

---

# THE POON-DOMINGOS PARAMETER LEARNING ALGORITHM FOR IMAGE COMPLETION AND CLASSIFICATION ON SUM-PRODUCT NETWORKS

---

**Renato Lui Geh**  
Computer Science  
Institute of Mathematics and Statistics  
University of São Paulo  
`renatolg@ime.usp.br`

ABSTRACT. In this document we describe the Poon-Domingos [PD11] parameter learning algorithm for image classification and completion.

## 1. STRUCTURE

The Poon-Domingos algorithm uses a fixed structure and then learns the weights through generative learning. We first give an overview on how to build the structure given an image and then provide a pseudo-code algorithm for building such structure. In this document we assume instances as images. However, the Poon-Domingos structure allows for any object with local dependencies.

### 1.1. Overview

The Poon-Domingos structure models a probability distribution over a set of variables with local dependencies. On the plain, one could argue it models rectangular neighborhoods for each point in the space. In the article [PD11], Poon and Domingos use images as a dataset, with dependencies being rectangular pixel neighborhoods. Images are an example of local dependencies, since a pixel has possible dependencies with their neighbors.

Dennis and Ventura explain an intuition of how the Poon-Domingos structure algorithm works [DV12]. We expand on this intuition, giving insights on how such an algorithm is built and showing a pseudo-code visualization of it. Once we have shown how to build the SPN structure, we describe generative learning through gradient descent, and later expectation-maximization.

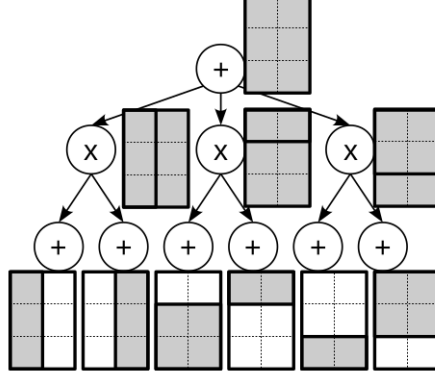


FIGURE 1. The Poon architecture with  $r = 1$  resolution and  $k = 1$  sum nodes per region on a  $2 \times 3$  image. At each  $r$  resolution axis-aligned rectangular decomposition, we create  $k$  sum nodes.[DV12]

## 1.2. Definitions and properties

**Definition 1.1** (Region). *A Region  $\mathcal{R}$  is a rectangular part of an image. Let  $p_0 = (x_0, y_0)$  and  $p_1 = (x_1, y_1)$  be the top-left and bottom-right pixels of  $\mathcal{R}$  relative to the image. These are called the coordinates of  $\mathcal{R}$ .*

**Definition 1.2** (Region Node). *A Region Node  $R$  has a one-to-one and onto mapping with a Region  $\mathcal{R}$ .  $R$  has  $k$  internal nodes associated with it. If  $\mathcal{R}$  is over an  $r \times r$  set of pixels (i.e. the atomic unit), then  $R$  has  $k$  leaf nodes (e.g.  $k$ -mixture of gaussians). Else,  $R$  has  $k$  sum nodes.*

**Definition 1.3** (Decomposition). *Let  $\mathcal{R}$  be a Region. A Decomposition  $\mathcal{D}$  is an axis-aligned partitioning of  $\mathcal{R}$  into two Regions  $\mathcal{R}_1$  and  $\mathcal{R}_2$ .*

The decomposition  $\mathcal{D}$  of a Region  $\mathcal{R}$  involves a few steps. Let  $\mathcal{R}_1$  and  $\mathcal{R}_2$  be the resulting subregions product of the decomposition. The resulting subgraph of the SPN  $S$  of this decomposition is a DAG  $G$ . If  $\mathcal{R}$  is the entire image, then the root of  $G$  is a single sum node and  $G = S$ . Otherwise, then the root of  $G$  is a region node and thus the root of  $G$  is a set of  $k$  sum nodes. Let  $R$  be the root node of  $G$ . Region nodes  $R_1$  and  $R_2$  will both have  $k$  sum nodes (or univariate distributions for leaves). We shall denote as  $R_i^j$  the  $j$ -th sum node of region node  $i$ . For each pairing of sum nodes  $(R_1^i, R_2^j)$ , we create a product node  $\pi$  and add  $R_1^i$  and  $R_2^j$  as children of  $\pi$ . The set  $\Pi$  of these product nodes are the decomposition nodes of  $\mathcal{R}$  into  $\mathcal{R}_1$  and  $\mathcal{R}_2$ . Once we have created all product nodes in this set, we add all of them as children of  $R$ . If  $R$  is a region node, then adding  $\Pi$  as children of  $R$  means, for every sum node  $\sigma$  in  $R$ , set product node  $\pi \in \Pi$  as a child of  $\sigma$ . Each product set  $\Pi$  will have  $\binom{m}{2}$  product nodes, since we take every distinct pairing of sum nodes in  $R_1$  and  $R_2$ .

