# Plan et Notations

**Plan**

1. Algorithmes de recherche de solutions (séquences ; arbres)
2. Méthodes de prédiction discriminante non structurées (rappels + descente de gradient)
3. Méthodes de prédiction de séquences (pos tagging)
   (a) Perceptron structuré
   (b) Champs conditionnels aléatoires
   (c) Prédiction stucturée avec des modèles locaux (MEMM, NN)
4. Méthodes de prédiction d'arbres (analyse syntaxique)
   (a) Analyse en constituants
   (b) Analyse en dépendances

**Support de cours** `https://github.com/bencrabbe/parsing-at-diderot`

**Évaluation** TP Kaggle

**Pseudo Code**

$X|Y$ concatène les listes $X$ et $Y$

$x_i$ le $i$-ème élément de la liste $X$

$_{\ominus}X$ la liste $X$ sans son premier élément

$X_{\ominus}$ la liste $X$ sans son dernier élément

`http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap12_slides.pdf` `http://www.aclweb.org/anthology/C08-5001` `http://www.cs.columbia.edu/~mcollins/`

# Chapter 1

# Aspects algorithmiques

## 1.1 Prédiction de séquences

Un nombre important de problèmes de traitement automatique des langues (TAL) peuvent se formaliser comme des problèmes de recherche de chemin le plus long (ou parfois le plus court) dans un graphe acyclique orienté (DAG).

Un exemple intuitif est le problème d'étiquetage de séquences appelé étiquetage morphosyntaxique. Étant donnée une séquence de $n$ mots $w_1 \dots w_n$, on se donne pour tâche de prédire leurs parties de discours $t_1 \dots t_n$.

En première approche, on peut considérer deux méthodes pour étiqueter des séquences de mots : d'une part on peut envisager étiqueter chaque mot $w_i$ par une étiquette $t_i$ en fonction du contexte $C$ où il apparaît à l'aide d'une fonction $f : W, C \mapsto T$ (où $W$ dénote un ensemble de mots, $C$ un ensemble de contextes[1] et $T$ un ensemble de parties de discours). La première approche met en jeu un **modèle local** qui prédit chacune des étiquettes $t_i$ indépendamment de toute autre étiquette $t_j \, (i \neq j)$.

L'approche alternative, dite globale et qui fait intervenir un **modèle structuré** cherche à prédire la séquence de tags $t_1 \dots t_n$ en une fois. La fonction de prédiction est de la forme $f : W^n \mapsto T^n$, autrement dit elle envoie une séquence de $n$ mots sur des séquences d'étiquettes de $n$ tags. Utiliser cette seconde alternative revient à supposer que les séquences forment une structure intéressante pour la prédiction. Ainsi un modèle local peut faire des erreurs pour prédire le tag $t_i$ qui sont liées à l'ignorance des prédictions faites pour les tags avoisinants (comme par exemple $t_{i-1}, t_{i-2} \dots$).

[1] Typiquement un élément de $C$ sera un tuple de mots qui sont situés à gauche et à droite de $w_i$

| Prédiction globale ? | | | Prédiction locale ? | | |
|---|---|---|---|---|---|
| D | A | N | D | A | **V** |
| Le | grand | est | Le | grand | est |

Le problème de prédiction globale ou structurée est plus complexe que le problème de prédiction locale. En effet pour un jeu de tags $T$ un modèle local doit choisir $n$ fois parmi $|T|$ alternatives pour étiqueter une phrase, alors qu'un modèle global doit choisir parmi un ensemble de $|T|^n$ alternatives, ce qui a un coût : réaliser ce choix naïvement demande d'enumérer un ensemble exponentiel d'alternatives d'étiquetages de la phrase.

**Fonction de pondération** Réaliser le tagging d'une séquence de mots $\mathbf{w} = w_1 \dots w_n$ demande de choisir une séquence de tags $\mathbf{t} = t_1 \dots t_n \in T^n$. Ce choix s'appuie en général sur une **fonction de pondération** $\sigma : T^n \times W^n \mapsto \mathbb{R}$ qui associe un score à toute séquence de tags. La méthode de pondération permet alors d'ordonner les différentes séquences de tags en fonction du score qui leur est associé. La **fonction de décision** réalise le choix de la séquence à préférer en utilisant souvent une forme du type :

$$\hat{\mathbf{t}} = \underset{\mathbf{t} \in T^n}{\operatorname{argmax}} \quad \sigma(\mathbf{t}, \mathbf{w}) \tag{1.1}$$

Généralement la fonction de pondération est instanciée par une méthode d'apprentissage automatique. Indiquons également que le calcul de la solution de (1.1) consiste essentiellement à résoudre un **problème d'optimisation** de la forme :

$$m = \max_{\mathbf{t} \in T^n} \quad \sigma(\mathbf{t}, \mathbf{w}) \tag{1.2}$$

à partir de là on tire généralement la solution de (1.1) par effet de bord.

Ce qui distingue 1.2 d'un problème d'optimisation classique, c'est qu'il s'agit de chercher une valeur optimale dans un ensemble énumérable de taille finie dont les valeurs sont structurées en séquences. Bien que de taille finie, l'ensemble des solutions est en général de taille considérable de telle sorte qu'une méthode de recherche de solutions qui consiste à énumérer exhaustivement chacune des solutions est en général inutilisable.

## 1.2 Arbre de recherche de solutions

La recherche de solutions à des problèmes de type (1.2) peut s'exprimer sous la forme générale d'un problème de recherche :

- Un espace de recherche qui est un ensemble $Q$ d'états
- Un état initial $q_0 \in Q$ qui est l'état à partir duquel le tagger commence.
- Un ensemble $F \subseteq Q$ d'états finaux qui indique les états dans lequel le tagger a terminé.
- Un ensemble $A$ d'actions, dans le cas d'un tagger la seule action possible est de tagguer le mot suivant. On définit un ensemble d'actions $A$ pour lequel chaque élément représente l'attribution d'un tag au mot suivant dans la phrase.
- Un ensemble $T \subseteq Q \times Q \times A$ de transitions qui induit une structure d'arbre.
- Une fonction de coût $\sigma$ ou de score qui représente le score d'une séquence d'états calculé depuis l'état initial. On suppose que toute transition $(q_i, q_{i+1}, a_i) \in T$ se voit attribuer un coût $\psi(q_i, q_{i+1}, a_i)$ par une fonction $\psi : T \mapsto \mathbb{R}$. On pose par convention que le coût de la séquence de transitions qui mène à l'état $q_k$ est le produit des coûts des transitions qui la composent :

$$\sigma(q_k) = \prod_{0 \leqslant i < k} \psi(q_i, q_{i+1}, a_i)$$

On peut illustrer la structure d'un problème de recherche par un arbre comme en figure 1.1 pour un problème d'étiquetage morphosyntaxique (les scores sont omis). En fait un nombre très important de problèmes de TAL peut s'analyser en termes de problème de recherche de solutions dans de grands ensembles à valeurs structurées. On verra qu'on peut ainsi définir des variantes quant à la nature des états, du système de transition, des actions et que la fonction de coût dépend en général de la méthode d'apprentissage utilisée. En principe, quand l'arbre de recherche a une taille raisonnable, l'exploration des solutions peut se faire exhaustivement. Mais en pratique cette première solution est en général utilisée rarement telle quelle : on utilise plutôt des méthodes de programmation dynamique, de recherche approximative ou une combinaison des deux.

