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.

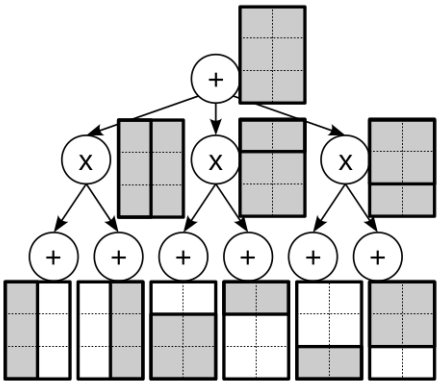

FIGURE 1. The Poon architecture with $r = 1$ resolution and $k = 1$ sum nodes per region on a 2×3 image. At each r resolution axis-aligned rectangular decomposition, we create k sum nodes.[DV12]

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 π and add R_1^i and R_2^j as children of π. The set Π 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 Π as children of R means, for every sum node σ in R, set product node $\pi \in \Pi$ as a child of σ. Each product set Π 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

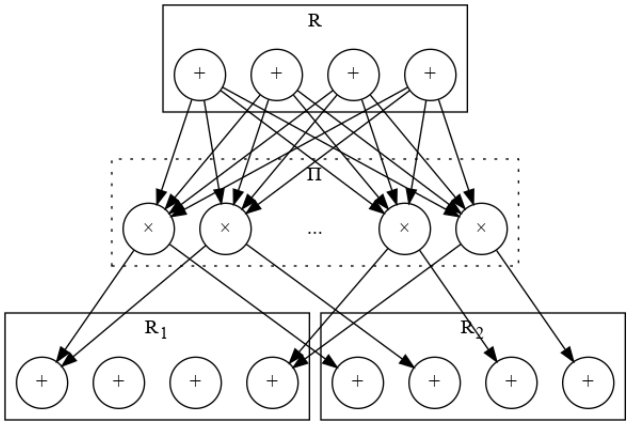

FIGURE 2. A decomposition of a Region $\mathcal{R}$ into two subregions $\mathcal{R}_1$ and $\mathcal{R}_2$. The set Π of product nodes are the decomposition nodes that connect the unsplit image to the partitions. Each element $\pi \in \Pi$ connects a pairing of a sum node of $\mathcal{R}_1$ and of $\mathcal{R}_2$.

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*

$$f : \mathbb{Z}_m \times \mathbb{Z}_n \times \mathbb{Z}_m \times \mathbb{Z}_n \to \mathbb{Z}_{m^2n^2}$$
$$f(x_1, y_1, x_2, y_2) = ((y_1 m + x_1)m + x_2)n + y_2$$

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:

$$((y_1 m + x_1)m + x_2)n + y_2 = ((y_1' m + x_1')m + x_2')n + y_2'$$
$$(m^2 y_1 + m x_1 + x_2)n + y_2 = (m^2 y_1' + m x_1' + x_2')n + y_2'$$
$$m^2 n y_1 + mn x_1 + n x_2 + y_2 = m^2 n y_1' + mn x_1' + n x_2' + y_2'$$
$$m^2 n(y_1 - y_1') + mn(x_1 - x_1') + n(x_2 - x_2') + (y_2 - y_2') = 0$$

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 $\mathbb{Z}_{m^2n^2}$, than it follows that f must be onto. □

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 $(x_1, y_1, x_2, y_2) \in \mathbb{Z}_m \times \mathbb{Z}_n \times \mathbb{Z}_m\mathbb{Z}_n$. Let $g = f^{-1}$. We define g as an algorithm as follows*

Algorithm 1 Function `Encode` $\coloneqq f^{-1} = g$

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: $y_2 \leftarrow i \mod n$
2: Let $c \in \mathbb{Z}_{m^2n^2}$
3: $c \leftarrow \frac{(r-y_2)}{n}$
4: $x_2 \leftarrow c \mod m$
5: $c \leftarrow \frac{c-x_2}{m}$
6: $x_1 \leftarrow c \mod m$
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 R is a sum node and is root
3. Else if R contains only gaussians, skip this region
4. Else, 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 Π 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 π and add it to set Π
8.2. Add R_1^i and R_2^j as children of π
8.3. Add π 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

