. 211–252.

Sukhbaatar, Sainbayar et al. (2015). "End-To-End Memory Networks". In: *Advances in Neural Information Processing Systems*, pp. 2431–2439.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*, pp. 3104–3112.

Tabor, Whitney (2000). "Fractal encoding of context-free grammars in connectionist networks". In: *Expert Systems* 17.1, pp. 41–56.

Tieleman, Tijmen and Geoffrey Hinton (2012). "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* 4.2.

Vapnik, Vladimir (1998). *Statistical learning theory. 1998*.

Vapnik, Vladimir N and A Ya Chervonenkis (2015). "On the uniform convergence of relative frequencies of events to their probabilities". In: *Measures of Complexity*. Springer, pp. 11–30.

Vinyals, Oriol, Samy Bengio, and Manjunath Kudlur (2015). "Order matters: Sequence to sequence for sets". In: *arXiv preprint arXiv:1511.06391*.

Vinyals, Oriol, Meire Fortunato, and Navdeep Jaitly (2015). "Pointer networks". In: *Advances in Neural Information Processing Systems*, pp. 2692–2700.

Weston, Jason, Sumit Chopra, and Antoine Bordes (2014). "Memory networks". In: *arXiv preprint arXiv:1410.3916*.

Widrow, Bernard, Marcian E Hoff, et al. (1960). "Adaptive switching circuits". In: *IRE WESCON convention record*. Vol. 4. 1. New York, pp. 96–104.

Zaremba, Wojciech and Ilya Sutskever (2014). "Learning to execute". In: *arXiv preprint arXiv:1410.4615*.

— (2015). "Reinforcement Learning Neural Turing Machines-Revised". In: *arXiv preprint arXiv:1505.00521*.