On donne en algorithme 1, un exemple d'algorithme de recherche pour un problème structuré en arbre. On peut notamment constater que la structure d'arbre de recherche n'est pas exprimée par une structure de donnée explicite mais plutôt implicitement par la structure d'appels récursifs du programme. Comme exercice, il peut être intéressant de repérer les différentes composantes du problème de recherche dans le pseudo-code (Algorithme 1).

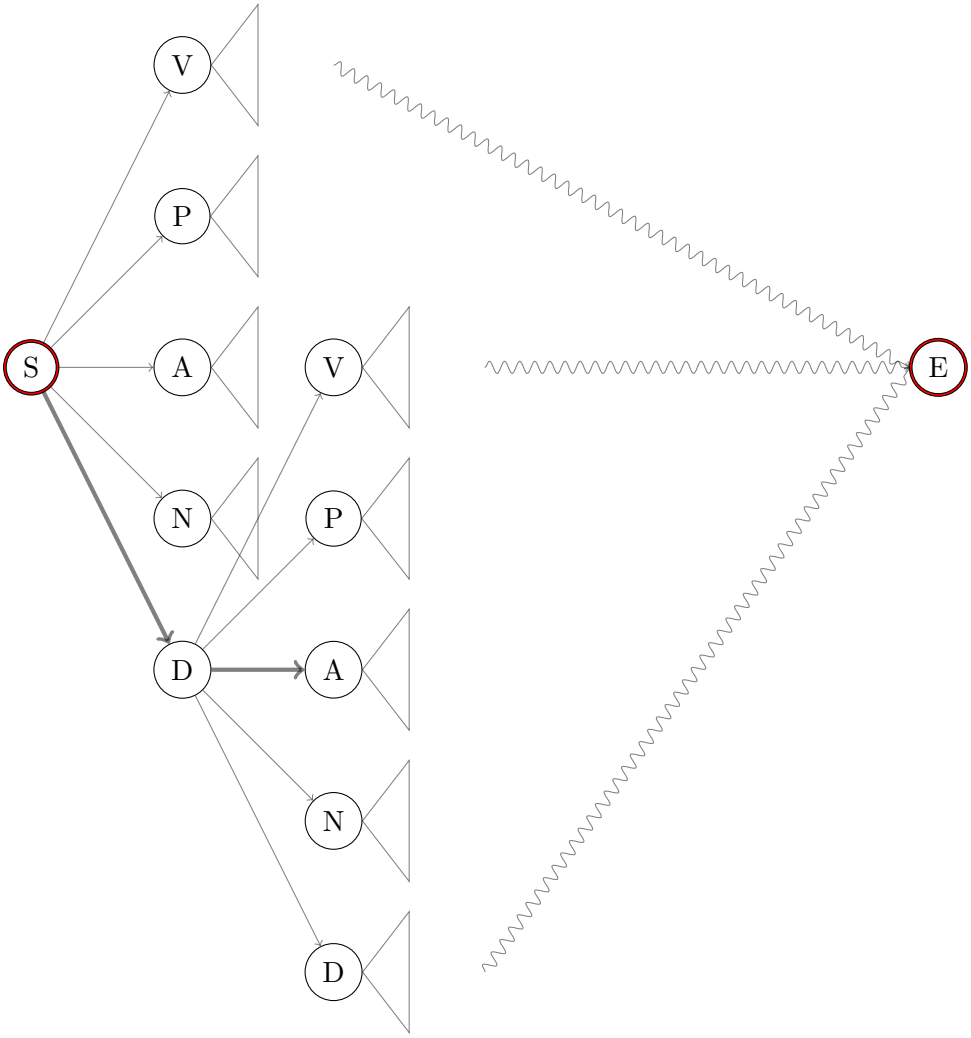



Figure 1.1: Arbre de recherche de solutions

**Algorithm 1** Algorithme de recherche structuré en arbre (cas du tagger)

```
1: function MaxSearch(q)
2:     if q ∉ F then
3:         maxscore ← 0
4:         hist ← ϵ
5:         for all (q, q', a) ∈ T do
6:             (suffscore, hist) ← MaxSearch(q')
7:             suffscore ← ψ(q, q', a) × suffscore
8:             if suffscore > maxscore then
9:                 maxscore ← suffscore
10:                hist ← a | hist
11:            end if
12:        end for
13:        return (maxscore,hist)
14:    else
15:        return (1,ϵ)
16:    end if
17: end function
```

**Exercice 1.1** *Réécrire l'algorithme de parcours d'arbre de recherche pour que le résultat renvoyé soit maintenant un couple qui comporte la valeur de score maximale mais aussi la séquence de tags correspondante*

## 1.3 Systèmes de transitions

Pour spécifier explicitement un problème de recherche de solutions en TAL, on utilise fréquemment une spécification sous forme de système de transitions. Ce type de spécification a été popularisé par la tâche d'analyse syntaxique en dépendances. Mais celle-ci est très générale, on trouve ce type de systèmes également pour spécifier des problèmes de planification en intelligence artificielle et (parfois) des méthodes de démonstration automatique.

On propose de les introduire immédiatement par un premier exemple qui caractérise explicitement une tâche de tagging. Spécifier un système de transitions revient à spécifier chacun des éléments suivants :

- L'ensemble $Q$ des états est structuré. Chaque état $q \in Q$ est un couple $(S, B)$ où $S$ est une séquence de tags déjà prédits et $B$ une séquence de mots encore à traiter dans la phrase.

- L'état initial $q_0$ est l'état $(\epsilon, B)$ où $B$ est la liste des mots de la phrase à étiqueter.
- L'ensemble des états finaux est l'ensemble $F$ des états tels que $B$ est vide.
- L'ensemble $A$ des actions est l'ensemble qui représente le jeu de tags $t \in A$ utilisé par le tagger.
- L'ensemble $T$ des transitions est la relation entre états qui satisfait le critère suivant :

  $$(S, b_0|B) \stackrel{t}{\Rightarrow} (S|t, B) \qquad (t \in A)$$

  Ce qui signifie que le premier mot de $B$ est enlevé; le tag qui correspond à l'action exécutée est ajouté à $S$.
- Le coût local d'une transition $\psi(S, B, t)$ est calculé par un modèle d'apprentissage approprié.

Notons que les systèmes de transitions peuvent servir à exprimer des automates et des transducteurs à nombre finis d'états ou des contreparties d'automates à pile. Mais il est commun en TAL de ne pas limiter l'usage des systèmes de transition au seul encodage de ce type de machines.

**Exercice 1.2** *Définir une variante de ce système de transition qui permet à un tagger d'accéder également aux mots qui précèdent dans la phrase avec la fonction de score $\psi(S, B, t)$ ou une de ses variantes.*

**Exercice 1.3** *Réécrire l'algorithme de parcours d'un arbre de recherche pour qu'il renvoie le poids de la solution de poids maximal à l'aide du système de transitions donné ci-dessus.*

```
1: function MaxSearch(S,B)
2:     if B ≠ ϵ then
3:         maxscore ← 0
4:         hist ← ϵ
5:         for all t ∈ A do
6:             (suffscore, hist) ← MaxSearch(S|t,⊖B)
7:             suffscore ← ψ(S, B, t) × suffscore
8:             if suffscore > maxscore then
9:                 maxscore ← suffscore
```

```
10:                 hist ← t | hist
11:             end if
12:         end for
13:         return (maxscore,hist)
14:     else
15:         return (1,ϵ)
16:     end if
17: end function
```

**Exercice 1.4** *Modifier la formulation de l'exercice précédent pour faire en sorte qu'il renvoie la séquence de tags ainsi que le poids de cette solution*

Lorsque l'espace des solutions est trop grand (ce qui est généralement le cas pour la plupart des problèmes de TAL) l'algorithme naïf de recherche vu jusqu'à présent, qui a une complexité exponentielle en $\mathcal{O}(A^n)$ est inutilisable. On se tourne alors vers des solutions qui s'appuient sur de la programmation dynamique (Section 1.5) ou des solutions de recherche approximative ou une combinaison des deux.