Let $n = w \cdot h$
for $i \leftarrow 0..(n-1)$ **do**
    $x_1 \leftarrow i \mod w$
    $y_1 \leftarrow i/w$
    for $x_2 \leftarrow (w-1)..x_1$ decreasingly **do**
        for $y_2 \leftarrow (h-1)..y_1$ decreasingly **do**
            if $x_1 = 0$ and $y_1 = 0$ and $x_2 = w-1$ and $y_2 = h-1$ **then**
                ▷ The entire image is the root sum node
                $j \leftarrow$`Encode`(x_1, y_1, x_2, y_2)
                $\mathcal{L}[j] \leftarrow$`SumNode`()
                ▷ `SumNode` returns a sum node
            end if
            Let R be a new Region node
            Let $d_x = x_2 - x_1$ and $d_y = y_2 - y_1$
            if $d_x < r$ or $d_y < r$ **then**
                $x \leftarrow \max(x_1 + r, x_2)$
                $y \leftarrow \max(y_1 + r, y_2)$
                $r \leftarrow$`Encode`(x_1, y_1, x, y)
                $R \leftarrow \mathcal{L}[r]$
            else if $d_x = r$ and $d_y = r$ **then**
                $R \leftarrow$`GaussianMixture`$(x_1, y_1, x_2, y_2, D, m)$
                ▷ `GaussianMixture` returns a node containing m gaussians
            else
                $R \leftarrow$`RegionNode`(m)
                ▷ `RegionNode` returns a node containing m sum nodes
            end if
            $j \leftarrow$`Encode`(x_1, y_1, x_2, y_2)
            $\mathcal{L}[j] \leftarrow R$
        end for
    end for
end for
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. □

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: $\mathcal{L} \leftarrow$`CreateRegions`(w, h, 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: $(x_1, y_1, x_2, y_2) \leftarrow$ `Decode`(k)
7: `LeftQuadrant`$(x_1, y_1, x_2, y_2, m, \mathcal{L})$
8: `BottomQuadrant`$(x_1, y_1, x_2, y_2, m, \mathcal{L})$
9: **end for**
10: $s \leftarrow$`Encode`$(0, 0, w - 1, h - 1)$
11: $S \leftarrow \mathcal{L}[s]$
12: ▷ 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.

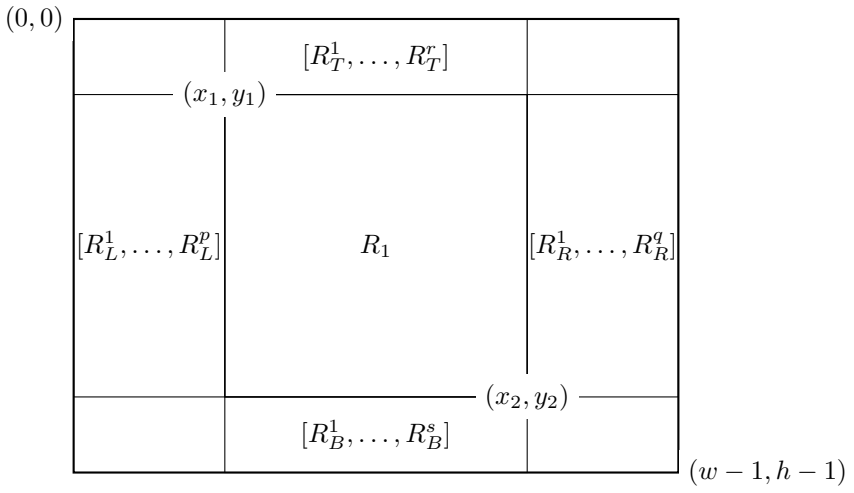

FIGURE 3. Given a decomposition of a region into two subregions, where one of them is R_1, we can find any possible complementary regions of R_1 by searching each quadrant. The image above shows all possible decompositions containing R_1, where the set $[R_Q^1, \ldots, R_Q^m]$, where Q is a quadrant, contains all complementary regions of R_1 in the direction of Q. A decompositon 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 Π and connect the parent region $R = R_1 \cup R_2$ to all the internal product nodes of Π. 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.