Since a region node  $R$  is unique, we may have different decompositions in which the same region appears more than once. For this reason we should create a single

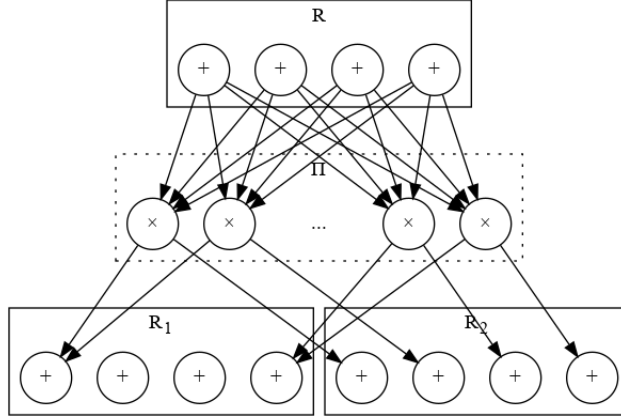


FIGURE 2. A decomposition of a Region  $\mathcal{R}$  into two subregions  $\mathcal{R}_1$  and  $\mathcal{R}_2$ . The set  $\Pi$  of product nodes are the decomposition nodes that connect the unsplit image to the partitions. Each element  $\pi \in \Pi$  connects a pairing of a sum node of  $\mathcal{R}_1$  and of  $\mathcal{R}_2$ .

Region Node for each possible region. We need a map function that takes the top-left and bottom-right pixel positions of a region and maps it to a number for storage. Since every region is unique, we need a one-to-one and onto function.

**Definition 1.4** (Region map function). *A region map function is a function that maps a region into an integer. We define it as*

$$f : \mathbb{Z}_m \times \mathbb{Z}_n \times \mathbb{Z}_m \times \mathbb{Z}_n \rightarrow \mathbb{Z}_{m^2n^2}$$

$$f(x_1, y_1, x_2, y_2) = ((y_1m + x_1)m + x_2)n + y_2$$

where  $x_1, x_2 \in \mathbb{Z}_m$  and  $y_1, y_2 \in \mathbb{Z}_n$ .

**Proposition 1.1.** *The region map function is one-to-one and onto.*

*Proof.* We first prove  $f$  is one-to-one. If  $f$  is injective, then  $f(x_1, y_1, x_2, y_2) = f(x'_1, y'_1, x'_2, y'_2) \Rightarrow (x_1, y_1, x_2, y_2) = (x'_1, y'_1, x'_2, y'_2)$ . Suppose  $f(x_1, y_1, x_2, y_2) = f(x'_1, y'_1, x'_2, y'_2)$  for some  $x_i \in \mathbb{Z}_m$  and  $y_i \in \mathbb{Z}_n$ . Then we have:

$$\begin{aligned} ((y_1m + x_1)m + x_2)n + y_2 &= ((y'_1m + x'_1)m + x'_2)n + y'_2 \\ (m^2y_1 + mx_1 + x_2)n + y_2 &= (m^2y'_1 + mx'_1 + x'_2)n + y'_2 \\ m^2ny_1 + mn x_1 + nx_2 + y_2 &= m^2ny'_1 + mn x'_1 + nx'_2 + y'_2 \\ m^2n(y_1 - y'_1) + mn(x_1 - x'_1) + n(x_2 - x'_2) + (y_2 - y'_2) &= 0 \end{aligned}$$

But  $m, n > 0$ . Therefore, it is easy to see that  $x_i - x'_i = 0$  and  $y_i - y'_i = 0$  is necessary for the equation to hold. Proof of surjection is simple. Since we know  $f$  is one-to-one and that  $\mathbb{Z}_m \times \mathbb{Z}_n \times \mathbb{Z}_m \times \mathbb{Z}_n$  has the same number of elements as  $\mathbb{Z}_{m^2n^2}$ , then it follows that  $f$  must be onto.  $\square$

Bijection of the region map function is necessary since we need the inverse function  $f^{-1}$  to be symmetrical to  $f$ . That is, we must be able to encode a region into

a number and later be able to find what region a number represents. We define the inverse function of  $f$  below.