## 1.4 Les méthodes de recherche approximatives

Si il existe des algorithmes comme Viterbi et Dijkstra (et A$\star$, sections 1.5, 1.6,1.8) qui sont conçus pour donner une solution optimale au problème de recherche du chemin de poids maximal (resp. minimal) et que ces algorithmes ont une complexité polynomiale – considérée comme acceptable – ces algorithmes sont potentiellement lents lorsque les phrases sont longues où lorsque l'espace des états est de taille considérable. Par exemple, Viterbi a une complexité en $\mathcal{O}(NK^2)$, lorsque la taille du jeu de tags $K$ est importante (ce qui est notamment le cas pour des modèles dont l'historique est très riche) les temps de calcul deviennent potentiellement prohibitifs.

Dans ce type de situation, on peut faire le choix d'utiliser des méthodes de recherche de solutions qui ne garantissent pas l'optimalité, comme la recherche gloutonne ou la recherche par faisceau. Il s'agit de méthodes qui n'explorent qu'une petite partie de l'espace de solutions et qui sont en général très efficaces.

Comme il s'agit de méthodes qui n'explorent qu'une toute petite partie de l'espace des solutions (séquences de tags possibles) celle-ci sont généralement utilisées en combinaison avec une fonction de score très bien informée sur le problème à traiter de telle sorte que la partie de l'espace explorée a, par hypothèse, de grandes chances de contenir la solution optimale.

On présente ici deux méthodes couramment utilisées : la recherche gloutonne et la recherche par faisceau (beam) comme des variantes de la méthode de recherche dans un arbre de solutions (Algorithme 1) pour des systèmes de transitions. Mais ces méthodes pourraient également se formuler dans un contexte où l'espace de recherche est représenté par un graphe.

### 1.4.1 Recherche gloutonne

La méthode de recherche gloutonne est le cas dégénéré de la méthode de recherche du meilleur d'abord. À chaque itération, l'algorithme évalue le score de tous les successeurs d'un état. C'est l'unique successeur de meilleur score qui est sélectionné pour la suite de la recherche (Algorithme 2). Il

**Algorithm 2** Algorithme de recherche glouton

```
1: function GreedySearch(S,B,σ)
2:     if B ≠ ϵ then
3:         maxscore ← 0̄
4:         argmaxt ← ∅
5:         for all t ∈ T do
6:             localscore ← σ × ψ(S, B, t)
7:             if localscore > maxscore then
8:                 maxscore ← localscore
9:                 argmaxt ← t
10:            end if
11:        end for
12:        return GreedySearch(S|argmaxt,⊖B, maxscore)
13:    else
14:        return σ
15:    end if
16: end function
```

est évident que cet algorithme donne des solutions approximatives et sans garantie d'optimalité. Par contre la complexité de cet algorithme, en $\mathcal{O}(nK)$, est linéaire en temps. Autrement dit, cet algorithme est très efficace à l'usage.

L'aspect approximatif de cet algorithme peut être contrebalancé en utilisant des scores locaux très bien choisis pour guider la recherche vers une solution proche de l'optimum global. Ce scénario est très utilisé à l'heure actuelle par les systèmes pondérés par des réseaux de neurones profonds. Ceux-ci obtiennent empiriquement de très bons résultats.

**Exercice 1.5** *Augmenter l'algorithme 2 pour qu'il renvoie non seulement le score du chemin de poids maximal, mais aussi la séquence de tags de ce chemin.*

**Exercice 1.6** *Comparer le meilleur chemin renvoyé par l'algorithme glouton avec celui renvoyé par l'algorithme de Viterbi à partir de l'exemple 1.3.*

### 1.4.2 Recherche par faisceau

La recherche par faisceau est une extension de la méthode par recherche gloutonne. La recherche par faisceau est un algorithme qui progresse essentiellement en largeur dans l'arbre de recherche. À chaque itération il avance en profondeur dans $|\mathcal{B}|$ branches de l'arbre jusqu'à atteindre les feuilles, et ce sans jamais faire marche arrière.

Les $|\mathcal{B}|$ branches sélectionnées pour continuer l'exploration à l'étape suivante constituent le faisceau. Le choix des branches destinées à continuer l'exploration est heuristique. Généralement on choisit les $|\mathcal{B}|$ branches qui ont le plus haut score.

**Algorithm 3** Algorithme de recherche en faisceau

```
1: function BeamSearch(B)
2:     B' ← ∅
3:     for all ⟨S, B, σ⟩ ∈ B do
4:         if B = ϵ then                    ▷ par hypothèse le faisceau est trié
5:             return σ
6:         end if
7:         for all t ∈ T do
8:             localscore ← σ × ψ(S, B, t)
9:             B' ← B' ∪ {⟨S|t,⊖ B, localscore⟩}
10:        end for
11:    end for
12:    B ← B-Select(B')
13:    return BeamSearch(B)
14: end function
```

L'algorithme est donné en Algorithme 3. Notons que la fonction $\mathcal{B}$-Select est chargée de sélectionner $|\mathcal{B}|$ éléments dans un ensemble. Il s'agit dans la très grande généralité des cas de sélectionner les $|\mathcal{B}|$ éléments de scores les plus élevés mais des variantes sont envisageables. Par exemple la recherche par faisceau stochastique consiste à sélectionner $|\mathcal{B}|$ éléments aléatoirement proportionnellement à leur score.

L'intérêt de cette méthode est son efficacité : la complexité reste faible : $\mathcal{O}(nK|\mathcal{B}|)$, c'est-à-dire essentiellement linéaire.

On peut penser que le faisceau peut corriger les faiblesses de la méthode gloutonne mais en pratique on observe souvent que les faisceaux contiennet des hypothèses très similaires. Comme la méthode gloutonne, l'algorithme de recherche en faisceau ne garantit pas de renvoyer une solution optimale. En effet, une solution optimale qui a un mauvais score préfixe lors des premières itérations ne sera plus jamais considérée.

Il faut bien garder à l'esprit que cette méthode renvoie un pseudo-résultat maximal, ce qui peut poser problème dans certains contextes d'utilisation (comme illustré en chapitre XX)

**Exercice 1.7** *Augmenter l'algorithme 3 pour qu'il renvoie non seulement le score du chemin de poids maximal, mais aussi la séquence de tags de ce chemin.*

## 1.5 Programmation dynamique

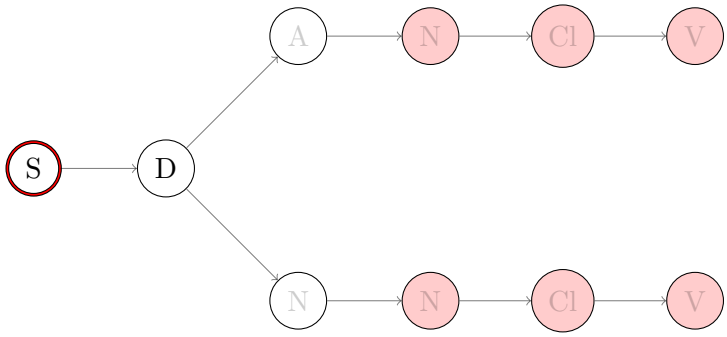
Une des faiblesses de l'algorithme de recherche naïf de solutions structuré en arbre (Algorithme 1) est qu'il réplique une quantité très importante de calculs.