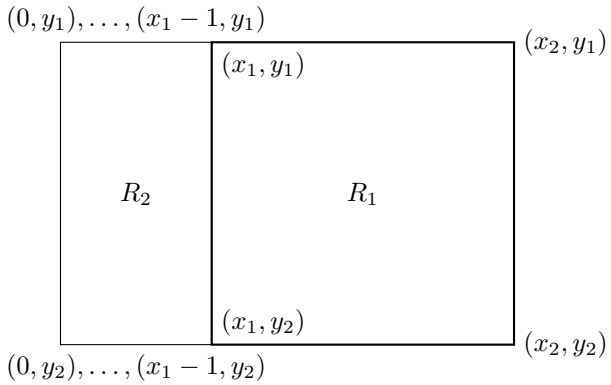

FIGURE 4. The left quadrant. Region R_2 is the complementary region of R_1 and can take any coordinates (x', y_1, x_1, y_2), where $0 \leq x' \leq x_1 - 1$.

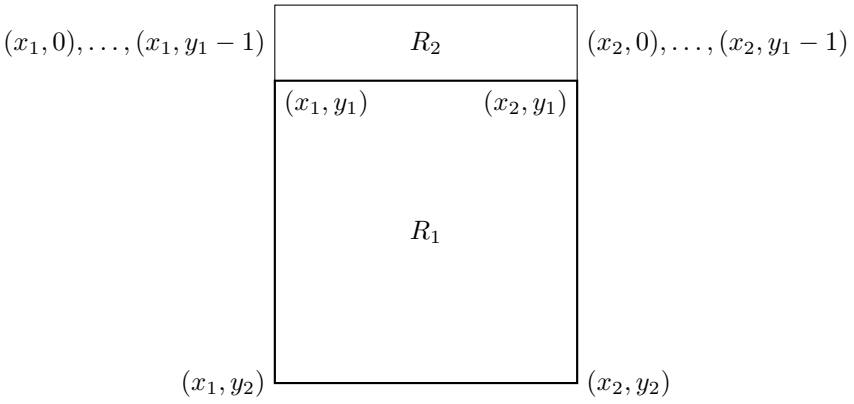

FIGURE 5. The bottom quadrant. Region R_2 is the complementary region of R_1 and can take any coordinates (x_1, y', x_2, y_1), where $0 \leq y' \leq y_1 - 1$.

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 $\mathcal{L}$

for $x \leftarrow 0..(x_1 - 1)$ **do**
  $R_2 \leftarrow \mathcal{L}[\texttt{Encode}(x, y_1, x_1, y_2)]$ ▷ R_2 is the complementary region of R_1
  $R \leftarrow \mathcal{L}[\texttt{Encode}(x, y_1, x_2, y_2)]$ ▷ R is the parent region (i.e. $R = R_1 \cup R_2$)
  for $i \leftarrow 1..m$ **do**
    for $j \leftarrow 1..m$ **do**
      $\pi \leftarrow \texttt{ProductNode}()$
      $\pi.\texttt{AddChild}(R_1^i)$, $\pi.\texttt{AddChild}(R_2^j)$
      if R is a Region node **then**
        for $l \leftarrow 1..m$ **do**
          $R^l.\texttt{AddChild}(\pi)$
        end for
      else ▷ R is a sum node and is the entire image
        $R.\texttt{AddChild}(\pi)$
      end if
    end for
  end for
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 $\mathcal{L}$

for $y \leftarrow 0..(y_1 - 1)$ **do**
  $R_2 \leftarrow \mathcal{L}[\texttt{Encode}(x_1, y, x_2, y_1)]$ ▷ R_2 is the complementary region of R_1
  $R \leftarrow \mathcal{L}[\texttt{Encode}(x_1, y, x_2, y_2)]$ ▷ R is the parent region (i.e. $R = R_1 \cup R_2$)
  for $i \leftarrow 1..m$ **do**
    for $j \leftarrow 1..m$ **do**
      $\pi \leftarrow \texttt{ProductNode}()$
      $\pi.\texttt{AddChild}(R_1^i)$, $\pi.\texttt{AddChild}(R_2^j)$
      if R is a Region node **then**
        for $l \leftarrow 1..m$ **do**
          $R^l.\texttt{AddChild}(\pi)$
        end for
      else ▷ R is a sum node and is the entire image
        $R.\texttt{AddChild}(\pi)$
      end if
    end for
  end for
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. The structural algorithms can be found at [Geh16], under the file `learn/poon.go`.