**Definition 1.5** (Inverse region map function). *The inverse of the region map function is given by the decomposition of an integer  $r \in \mathbb{Z}_{m^2n^2}$  into a tuple  $(x_1, y_1, x_2, y_2) \in \mathbb{Z}_m \times \mathbb{Z}_n \times \mathbb{Z}_m \times \mathbb{Z}_n$ . Let  $g = f^{-1}$ . We define  $g$  as an algorithm as follows*

---

**Algorithm 1** Function  $\text{Encode} := f^{-1} = g$

---

**Input**  $r \in \mathbb{Z}_{m^2n^2}$

**Output**  $(x_1, y_1, x_2, y_2) \in \mathbb{Z}_m \times \mathbb{Z}_n \times \mathbb{Z}_m \times \mathbb{Z}_n$

- 1:  $y_2 \leftarrow r \bmod n$
  - 2: Let  $c \in \mathbb{Z}_{m^2n^2}$
  - 3:  $c \leftarrow \frac{r - y_2}{n}$
  - 4:  $x_2 \leftarrow c \bmod m$
  - 5:  $c \leftarrow \frac{c - x_2}{m}$
  - 6:  $x_1 \leftarrow c \bmod m$
  - 7:  $y_1 \leftarrow \frac{c - x_1}{w}$
  - 8: **return**  $(x_1, y_1, x_2, y_2)$
- 

### 1.3. Structural algorithm

We now show how to construct the Poon-Domingos architecture given an image  $I$  (which is equivalent to a Region consisting of the entire image), resolution  $r$  and  $m$  sum nodes per region.

The algorithm, as summarized in the last subsection, can be constructed recursively. However we avoid this technique in favor of an iterative version, which uses less memory. We must first do a preprocessing of Regions. We iterate over all possible subrectangles in  $I$ , create a Region for each of them and assign an identification number to it. This number is the value of the region map function given the region's position. We name the region map function as **Encode** that takes a region position and returns a non-negative integer. Similarly, we name the inverse function of **Encode** as **Decode** that takes a non-negative integer and returns a region position.

We denote by  $[\mathcal{R}]$ , where  $\mathcal{R}$  is a region, a function that returns a pair of positive integers that represent the width and height of  $\mathcal{R}$ . Once we have constructed all possible regions, we iterate over all possible decompositions according to the following steps:

1. Select each possible region  $\mathcal{R}$  in image  $I$
2. If  $\mathcal{R} = I$ , then  $\mathcal{R}$  is a sum node and is root
3. Else if  $\mathcal{R}$  contains only gaussians, skip this region
4. Else,  $\mathcal{R}$  exists and is a set of sum nodes
5. Partition  $\mathcal{R}$  into a pairing of subregions  $\mathcal{R}_1$  and  $\mathcal{R}_2$
6. For each subregion  $\mathcal{R}_i$ 
  - 6.1 Let  $(w, h) \leftarrow [\mathcal{R}_i]$

- 6.2 If  $w > r$  and  $h > r$  then region node  $R_i$  must be a set of  $m$  gaussians
- 6.3 Else, region node  $R_i$  must be a set of  $m$  sum nodes
- 7. Create a set  $\Pi$  of product nodes
- 8. For each pair  $(R_1^i, R_2^j)$ , where  $R_p^q$  is the inner node  $q$  of subregion node  $R_p$ 
  - 8.1 Create a product node  $\pi$  and add it to set  $\Pi$
  - 8.2 Add  $R_1^i$  and  $R_2^j$  as children of  $\pi$
  - 8.3 Add  $\pi$  as child of all inner sum nodes of  $R$
- 9. Go to step 1.