Les solutions approximatives présentées jusqu'à présent permettent d'éviter une reduplication outrancière des calculs en limitant le nombre d'hypothèses explorées. L'inconvénient est qu'elles ne garantissent pas l'optimalité de la solution renvoyée.

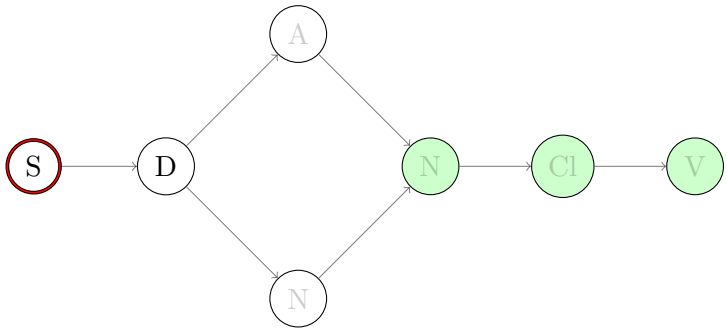
Les méthodes de programmation dynamique cherchent à renvoyer une solution optimale au problème de recherche en évitant la reduplication de calculs. On propose de motiver la technique par un exemple introductif en considérant deux séquences de tags :

| | | | | |
|---|---|---|---|---|
| D | A | N | Cl | V |
| D | N | N | Cl | V |
| La | belle | porte | le | cache |

qui diffèrent par un élément. On constate qu'une méthode de recherche en arbre va recalculer au moins deux foix la valeur du suffixe *N Cl V*. De manière générale la méthode de recherche de solutions en arbre reduplique une quantité considérable de calculs de suffixes, ce qui est largement inefficace.




L'idée des méthodes de **programmation dynamique**, c'est d'éviter les reduplications de calculs inutiles pour réutiliser des résultats intermédiaires (partage de calcul). Ainsi l'espace des états est organisé en graphe (DAG) plutôt qu'en arbre. Cette nouvelle organisation permet de proposer des solutions algorithmiques en temps polynomial plutôt qu'en temps exponentiel aux problèmes de recherche qui nous concernent.




### 1.5.1 Graphe Acyclique orienté

**Définition 1.1 (DAG)** *Un graphe acyclique orienté (DAG) est un graphe $G = \langle V, E \rangle$ où $V$ est un ensemble de noeuds et $E$ un ensemble d'arcs ($E \subseteq V \times V$) tel qu'il ne comporte pas de circuit. Dans le cas pondéré, on y ajoute une fonction $s : E \mapsto \mathbb{R}$ qui donne un score à chacun des arcs.*

**Définition 1.2 (Arc entrants)** *Soit un noeud $x \in V$, l'ensemble $AE(x) = \{(y, x) \,|\, (y, x) \in E, y \in V\}$ est l'ensemble des arcs entrants sur ce noeud.*

**Définition 1.3 (Arc sortants)** *Soit un noeud $x \in V$, l'ensemble $AS(x) = \{(x, y) \,|\, (x, y) \in E, y \in V\}$ est l'ensemble des arcs sortants de ce noeud.*

**Définition 1.4 (Chemin)** *Un chemin de longeur $n$ est une séquence de noeuds $\pi \in V^n$ de la forme $\pi = v_1 v_2 \dots v_n$ tel que $(v_i, v_{i+1}) \in E$ pour tout $1 \leqslant i < n$. Le score $\sigma(\pi)$ d'un chemin est le produit :*

$$\sigma(\pi) = \prod_{i=1}^{n-1} \psi(v_i, v_{i+1})$$

**Définition 1.5 (Poids maximal depuis la source)** *Soit un* DAG *dont un noeud distingué $s \in V$ est appelé noeud source. En notant $P(x)$ l'ensemble des chemins qui mènent de la source à $x$, on définit le poids maximal de $x$ comme suit :*

$$\delta(x) = \begin{cases} 1 & si\ x = s \\ max_{\pi \in P(x)}\, \sigma(\pi) & sinon \end{cases}$$

### 1.5.2 Mémorisation de résultats intermédiaires

Un problème de programmation dynamique est typiquement formulé comme un problème d'optimisation entre différentes alternatives : il s'agit par exemple de trouver un chemin de poids maximum parmi plusieurs chemins pondérés. On illustre en figure 1.2 un exemple de structuration du problème de recherche en DAG.

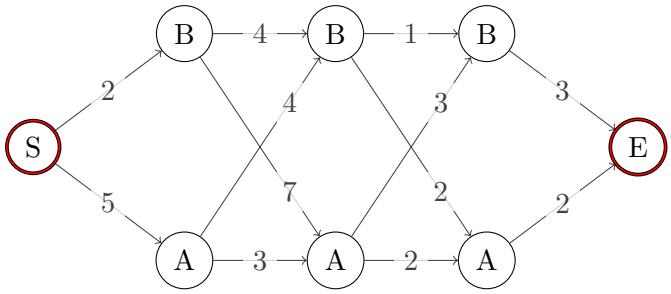



Figure 1.2: Exemple de DAG de programmation dynamique

Lorsque certains sous-problèmes sont à recalculer plusieurs fois, on dit qu'il y a **recouvrement de sous-problèmes**. Les techniques de programmation dynamique consistent à éviter l'évaluation multiple d'un même sous-problème en mémorisant les solutions intermédiaires.

Plusieurs techniques pour réaliser la mémorisation seront présentées mais celles-ci font généralement appel à une table dite table de programmation dynamique pour mémoriser les résultats intermédiaires. Celle-ci mémorise

les valeurs $\delta(x)$ des états $x$ intermédiaires (noeuds du DAG) évalués lors de la résolution du problème.

Une solution directe pour exprimer cette idée est la technique dite de **mémoisation**. Celle-ci repose sur l'usage de **mémo-fonctions**. L'idée est de mémoriser la solution $\delta(x)$ dans une table dès que celle-ci est déterminée. Cette table est alors consultée dans les étapes ultérieures de l'algorithme de telle sorte que la valeur mémorisée est réutilisée au lieu de réexécuter l'appel récursif.

L'algorithme 4 illustre le principe de la mémoisation. On suppose que le DAG a un état final $q_f$, et que la table *memo* est initialisée à $\delta(x) = 0$ pour tout $x \in V$ (sauf $\delta(q_0) = 1$). La fonction est initialement appelée avec le paramètre $q_f$.

**Algorithm 4** Algorithme de recherche d'une valeur optimale mémoisé

```
1: function MaxSearchMemo(x)
2:     if δ(x) ≠ 0 then                                  ▷ Score mémoisé
3:         return δ(x)
4:     end if
5:     for all y ∈ AE(x) do
6:         δ(x) ← Max(δ(x),MaxSearchMemo(y) ×ψ(y, x))
7:     end for
8:     return δ(x)
9: end function
```

On illustre en figure 1.3 le DAG résultant de l'exécution de l'algorithme de mémoisation sur le DAG donné en Figure 1.2.