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: $S \leftarrow$ `Structure`$(w, h, \mathcal{D}, m, r)$
2: `InitializeWeights`(S)
3: **repeat**
4: `UpdateWeights`$(S, \mathcal{D})$
5: **until** convergence
6: $S \leftarrow$ `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`.Enqueue`(S), T`.Push`(S)
6: $V[S] \leftarrow$ **true**, $L[S] \leftarrow [\,]$

```
7: for each child c of S do
8:     L[c].Add(S)
9: end for
10: while not Q.Empty() do
11:     s ← Q.Dequeue()
12:     for each child c of s do
13:         L[c].Add(s)
14:         if not V[c] then
15:             Q.Enqueue(c), T.Push(c)
16:             V[c] ← 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.Empty() do
22:     s ← T.Dequeue()
23:     if s is a sum node then
24:         for each edge e_{s,i} in s do
25:             if w_{s,i} = 0 then
26:                 Delete edge e_{s,i}
27:                 L[i].Remove(s)
28:                 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 = 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 $w_{i,j}$ 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.Enqueue(S)
3: while not Q.Empty() do
4:     s ← Q.Dequeue()
5:     if L[s].Empty() then
6:         M[s] = true
7:         for each child c of s do
8:             L[c].Remove(s)
9:         end for
10:        for each child c of s do
11:            Q.Enqueue(c)
12:        end for
13:    end if
14: end while
```

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

2.1. Gradient descent

Computing the gradient descent requires first finding the derivatives of each node in the SPN. We show the derivatives as they appear in [PD11]. Note that these definitions assume a certain particular SPN structure. Namely that the SPN is composed of alternating layers of sum and product nodes.

We denote by $\mathrm{Pa}(i)$ the set of parents of a node i and by $\mathrm{Ch}(i)$ the set of children. Additionally, we use the following notation $T_{-i} \coloneqq T \setminus \{i\}$, where T is a set and i an element of T.

Definition 2.1 (Sum node derivative)**.** *Let S be an SPN and S_i a non-root sum node in S. By assumption, all parents of S_i are product nodes. The derivative of S wrt S_i is given by*

$$\frac{\partial S}{\partial S_i}(x) = \sum_{S_k \in Pa(S_i)} \left(\frac{\partial S}{\partial S_k}(x) \right) \prod_{S_l \in Ch_{-S_i}(S_k)} S_l(x).$$

Definition 2.2 (Product node derivative)**.** *Let S be an SPN and S_i a non-root product node in S. By assumption, all parents of S_i are sum nodes. The derivative of S wrt S_i is given by*

$$\frac{\partial S}{\partial S_i}(x) = \sum_{S_k \in Pa(S_i)} w_{S_k,S_i} \frac{\partial S}{\partial S_k}(x)$$

As we have mentioned earlier, Poon and Domingos assume that all parents of products are sums, and all parents of sums are products. We show next that the differentiation results do not change when a sum has an arbitrary number of sum nodes as parents, or when a product has products as parents.

To prove this result, we must first show that, given an arbitrary structure (i.e. a node can have any type of node as parent), we can modify such structure to fit the alternating layers structure.

Lemma 2.1. *Let S be an SPN and S_i a non-root sum node of S where n parents of S_i are sum nodes and m parents of S_i are product nodes. The derivative of S wrt S_i is given by Definition 2.1.*

Proof. From the hypothesis we have that the sub-SPN composed solely of S_i and its parents is given by Figure 6. We can modify such a structure with no change in

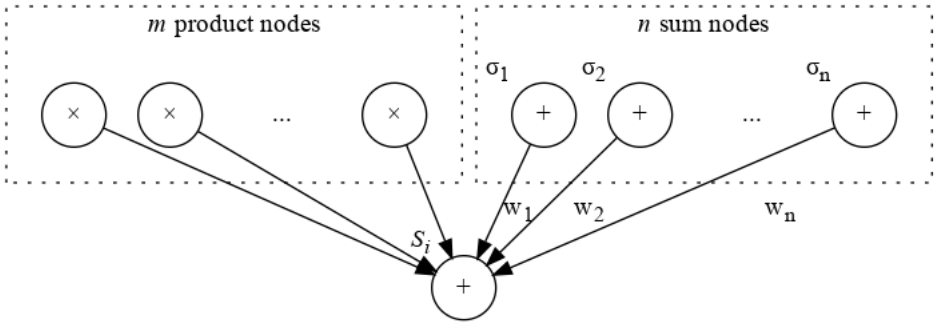

FIGURE 6. S_i has m product parents and n sum parents. It has an arbitrary structure because of sum parents.