---

**Algorithm 2** CreateRegions

---

**Input** A pair  $(w, h)$  representing the image  $I$  dimensions

**Input** Dataset  $\mathcal{D}$

**Input** Integer  $m$  as number of components in each Region

**Input** Integer  $r$  as resolution

**Output** A list  $\mathcal{L}$  of Nodes indexed by their region map function value

```

1: Let  $n = w \cdot h$ 
2: for  $i \leftarrow 0..(n-1)$  do
3:    $x_1 \leftarrow i \bmod w$ 
4:    $y_1 \leftarrow i/w$ 
5:   for  $x_2 \leftarrow (w-1)..x_1$  decreasingly do
6:     for  $y_2 \leftarrow (h-1)..y_1$  decreasingly do
7:       if  $x_1 = 0$  and  $y_1 = 0$  and  $x_2 = w-1$  and  $y_2 = h-1$  then
8:          $\triangleright$  The entire image is the root sum node
9:          $j \leftarrow \text{Encode}(x_1, y_1, x_2, y_2)$ 
10:         $\mathcal{L}[j] \leftarrow \text{SumNode}()$ 
11:         $\triangleright \text{SumNode}$  returns a sum node
12:       end if
13:       Let  $R$  be a new Region node
14:       Let  $d_x = x_2 - x_1$  and  $d_y = y_2 - y_1$ 
15:       if  $d_x < r$  or  $d_y < r$  then
16:          $x \leftarrow \max(x_1 + r, x_2)$ 
17:          $y \leftarrow \max(y_1 + r, y_2)$ 
18:          $r \leftarrow \text{Encode}(x_1, y_1, x, y)$ 
19:          $R \leftarrow \mathcal{L}[r]$ 
20:       else if  $d_x = r$  and  $d_y = r$  then
21:          $R \leftarrow \text{GaussianMixture}(x_1, y_1, x_2, y_2, D, m)$ 
22:          $\triangleright \text{GaussianMixture}$  returns a node containing  $m$  gaussians
23:       else
24:          $R \leftarrow \text{RegionNode}(m)$ 
25:          $\triangleright \text{RegionNode}$  returns a node containing  $m$  sum nodes
26:       end if
27:        $j \leftarrow \text{Encode}(x_1, y_1, x_2, y_2)$ 
28:        $\mathcal{L}[j] \leftarrow R$ 
29:     end for
30:   end for
31: end for
32: return  $\mathcal{L}$ 

```

---

The structural algorithm is almost complete. The main object of interest now is on finding all possible regions in an image. Consider the matrix  $M_{n \times m}$  as the image representation of  $I$  indexed by 0. Then we can find all possible regions by iteration over every top-left position  $(x, y)$  and for each of these positions iterating over all bottom-right pixels  $(x + p, y + q)$ ,  $0 \leq p < n, 0 \leq q < m$ .

We assume a dataset  $\mathcal{D}$  where each instance  $\mathcal{D}[X]$  gives an ordered sequence of integers corresponding to the image pixels in expanded form. Let  $R$  be a region,  $(x_1, y_1)$  be its top-left position and  $(x_2, y_2)$  its bottom-right position. We call  $(x_1, y_1, x_2, y_2)$  the coordinates of  $R$ .

**Proposition 1.2.** *Let  $R_1$  be the region in Line 13 in function `CreateRegions`. Line 19 always yields a region  $R_2$  that has already been previously created.*

*Proof.* Let  $(x_1, y_1)$  be  $R_1$ 's top-left position. We fix  $x_2 \geq x_1 + r$ . The third **for** loop in `CreateRegions` yields a decreasing sequence from  $h - 1$  to  $y_1$ . So for each  $y_2$  in this ordered sequence, the previous instance will have already been computed. It then follows that for every  $y_2 = y_1 + r - k$ ,  $0 \leq k \leq r$ , it is true that  $y_1 + r$  has already been computed. We do the same for  $x_2$  by fixing  $y_2 \geq y_1 + r$ . When both  $x_2 < x_1 + r$  and  $y_2 < y_1 + r$ , we fall into the previous subcases.  $\square$

---

### Algorithm 3 Structure

---

**Input** A pair  $(w, h)$  representing the image  $I$  dimensions

**Input** Dataset  $\mathcal{D}$

**Input** Integer  $m$  as number of components in each Region

**Input** Integer  $r$  as resolution

**Output** An SPN  $S$  containing the learned structure from image  $I$

```

1:  $\mathcal{L} \leftarrow \text{CreateRegions}(w, h, \mathcal{D}, m, r)$ 
2: for each key and value pair  $(k, R)$  in  $\mathcal{L}$  do
3:   if  $k = s$  then
4:     skip
5:   end if
6:    $(x_1, y_1, x_2, y_2) \leftarrow \text{Decode}(k)$ 
7:    $\text{LeftQuadrant}(x_1, y_1, x_2, y_2, m, \mathcal{L})$ 
8:    $\text{BottomQuadrant}(x_1, y_1, x_2, y_2, m, \mathcal{L})$ 
9: end for
10:  $s \leftarrow \text{Encode}(0, 0, w - 1, h - 1)$ 
11:  $S \leftarrow \mathcal{L}[s]$ 
12:  $\triangleright S$  is the root node
13: return  $S$ 
```