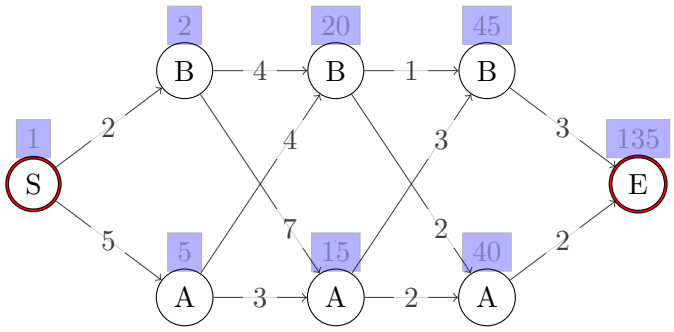



Figure 1.3: DAG annoté par $\delta(x)$

**Exercice 1.8** *Reformuler l'algorithme 4 de telle sorte que le chemin optimal soit celui de poids minimum. Simuler son exécution sur papier.*

**Exercice 1.9** *Reformuler l'algorithme 4 en supprimant la conditionnelle qui réalise la mémoisation. Tenter de le simuler sur papier.*

## 1.6 Algorithme de Viterbi

L'algorithme mémoisé présenté dans la section précédente peut se reformuler par une version plus directe. C'est l'**algorithme de Viterbi**.

**Définition 1.6 (Ordre topologique)** *Un ordre topologique sur un* DAG $G = (V, E)$ *est tout ordre total sur les noeuds* $V$ *de ce* DAG *tel que pour tout couple de noeuds* $(x, y) \in E$, $x \prec y$. *Il existe en général plusieurs ordres topologiques valides pour un* DAG *donné.*

L'algorithme de Viterbi est un algorithme de recherche du chemin de poids maximal dans un DAG (Figure 5). En supposant un noeud source $s$ unique dont le poids $\delta(s)$ est initialisé à 1, l'algorithme parcourt le DAG en suivant l'ordre topologique. Chaque noeud est à son tour valué par le poids $\delta(s)$ du chemin maximal qui mène de la source jusqu'à ce noeud. Toute l'idée de l'algorithme consiste à mémoriser les poids $\delta(s)$ au fur et à mesure qu'ils sont calculés pour les réutiliser lors de calculs ultérieurs.

**Algorithm 5** Algorithme de Viterbi

```
function ViterbiTopologique(S,V,s)
    TriTopologique(S, V, s)
    δ(s) ← 1
    for all s ∈ S (suivant ordre topologique) do
        δ(s) ← 0
        for all (s', s) ∈ AE(s) do
            δ(s) ← Max(δ(s), δ(s') × ψ(s', s))
        end for
    end for
end function
```

On donne en figure 1.3 un exemple de résultat de l'exécution de l'algorithme sur un cas concret. Chacun des noeuds du DAG est annoté (case bleue) par la valeur du $\delta(x)$ qui lui correspond.

**Exercice 1.10 (Limitation aux DAGS)** *L'algorithme de Viterbi ne peut pas être utilisé si le graphe contient au moins un cycle (le graphe n'est pas un DAG). Expliquer pourquoi par un exemple.*

**Exercice 1.11 (Extraction de la séquence de poids maximal)** *Donner une extension de l'algorithme donné en figure 5 qui permet de renvoyer comme résultat non seulement le score du chemin de poids maximal (plus long chemin) mais aussi la séquence de tags de ce chemin.*

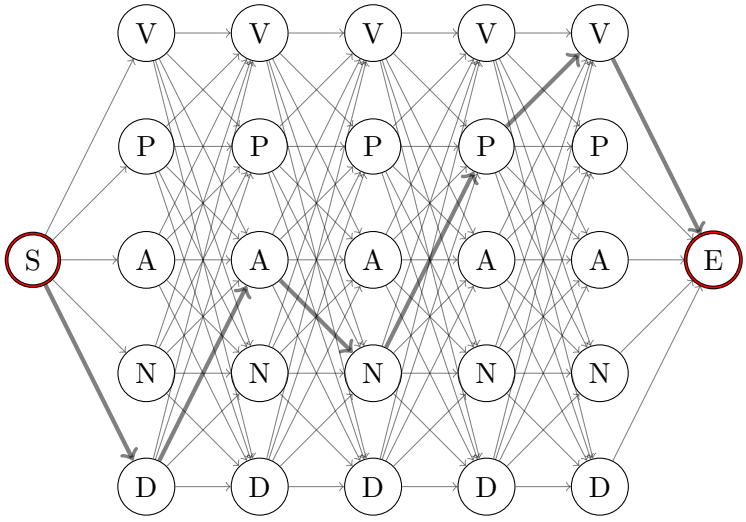



Figure 1.4: Graphe acyclique orienté pour énumérer les solutions de manière compacte dans un cas de tagging

**Viterbi pour l'étiquetage morphosyntaxique** Dans le cas de l'étiquetage morphosyntaxique en TAL, le DAG de programmation dynamique est conventionnellement construit comme illustré en figure 1.4. Pour chaque occurrence de mot $w_i$ dans la phrase, on construit un noeud correspondant à chacun des tags possibles. Chacun de ces noeuds est connecté par un arc à l'ensemble des noeuds de la position suivante $w_{i+1}$ dans la phrase.

En utilisant cette représentation, évaluer successivement les noeuds de gauche à droite, c'est-à-dire en valuant l'ensemble des mots de $w_i$ avant ceux de $w_{i+1}$ revient à valuer le graphe suivant un ordre topologique valide.

Dans ce contexte spécifique, il est classique de stocker les quantités $\delta$ dans une matrice $\Delta$ à $i$ colonnes et $j$ lignes. Chaque ligne correspond à un pos tag parmi un ensemble de $K$ tags et chaque colonne à une position dans une phrase de $N$ mots. Ainsi $\delta(i, j)$ correspond au score du noeud en position $i$ taggué par le tag $j$. L'algorithme prend alors la forme donnée en Algorithme 6 où on choisit de ne pas expliciter l'état cible noté $E$ dans les exemples précédents.

Cette dernière version permet également de mettre en évidence que la complexité de l'algorithme est en $\mathcal{O}(NK^2)$. Autrement dit la méthode de programmation dynamique permet de donner une solution en temps polynomial à un problème dont la résolution naïve est en temps exponentiel.

**Algorithm 6** Algorithme de Viterbi (version tabulaire)

```
function VITERBI(N,K,s)
    for 0 ⩽ j < K do                                   ▷ Initialisation
        δ(0, j) ← ψ(s, j)
    end for
    for 0 < i < N do                                   ▷ Recurrence
        for 0 ⩽ j < K do
            δ(i, j) ← 0
            for 0 ⩽ k < K do
                δ(i, j) ← MAX(δ(i, j), δ(i − 1, k) ⊗ ψ(k, j))
            end for
        end for
    end for
end function
```

Ce type de formulation est fréquemment utilisée dans les implémentations. Signalons que dans un contexte d'implémentation il est d'usage de ne pas représenter explicitement les arcs du DAG mais plutôt de construire directement la matrice de scores $\Delta$.