the value of S by adding a single product node as child of each sum parent and then connecting it to S_i, as shown in Figure 7. S has no change in value because every node σ_j has no change in its value, as we show next. The value of σ_j in Figure 6 is

$$\sigma_j(x) = \sum_{k \in \mathrm{Ch}(\sigma_j)} w_{\sigma_j,k} k(x) = w_j S_i + \sum_{k \in \mathrm{Ch}_{-S_i}(\sigma_j)} w_{\sigma_j,k} k(x).$$

The value of σ_j in Figure 7, which we shall rename to σ'_j is given by the expression

$$\sigma'_j(x) = \sum_{k \in \mathrm{Ch}(\sigma'_j)} w_{\sigma'_j,k} k(x) = w_j \pi_j(x) + \sum_{k \in \mathrm{Ch}_{-\pi_j}(\sigma'_j)} w_{\sigma'_j,k} k(x).$$

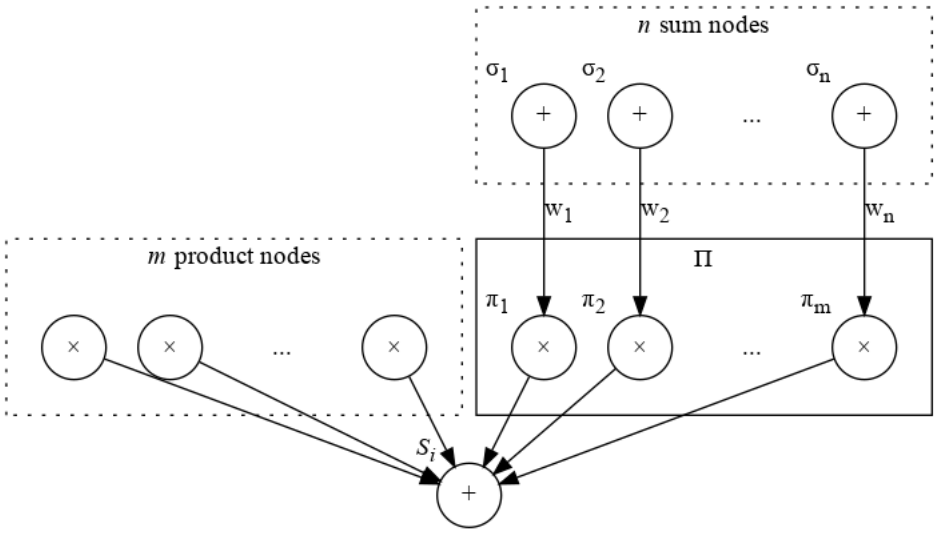

FIGURE 7. We add the set Π of product nodes between S_i and $\sigma_1, \ldots, \sigma_n$. This does not change the value of the SPN.

But $\pi_j(x) = S_i(x)$, therefore:

$$\sigma'_j(x) = w_j S_i(x) + \sum_{k \in \mathrm{Ch}_{-\pi_j}(\sigma'_j)} w_{\sigma'_j,k} k(x) = w_j S_i(x) + \sum_{k \in \mathrm{Ch}_{-S_i}(\sigma_j)} w_{\sigma_j,k} k(x) = \sigma_j(x)$$

We now need to verify that the derivatives do not change either. We know that for all S_i parents, the differentials do not change, since they take their values from their parents. So it suffices to show only that the differential at S_i does not change. After applying the structure transformation from Figure 7, we now have all parents of S_i as products. From that we have that the derivative is given by

$$\begin{aligned} \frac{\partial S}{\partial S_i}(x) &= \sum_{S_k \in \mathrm{Pa}(S_i)} \frac{\partial S}{\partial S_k}(x) \prod_{S_l \in \mathrm{Ch}_{-S_i}(S_k)} S_l(x) \\ &= \sum_{j=1}^{n} \frac{\partial S}{\partial \pi_j}(x) \prod_{S_l \in \mathrm{Ch}_{-S_i}(\pi_j)} S_l(x) + \sum_{j=1}^{n} \frac{\partial S}{\partial Z_j}(x) \prod_{S_l \in \mathrm{Ch}_{-S_i}(Z_j)} S_l(x) \end{aligned} \tag{2.1}$$