---

Next, we clarify functions `LeftQuadrant` and `BottomQuadrant`. Recall the algorithm outline we listed earlier. For each decomposition of a region into two subregions, we must create a set of product nodes whose internal products nodes are connected the two distinct subregion internal nodes. Let  $R$  be a region. Our goal is to find the parent regions of  $R$ . In other words, we must find all possible decompositions in which  $R$  is one of the two disjoint subregions. Since we only take

axis-aligned decompositions into account, we can represent all possible decompositions of  $R$  as tightly-bound subrectangles.

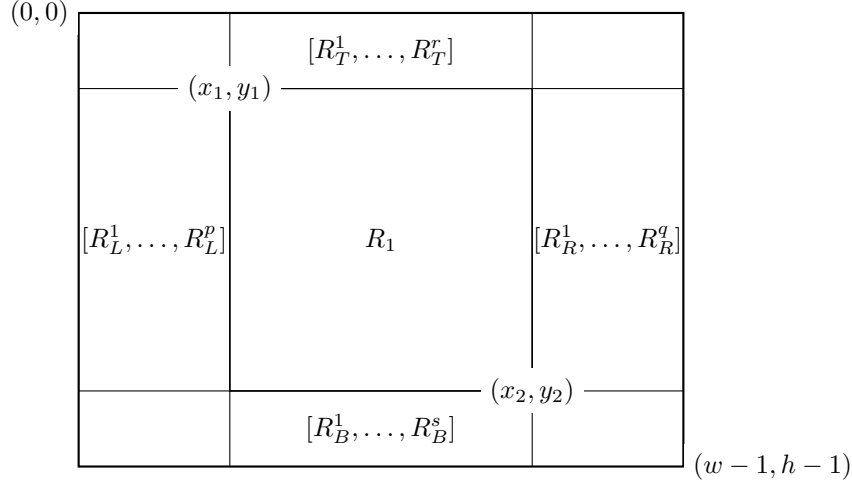


FIGURE 3. Given a decomposition of a region into two subregions, where one of them is  $R_1$ , we can find any possible complementary regions of  $R_1$  by searching each quadrant. The image above shows all possible decompositions containing  $R_1$ , where the set  $[R_Q^1, \dots, R_Q^m]$ , where  $Q$  is a quadrant, contains all complementary regions of  $R_1$  in the direction of  $Q$ . A decomposition is two regions  $R_1$  and  $R_Q^i$ .

To find all the decompositions that contains a region  $R_1$ , we take each possible quadrant (we name them  $\{L, R, T, B\}$  for Left, Right, Top and Bottom) and then take all possible  $R_Q^i$  region, where  $Q$  is the quadrant and  $i$  the  $i$ -th region in  $Q$ . It suffices to find decompositions from  $L$  and  $B$ , as we will eventually do the same for the complementary region of  $R_1$ , which will account for the  $R$  and  $T$  quadrants.

Figure 4 shows a decomposition that contains subregions  $R_1$  and  $R_2$ . Let  $(x_1, y_1, x_2, y_2)$  be the coordinates to  $R_1$ . We have  $x_1$  subregions  $R_2$  that are possible decompositions with  $R_1$ . Similarly for Figure 5, we have  $y_1$  possible subregions  $R_2$ . Consider the left quadrant. If  $R_1$  switches place with  $R_2$ , it becomes clear that we now have the right quadrant. Similarly for the bottom quadrant, by switching  $R_1$  and  $R_2$ , we have the top quadrant. This means our previous assumption that it is enough to take the left and bottom quadrant is correct.

At each pairing of subregions  $R_1$  and  $R_2$  in each quadrant  $Q$ , we must create a set of product nodes  $\Pi$  and connect the parent region  $R = R_1 \cup R_2$  to all the internal product nodes of  $\Pi$ . We now define functions **LeftQuadrant** and **BottomQuadrant**. Both functions are similar, and the only difference is for the left quadrant we iterate horizontally searching for subregions complementary to  $R_1$ . In the bottom quadrant, we iterate vertically searching for the complementaries.