**Exercice 1.12 (Historique de dérivation)** *Augmenter l'algorithme 6 pour qu'il renvoie également la séquence de tags de score maximal.*

**Equation de Bellmann** Les techniques de programmation dynamique ne sont pas limitées au cas du tagging mais sont héritées de méthodes plus

générales de résolution de problèmes d'optimisation de fonctions récursives.

En notant $\delta(x_{-1}), \delta(x_{-2}) \dots \delta(x_{-k})$ les solutions de sous problèmes qu'il faut résoudre récursivement pour trouver la solution du problème $\delta(x)$, la résolution par programmation dynamique consiste à calculer $\delta(x)$ à partir des valeurs mémorisées pour les sous-problèmes à traiter $\delta(x_{-1}), \delta(x_{-2}) \dots \delta(x_{-k})$ en suivant le schéma donné par l'équation suivante :

$$\delta(x) = \begin{cases} 1 & \text{si } x = q_0 \\ \max\Big(\delta(x_{-1}) \times \psi(x_{-1}, x), \dots, \delta(x_{-k}) \times \psi(x_{-k}, x)\Big) & \text{sinon} \end{cases} \tag{1.3}$$

qui est appelée équation de programmation dynamique ou **équation de Bellmann**. Il s'agit d'une formule récursive qui fait intervenir deux opérations : MAX est une opération d'aggrégation et $\times$ est une opération de composition destinée à calculer le score de chemins dans le DAG de calcul.

Des variantes et des généralisations sur les opérations d'aggrégation et de composition sont possibles, c'est ce qu'on propose d'examiner dans la section suivante.

## 1.7 Abstractions algébriques

Les algorithmes que nous présentons ici sont destinés à être utilisés en combinaison avec une méthode d'apprentissage. C'est cette dernière qui donne une méthode pour valuer la fonction $\psi$ et pour estimer les éventuels paramètres qui lui sont associés.

L'algorithme de Viterbi est un algorithme qui ne fait rien d'autre que de calculer un *max* ou un *argmax* pour un très grand ensemble de séquences.

Or chaque méthode d'apprentissage manipule des poids qui se combinent différemment. Par exemple pour calculer le poids d'une séquence avec un HMM il faut réaliser une multiplication alors qu'avec un perceptron il faut réaliser une addition. L'algorithme de Viterbi peut être réutilisé pour chacun de ces paradigmes mais avec les ajustements nécessaires au système de pondération du paradigme en question.

Nous introduisons ici la notion de demi-anneau car elle permet de spécifier l'interface entre les algorithmes présentés ici et des modèles d'apprentissage variés. Plus spécifiquement cette notion aide à caractériser les conditions d'utilisation d'un algorithme de recherche (notamment celui de Dijkstra) pour un un modèle d'apprentissage. En second lieu elle donne des points de repères sur les paramètres à considérer pour adapter les algorithmes aux

problèmes d'apprentissage traités. Et finalement cela permet de dériver de nouvelles utilisations pour un algorithme donné.

Un **demi-anneau** est une structure algébrique abstraite qui permet de généraliser le calcul avec des nombres naturels (ensemble $\mathbb{N}$)[2]. Le calcul suppose deux opérations : l'addition qui est commutative et qui a un élément neutre noté $\bar{0}$, la multiplication qui peut être commutative et qui a un élément neutre noté $\bar{1}$. De plus la multiplication distribue sur l'addition et $\bar{0}$ est absorbant pour la multiplication. Contrairement aux **anneaux**, les demi-anneaux n'ont pas nécessairement un additif inverse tel que $a \oplus -a = \bar{0}$.

Un demi anneau est un quintuple $(E, \oplus, \otimes, \bar{0}, \bar{1})$ où $E$ est un ensemble, $\oplus$ l'opération d'addition, $\otimes$ l'opération de multiplication, $\bar{0}$ le neutre pour l'addition et $\bar{1}$ le neutre pour la multiplication.

L'intérêt d'utiliser cette généralisation dans la spécification des algorithmes est de donner une formulation unique d'un algorithme qu'il ne reste plus qu'à instancier avec le demi-anneau correspondant au problème d'apprentissage à traiter.

On donne en Figure 7 une reformulation de l'algorithme de Viterbi (Algorithme 5) en utilisant la notation en semi-anneau. Par ailleurs, on donne en figure 1.5 quelques exemples de demi-anneaux qui sont utilisés dans ce cours.

**Algorithm 7** Algorithme de Viterbi avec opérations abstraites

```
function VITERBI(S,V,s)
    TRITOPOLOGIQUE(S, V, s)
    δ(s) ← 1̄
    for all s ∈ S (suivant ordre topologique) do
        δ(s) ← 0̄
        for all (s', s) ∈ AE(s) do
            δ(s) ← ⊕(δ(s), δ(s') ⊗ ψ(s', s))
        end for
    end for
end function
```

On termine par donner quelques définitions et propriétés qui seront utiles notamment dans les sections suivantes. Un demi-anneau $(E, \oplus, \otimes, \bar{0}, \bar{1})$ est **idempotent** si $e \oplus e = e \qquad (\forall e \in E)$. Si le demi-anneau est idempotent,

[2]Le calcul avec des entiers (ou des réels) se généralise par une structure d'**anneau**. C'est-à-dire qu'on a nécessairement des additifs inverses.

| Nom | Ensemble | $\oplus$ | $\otimes$ | $\bar{0}$ | $\bar{1}$ | Usage possible |
|---|---|---|---|---|---|---|
| Viterbi | $[0,1]$ | max | $\times$ | 0 | 1 | meilleure séquence (HMM) |
| Viterbi-CRF | $\mathbb{R}^+$ | max | $\times$ | 0 | 1 | meilleure séquence (CRF) |
| Viterbi-réel | $\mathbb{R} \cup \{-\infty\}$ | max | $+$ | $-\infty$ | 0 | meilleure séquence (perceptron) |
| Tropical | $\mathbb{R}^+ \cup \{-\infty\}$ | min | $+$ | $-\infty$ | 0 | plus court chemin ($-log(p)$) |
| Avant | $[0,1]$ | $+$ | $\times$ | 0 | 1 | Algorithme avant (HMM) |
| Avant-CRF | $\mathbb{R}^+$ | $+$ | $\times$ | 0 | 1 | Algorithme avant (CRF) |
| Comptage | $\mathbb{N}$ | $+$ | $\times$ | 0 | 1 | Compte le nombre de séquences |

Figure 1.5: Quelques demi-anneaux utilisés dans ce cours

on peut alors définir la relation d'ordre partiel $\leqslant$ comme suit :

$$a \leqslant b \Leftrightarrow (a \oplus b) = a$$

appelée ordre naturel de $E$. On dit qu'un demi-anneau a la propriété de **supériorité** si pour tout couple $a, b \in E$:

$$a \leqslant a \otimes b, \qquad b \leqslant a \otimes b$$

Ce qui revient à dire que combiner par multiplication deux quantités $a, b$ renvoie comme résultat une valeur plus grande. Cette propriété est requise pour utiliser l'algorithme de Dijkstra ou ses dérivés (y compris l'algorithme de Knuth) dans un contexte de prédiction structurée.

**Exercice 1.13 (Algorithme somme produit)** *Instancier l'algorithme de Viterbi (Algorithme 5) en utilisant le demi anneau appelé Avant-*CRF *(Figure 1.5). Simuler ce nouvel algorithme à l'aide de l'exemple de* DAG *donné en figure 1.3. Donner en français une explication de ce que cet algorithme calcule.*

## 1.8 Algorithme de Dijkstra et recherche A$\star$

Dans certains cas, il est possible de reformuler le problème de recherche de la séquence de poids maximal dans un DAG comme un problème du plus court chemin dans un graphe de telle sorte que le problème se résoud avec l'algorithme de Dijkstra.