Where we named nodes Z_j as the product nodes of S_i that are not in the Π set. Our claim is that, for each π_i, its differential is the same as σ_i. If we show this claim to be true, than $\partial S/\partial S_i(x)$ must be the same as the one in the original structure. From the definition, $\partial S/\partial \pi_i(x) = \partial S/\partial \sigma_i(x)$. Applying this on Equation 2.1, we have

$$\frac{\partial S}{\partial S_i}(x) = \sum_{j=1}^{n} \frac{\partial S}{\partial \sigma_j}(x) \prod_{S_l \in \mathrm{Ch}_{-S_i}(\pi_j)} S_l(x) + \sum_{j=1}^{n} \frac{\partial S}{\partial Z_j}(x) \prod_{S_l \in \mathrm{Ch}_{-S_i}(Z_j)} S_l(x)$$

Which is exactly the derivative of the original SPN without the structure modification. □

Lemma 2.2. *Let S be an SPN and S_i a non-root product node of S where n parents of S_i are sum nodes and m parents of S_i are product nodes. The derivative of S wrt S_i is given by Definition 2.2.*

Proof. The proof goes exactly like the previous proof, but on with the structure alterations depicted in Figure 9. Figure 8 shows the structure before the transformation. Similar to sum case, it is easy to prove that the SPN has no change in

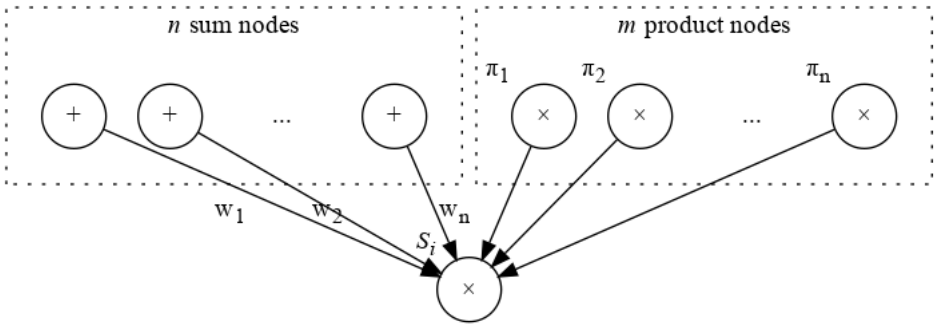

FIGURE 8. Node S_i and its parents.

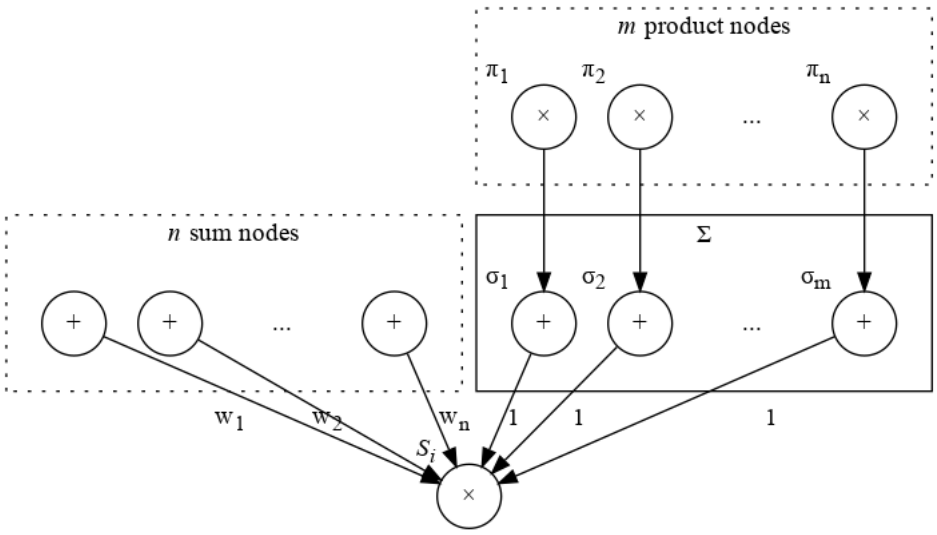

FIGURE 9. The above SPN with the modified structure.

value and that the derivative of each parent of S_i remains the same. From that, we then apply the definition of derivative of a product node and show that it does not change with the applied transformation to the SPN. □

Theorem 2.1. *The derivatives given in Definition 2.1 and Definition 2.2 hold for any SPN structure.*

Proof. Let S be an SPN. For each node S_i in S, if $\mathrm{Pa}(S_i)$ are all of different types than S_i, the claim is trivially true. Else, if S_i is a sum node, we know from Lemma 2.1 that the derivative is equal to Definition 2.1; and if S_i is a product, from Lemma 2.2 we have that the derivative is the same as Definition 2.2. □