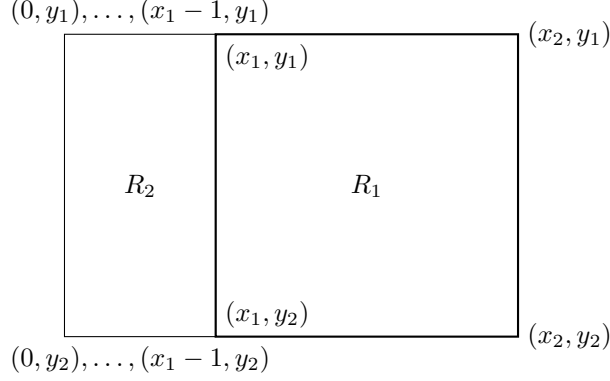


FIGURE 4. The left quadrant. Region  $R_2$  is the complementary region of  $R_1$  and can take any coordinates  $(x', y_1, x_1, y_2)$ , where  $0 \leq x' \leq x_1 - 1$ .

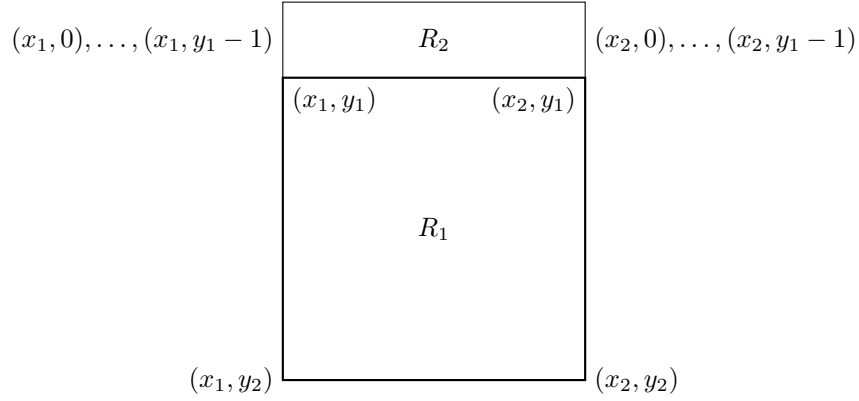


FIGURE 5. The bottom quadrant. Region  $R_2$  is the complementary region of  $R_1$  and can take any coordinates  $(x_1, y', x_2, y_1)$ , where  $0 \leq y' \leq y_1 - 1$ .

In both functions, we are taking region partitions  $R_1$  and  $R_2$ , getting the parent region  $R$ , which is equivalent to getting the union of  $R_1$  and  $R_2$ , and then iterating over all possible 2-combinations of internal nodes of  $R_1$  and  $R_2$ . We then add all product nodes as children of  $R$ 's internal sum nodes. If  $R$  is a single regular sum node, then we have reached the root node, and it is sufficient to simply add the product nodes as children of  $R$ , since  $R$  has no internal nodes.

In this structural algorithm we do not mention edge weights. However we could have initialized weights randomly (in the  $[0, 1]$  range), set all weights to zero or set them to a uniform distribution. We talk about weight learning in more depth in the next section.

We now have most of the structural algorithm formally defined. However, we have not clearly defined two data structures: **GaussianMixture** and **RegionNode**.



---

**Algorithm 4** LeftQuadrant

---

**Input** Coordinates  $(x_1, y_1, x_2, y_2)$  of region  $R_1$ **Input** Integer  $m$  number of components**Input** Node map  $\mathcal{L}$ 

```

1: for  $x \leftarrow 0..(x_1 - 1)$  do
2:    $R_2 \leftarrow \mathcal{L}[\text{Encode}(x, y_1, x_1, y_2)]$   $\triangleright R_2$  is the complementary region of  $R_1$ 
3:    $R \leftarrow \mathcal{L}[\text{Encode}(x, y_1, x_2, y_2)]$   $\triangleright R$  is the parent region (i.e.  $R = R_1 \cup R_2$ )
4:   for  $i \leftarrow 1..m$  do
5:     for  $j \leftarrow 1..m$  do
6:        $\pi \leftarrow \text{ProductNode}()$ 
7:        $\pi.\text{AddChild}(R_1^i), \pi.\text{AddChild}(R_2^j)$ 
8:       if  $R$  is a Region node then
9:         for  $l \leftarrow 1..m$  do
10:           $R^l.\text{AddChild}(\pi)$ 
11:        end for
12:       else  $\triangleright R$  is a sum node and is the entire image
13:          $R.\text{AddChild}(\pi)$ 
14:       end if
15:     end for
16:   end for
17: end for