Intuitivement l'algorithme de Dijkstra peut s'utiliser dans un contexte de tagging lorsque les scores associés aux arcs s'interprètent comme des mesures dites de **surprise** que l'on obtient à partir de probabilités (surprise $= -\log_2(p)$).

**Algorithm 8** Algorithme de Dijkstra

```
function DIJKSTRA(V,E,s)
    δ(x) ← ∞        (∀x ∈ V)
    δ(s) ← 0
    PUSHPRIORITYQUEUE(Q, V)
    while Q ≠ ∅ do
        x ← POPMIN(Q)
        for y ∈ AS(x) do
            δ(y) ← MIN(δ(y), δ(x) + ψ(x, y))
            UPDATEPRIORITYQUEUE(Q, δ(y))
        end for
    end while
end function
```

Commençons par rappeler le fonctionnement de l'algorithme de Dijkstra dans le cas classique. On suppose un graphe dont les arcs sont pondérés par des distances entre des stations de métro. Le problème est de trouver le plus court chemin entre deux stations, la première est appelée la source, la seconde la destination.

L'algorithme suppose un graphe $G = \langle S, E \rangle$ une file de priorité $Q$. Une file de priorité est une structure de donnée qui maintient ses arguments triés. On peut ainsi lui ajouter des éléments pondérés et extraire l'élément de poids minimal[3].

L'algorithme de Dijkstra, détaillé en Algorithme 8, est conceptuellement très simple. L'invariant se résume comme suit. L'algorithme procède en évaluant comment progresser à partir du noeud $s$ du graphe qui est le plus proche de la source. Aucun chemin alternatif plus court ne peut mener à $s$ sinon ce serait un noeud sur ce chemin alternatif qui serait sélectionné à la place de $s$. L'algorithme termine quand le noeud $s$ est le but à atteindre.

Il faut remarquer que l'algorithme de Dijkstra fonctionne uniquement si les poids des chemins sont positifs. Dans le cas où certains chemins ont des poids négatifs, l'algorithme donne un résultat incorrect.

### 1.8.1 Algorithme A⋆

L'algorithme $A\star$ est un algorithme de recherche du plus court chemin qui est à voir comme une extension de l'algorithme de Dijkstra. L'invariant de

[3]On suppose que les poids des éléments définissent la relation d'ordre.

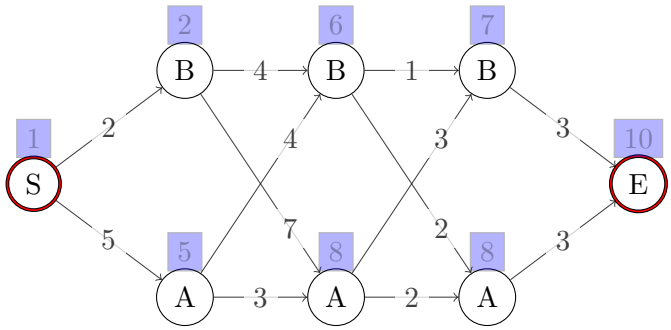



Figure 1.6: Exemple de recherche du plus court chemin avec Dijkstra

l'algorithme de Dijkstra est conservé : à chaque itération c'est le noeud $s$ qui a le coût $\delta(s)$ le plus faible qui est sélectionné pour la suite de l'exploration.

Par contre l'algorithme $A\star$ change la méthode d'évaluation du coût. Ce n'est plus uniquement $\delta(s)$ qui représente le coût mais la combinaison :

$$\phi(s) = \delta(s) + h(s)$$

Cette fois-ci, le coût $c(s)$ est la somme du coût $\delta(s)$ du chemin déjà parcouru et d'une heuristique $h(n)$ qui estime le coût du chemin qui reste à parcourir. Ainsi l'algorithme de Dijkstra classique est un algorithme $A\star$ pour lequel $h(s) = 0\,(\forall s \in S)$.

**Algorithm 9** Algorithme A star

```
function ASTAR(V,E,s)
    δ(x) ← ∞        (∀x ∈ V)
    δ(s) ← 0
    PUSHPRIORITYQUEUE(Q, V)
    while Q ≠ ∅ do
        x ← POPMIN(Q)
        for y ∈ AS(x) do
            δ(y) ← MIN(δ(y), δ(x) + ψ(x, y))
            UPDATEPRIORITYQUEUE(Q, δ(y) + h(y))
        end for
    end while
end function
```

**Conception de l'heuristique** La difficulté lors de la conception d'un algorithme $A\star$ consiste à définir une heuristique qui permette d'obtenir une **solution optimale**. Un algorithme de recherche de court chemin qui renvoie une solution optimale est un algorithme qui renvoie effectivement comme résultat la valeur du plus court chemin entre deux points. C'est ce que garantit l'algorithme de Dijkstra.

Si l'heuristique $h(s)$ est mal définie l'algorithme $A\star$ n'est pas garanti de renvoyer la solution optimale. On propose d'illustrer ce point par l'exemple

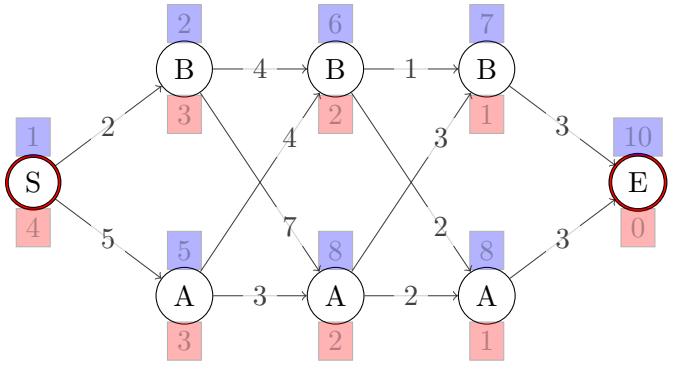



Figure 1.7: Exemple d'heuristique valide si $\psi(x, y) \geqslant 1$

Une heuristique $h$ est dite **admissible** si pour tout $s \in S$ la distance estimée $h(s)$ n'est jamais strictement supérieure à la distance minimale réelle qu'il reste à parcourir pour atteindre la destination depuis $s$.

La seconde condition que l'heuristique $h$ doit satisfaire pour garantir l'optimalité de l'algorithme est la condition de **monotonie** (ou de consistance) :

$$h(s) \leqslant \psi(s, s') + h(s') \qquad \forall (s, s') \in AS(s)$$

Ce qui revient à dire que l'estimation du coût pour arriver à destination depuis $n$ doit être inférieur au coût pour arriver à destination à partir de chacun de ses successeurs dans le graphe. C'est une généralisation de la condition excluant les chemins de poids négatif pour l'algorithme de Dijkstra. On peut alternativement reformuler cette condition en incluant le terme $\delta(s)$ :

$$\delta(s) + h(s) \leqslant \delta(s) + \psi(s, s') + h(s') \qquad \forall (s, s') \in AS(s)$$

pour exprimer explicitement que la longueur d'un chemin ne peut pas raccourcir. En général satisfaire l'une des deux conditions permet de satisfaire

l'autre. Mais ce n'est pas toujours vrai.