Definition 2.3 (Weight derivative)**.** *Let S be an SPN and S_i a sum node of S. Let $W = \{w_{i,1}, w_{i,2}, \ldots, w_{i,n}\}$ be the set of edge weights, where $w_{i,j}$ is the weight of an edge going from node S_i to S_j. The weight derivative of S wrt $w_{i,j}$ is given by*

$$\frac{\partial S}{\partial w_{i,j}}(x) = \frac{\partial S}{\partial S_i}(x) S_j(x)$$

The weight derivative $\partial S/\partial w_{i,j}$ computes the gradient descent variation on weight $w_{i,j}$. We wish to compute the likelihood gradient and optimize (i.e. maximize) it with gradient descent, meaning we wish to "follow" the gradient direction:

$$\Delta w_{i,j} = \eta \frac{\partial S}{\partial w_{i,j}}(x) = \eta \left(\frac{\partial S}{\partial S_i}(x) S_j(x) \right) \tag{2.2}$$

Equation 2.2 shows the gradient update on a weight $w_{i,j}$, where η is the learning rate. The weight derivative depends on the derivative of the node in question, which in turn depends on the derivative of its parents. It also needs the marginal values of each of the node's children as well.

A way to compute the derivatives of each node is shown in [GD12]. We follow this backpropagation algorithm in this document. Instead of computing the derivatives of a node S_i by finding the necessary values from its parents first and then computing $\partial S/\partial S_i$, in a clear bottom-up approach, we perform a top-down computation on the SPN. At each node S_i, we add the element of the sum of derivatives concerning S_i to the variable that stores the derivative of the $\partial S/\partial S_j$, where $S_j \in \mathrm{Ch}(S_i)$. That is, we follow the following algorithm:

1. Select an unvisited node S_i in SPN S:
2. For each child S_j of S_i:
   - 2.1. If S_i is a sum node:
     - 2.1.1. $\frac{\partial S}{\partial S_j} \leftarrow \frac{\partial S}{\partial S_j} + w_{i,j} \frac{\partial S}{\partial S_i}(x)$
     - 2.1.2. $\frac{\partial S}{\partial w_{i,j}} \leftarrow S_j(x) \frac{\partial S}{\partial S_i}(x)$
   - 2.2. Else if S_i is a product node:
     - 2.2.1 $\frac{\partial S}{\partial S_j} \leftarrow \frac{\partial S}{\partial S_j} + \frac{\partial S}{\partial S_i}(x) \prod_{S_k \in \mathrm{Ch}_{-S_j}(S_i)} S_k(x)$

In this top-down algorithm, we need only to have the marginal values precomputed before finding the derivatives. So in order for us to compute the derivative $\partial S/\partial S_i(x)$ for a certain node S_i, we need to have all the marginals of its

dependencies (i.e. its ancestors) already computed and stored somewhere. The algorithm for storing the marginal values is simply. Topologically sort the graph of S and for each node compute its marginal value and store it into a dynamic programming (DP) table. The topological order guarantees that all dependencies will be met, since there are no cycles. For computing the derivatives, we perform a breadth-first search (BFS) on the graph and at each node we compute the values described in the algorithm above, storing $\partial S/\partial S_j$ and $\partial S/\partial w_{i,j}$ in a DP matrix. Once we have all values, we can apply the gradient update at each node. We do this once again by performing a BFS, where at each node S_i, we apply to the weight $w_{i,j}$ the weight update $\Delta w_{i,j}$ described in Equation 2.2.

The implementation of the derivation algorithms can be found at [Geh16] under the file `learn/derive.go`. File `learn/generative.go` shows an implementation of the generative gradient update.

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).

[GD12] Robert Gens and Pedro Domingos. *Discriminative Learning of Sum-Product Networks*. Talk/Lecture. Advances in Neural Information Processing Systems 25 (NIPS 2012). 2012. URL: `http://videolectures.net/nips2012_gens_discriminative_learning/`.

[Geh16] Renato L. Geh. *GoSPN: an implementation of sum-product networks in Go*. 2016. URL: `https://github.com/RenatoGeh/gospn`.

[PD11] Hoifung Poon and Pedro Domingos. "Sum-Product Networks: A New Deep Architecture". In: *Uncertainty in Artificial Intelligence* 27 (2011).