```

---



---

**Algorithm 5** BottomQuadrant

---

**Input** Coordinates  $(x_1, y_1, x_2, y_2)$  of region  $R_1$ **Input** Integer  $m$  number of components**Input** Node map  $\mathcal{L}$ 

```

1: for  $y \leftarrow 0..(y_1 - 1)$  do
2:    $R_2 \leftarrow \mathcal{L}[\text{Encode}(x_1, y, x_2, y_1)]$   $\triangleright R_2$  is the complementary region of  $R_1$ 
3:    $R \leftarrow \mathcal{L}[\text{Encode}(x_1, y, x_2, y_2)]$   $\triangleright R$  is the parent region (i.e.  $R = R_1 \cup R_2$ )
4:   for  $i \leftarrow 1..m$  do
5:     for  $j \leftarrow 1..m$  do
6:        $\pi \leftarrow \text{ProductNode}()$ 
7:        $\pi.\text{AddChild}(R_1^i), \pi.\text{AddChild}(R_2^j)$ 
8:       if  $R$  is a Region node then
9:         for  $l \leftarrow 1..m$  do
10:           $R^l.\text{AddChild}(\pi)$ 
11:        end for
12:       else  $\triangleright R$  is a sum node and is the entire image
13:          $R.\text{AddChild}(\pi)$ 
14:       end if
15:     end for
16:   end for
17: end for

```

---

The former can be thought of as a sum node with  $m$  gaussians, and the latter simply a set of  $m$  sum nodes. Let  $G$  be an  $m$ -gaussian node. We assume that  $G$  is

a sum node with  $m$  gaussians as leaves. Additionally, the value of  $G$  is the value of the sum node, and  $G$  itself will be, for simplicity purposes, considered a leaf itself.

## 2. PARAMETER LEARNING

The [PD11] article mentions two ways of weight updating, gradient descent (GD) through backpropagation and expectation-maximization (EM). Both suffer from the gradient diffusion problem, that is, for deep structures, the signal fades away as we go down layers, reaching zero in a few steps. This is countered by using a hard version based on the MAP states instead of marginal inference. For each weight update algorithm, we first show the soft version using marginal inference, and then show the hard version using MAP inference. Both algorithms can be applied the same way in the master learning algorithm from [PD11] shown below.

---

### Algorithm 6 LearnSPN

---

**Input** A pair  $(w, h)$  representing the image  $I$  dimensions  
**Input** Dataset  $\mathcal{D}$   
**Input** Integer  $m$  as number of components in each Region  
**Input** Integer  $r$  as resolution  
**Output** An SPN with learned weights  
1:  $S \leftarrow \text{Structure}(w, h, \mathcal{D}, m, r)$   
2:  $\text{InitializeWeights}(S)$   
3: **repeat**  
4:    $\text{UpdateWeights}(S, \mathcal{D})$   
5: **until** convergence  
6:  $S \leftarrow \text{PruneZeroWeights}(S)$   
7: **return**  $S$

---

Weight initialization, as mentioned in the previous section, can be done either through randomization in the unit interval, a uniformed distribution or all weights set to zero.

Pruning zero weights is done by recursively removing edges that have zero weights. Then, while there exists non-root parentless nodes, remove each of them.

---

### Algorithm 7 PruneZeroWeights

---

**Input** An SPN  $S$   
**Output** The resulting SPN after pruning  
1: Let  $Q$  be a queue data structure  
2: Let  $T$  be a stack data structure  
3: Let  $V$  be a list of booleans, where  $V[i]$  is true if node  $i$  has already been visited  
4: Let  $L$  be an adjacency list where  $L[i]$  contains a list of nodes  
5:  $Q.\text{Enqueue}(S), T.\text{Push}(S)$   
6:  $V[S] \leftarrow \text{true}, L[S] \leftarrow []$

---

---

```

7: for each child  $c$  of  $S$  do
8:    $L[c].\text{Add}(S)$ 
9: end for
10: while not  $Q.\text{Empty}()$  do
11:    $s \leftarrow Q.\text{Dequeue}()$ 
12:   for each child  $c$  of  $s$  do
13:      $L[c].\text{Add}(s)$ 
14:     if not  $V[c]$  then
15:        $Q.\text{Enqueue}(c), T.\text{Push}(c)$ 
16:        $V[c] \leftarrow \text{true}$ 
17:     end if
18:   end for
19: end while
20: Let  $M$  be a map where  $M[i]$  is true if node  $i$  is marked for deletion.
21: while not  $T.\text{Empty}()$  do
22:    $s \leftarrow T.\text{Dequeue}()$ 
23:   if  $s$  is a sum node then
24:     for each edge  $e_{s,i}$  in  $s$  do
25:       if  $\text{Val}(e_{s,i}) = 0$  then
26:         Delete edge  $e_{s,i}$ 
27:          $L[i].\text{Remove}(s)$ 
28:          $\text{RecursivelyDeleteOrphans}(i, L, M)$ 
29:       end if
30:     end for
31:   end if
32: end while
33: for each key value pair  $(s, e)$  in  $M$  do
34:   if  $e = \text{true}$  then
35:     Delete node  $s$  from  $S$ 
36:   end if
37: end for
38: return  $S$ 

```

---

We denote  $e_{i,j}$  as an edge that goes from node  $i$  to node  $j$ . We denote  $\text{Val}(e_{i,j})$  as the weight value of edge  $e_{i,j}$ . In Algorithm 7, we traverse the graph of SPN  $S$  in a bottom-up fashion. We run through the graph this particular way to avoid recomputations. By going bottom-up, at every zero weight edge of a sum node  $s$  we encounter, we can guarantee that there are no zero edges on the descendants of  $s$ . In function `PruneZeroWeights`, we first store a copy of the graph through an adjacency list  $L$ . List  $L$  stores the parents of a node  $i$ . That is, for every entry  $L[i]$ , where  $i$  is a node,  $L[i]$  is a list of parent nodes of  $i$ . We only need to account for sum nodes, as they are the only type of node who have weights on their outgoing edges. Once we start iterating over every sum node  $s$ , we check each outgoing edge of  $s$ . If we find a zero weight edge, we first delete such edge from  $S$ , remove  $s$  as a parent from the copied graph adjacency list  $L$  and run function `RecursivelyDeleteOrphans`. This function will delete every descendant of  $s$  that has no parent nodes. Since we traverse the graph bottom-up, we guarantee that at every call of `RecursivelyDeleteOrphans`, there exists no zero weight edge in

the graph scope of the function. Therefore, it suffices to recursively delete orphan nodes.

Algorithm 8 simply performs a breadth-first search on the given sub-SPN, searching for any orphan nodes according to list  $L$ . When it finds such a node  $s$ , it marks  $s$  for deletion and removes  $s$  as parent of its children. Then, it does the same for all children of  $s$ .

---

**Algorithm 8** RecursivelyDeleteOrphans

---

**Input** An SPN  $S$   
**Input**  $L$  adjacency list from PruneZeroWeights  
**Input**  $M$  map to mark node deletion

```

1: Let  $Q$  be a queue data structure
2:  $Q.\text{Enqueue}(S)$ 
3: while not  $Q.\text{Empty}()$  do
4:    $s \leftarrow Q.\text{Dequeue}()$ 
5:   if  $L[s].\text{Empty}()$  then
6:      $M[s] = \text{true}$ 
7:     for each child  $c$  of  $s$  do
8:        $L[c].\text{Remove}(s)$ 
9:     end for
10:    for each child  $c$  of  $s$  do
11:       $Q.\text{Enqueue}(c)$ 
12:    end for
13:   end if
14: end while
```

---

Function `UpdateWeights` is what determines whether the algorithm uses gradient descent or expectation-maximization. In the next subsections we detail on how to perform generative parameter learning using the two techniques.

### 2.1. Gradient descent

### 2.2. Expectation-maximization

## REFERENCES

- [DV12] Aaron Dennis and Dan Ventura. “Learning the Architecture of Sum-Product Networks Using Clustering on Variables”. In: *Advances in Neural Information Processing Systems* 25 (2012).
- [PD11] Hoifung Poon and Pedro Domingos. “Sum-Product Networks: A New Deep Architecture”. In: *Uncertainty in Artificial Intelligence* 27 (2011).