En TAL la recherche $A\star$ a surtout été utilisée dans des contextes d'analyse syntaxique automatique (comme extensions de l'algorithme de Knuth).

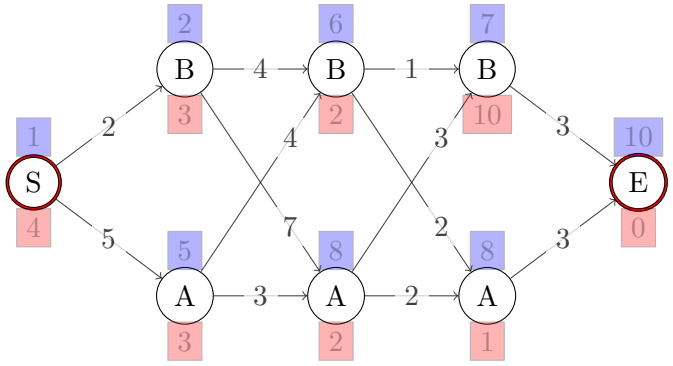



Figure 1.8: Exemple d'heuristique invalide si $\psi(x, y) \geqslant 1$

En résumé, la conception d'une bonne heuristique $A\star$ est en général non triviale. L'heuristique évidente $h(s) = 0$ n'aide pas à réaliser une meilleure recherche que l'algorithme de Dijkstra. Une heuristique plus informative qui préserve l'optimalité demande de vérifier des propriétés qui ne sont pas triviales à satisfaire en pratique. Pour cette raison, on trouve souvent dans la littérature des heuristiques approximatives qui sacrifient l'optimalité de la solution.

# Chapter 2

# Rappels de classification non structurée

= modèles + descente de (sous-) gradient.

## 2.1 Minimum d'une fonction strictement convexe

### 2.1.1 Fonctions monovariées

Si on sait que la fonction est strictement convexe (resp. concave) — ce qui est le cas pour tous les modèles d'apprentissage décrits dans ce cours — (à l'exception des réseaux de neurones), alors il existe un minimum (resp. maximum) global unique. On trouve ce minimum en résolvant :

$$\frac{\delta f(x)}{\delta x} = 0 \tag{2.1}$$

Par exemple si $f(x) = (x-3)^2$, on a que $\frac{\delta f(x)}{\delta x} = 2(x-3)$. En résolvant $2(x-3) = 0$, on trouve $x = 3$. Comme la fonction est convexe, on sait que $x = 3$ est le minimum unique. Cette méthode de résolution est appelée méthode analytique.

Pour la très grande majorité des fonctions utilisées par les modèles d'apprentissage utilisés en TAL, résoudre (2.1) n'est pas possible analytiquement. On utilise plutôt une méthode de résolution numérique appelée **descente de gradient**. Étant donnée une première hypothèse $x_0 = c$, la méthode de descente de gradient consiste à calculer une suite d'hypothèses $x_0, x_1, \ldots x_k$ qui approchent progressivement la solution. La dérivée en un point $x_i$, $\frac{\delta f(x_i)}{\delta x}$ nous donne la pente de la tangente en $x_i$. On sait que pour

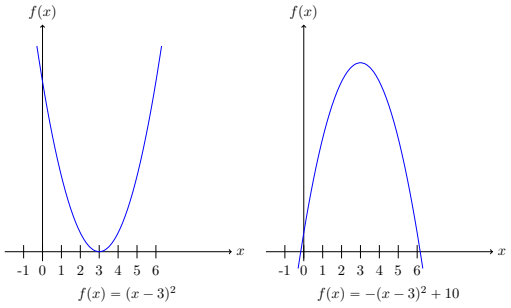



Figure 2.1: Fonctions strictement convexe (gauche) et concave (droite)

se rapprocher de la solution, il faut choisir un point $x_{i+1} = x_i - \frac{\delta f(x_i)}{\delta x}$. Pour se déplacer plus ou moins vite dans une direction, on utilise généralement un pas de gradient noté $\alpha$ de telle sorte qu'on choisit $x_{i+1}$ comme suit :

$$x_{i+1} = x_i - \alpha \frac{\delta f(x_i)}{\delta x}$$

Prenons l'exemple de la fonction $f(x) = (x-3)^2$ et supposons $x_0 = 0$ et $\alpha = 0.1$, le début de la suite $x_0, x_1, x_2 \ldots$ prend la forme suivante :

| $x_i$ | $\alpha\, 2(x-3)$ |
|---|---|
| 0 | $0.1 \times 2 \times (0-3) = -0.6$ |
| 0.6 | $0.1 \times 2 \times (0.6-3) = -0.48$ |
| 1.08 | $0.1 \times 2 \times (1.08-3) = -0.384$ |
| 1.464 | $\ldots$ |

### 2.1.2 Fonctions de plusieurs variables

La méthode de descente de gradient se généralise au cas de fonctions de plusieurs variables, de la forme $f(\mathbf{w})$, où $\mathbf{w}$ est un vecteur de variables ($\mathbf{w} \in \mathbb{R}^d$).

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \nabla f(\mathbf{w})$$

où $\nabla f(\mathbf{w})$ est le **vecteur gradient** au point $\mathbf{w}$. Un vecteur gradient est un vecteur de dérivées partielles tel que $\nabla f(\mathbf{w}) = \frac{\partial f}{\partial w_1}(\mathbf{w}) \ldots \frac{\partial f}{\partial w_d}(\mathbf{w})$. Pour trouver le minimum d'une fonction strictement convexe $f(\mathbf{w})$, il faut alors

résoudre $\nabla f(\mathbf{w}) = \mathbf{0}$, c'est-à-dire un système d'équations de la forme :

$$\begin{bmatrix} \frac{\partial f}{\partial w_1}(\mathbf{w}) & = & 0 \\ \vdots & \vdots & \vdots \\ \frac{\partial f}{\partial w_d}(\mathbf{w}) & = & 0 \end{bmatrix}$$

Prenons l'exemple de la fonction de deux variables $f(w_1, w_2) = w_1^2 + w_2^2 + 2w_1 + 8w_2$ dont le graphe est représenté en figure 2.2. On a que $\frac{\partial f}{\partial w_1}(\mathbf{w}) = 2w_1 + 2$ et que $\frac{\partial f}{\partial w_2}(\mathbf{w}) = 2w_2 + 8$. Pour annuler le gradient, il faut donc résoudre analytiquement le système d'équations :

$$\begin{bmatrix} 2w_1 + 2 & = & 0 \\ 2w_2 + 8 & = & 0 \end{bmatrix}$$

ce qui donne $(w_1, w_2) = (-1, -4)$ et ce qui nous permet de déterminer que sa valeur au minimum est $-1^2 - 4^2 - 2 - 32 = -17$. Par la suite on utilisera la notation suivante pour référencer ces deux valeurs :

$$\begin{aligned} (-1, -4) &= \underset{\mathbf{w} \in \mathbb{R}^2}{\operatorname{argmin}} f(\mathbf{w}) \\ -17 &= \min f(\mathbf{w}) \end{aligned}$$

La résolution par la méthode numérique suit le même principe que précédemment. Pour le déroulé de l'exemple, on suppose que $\mathbf{w}_0 = (0, 0)$ et $\alpha = 0.1$.

| $\mathbf{w}_i$ | $\alpha\,(2w_1 + 2))$ | $\alpha\,(2w_2 + 8))$ |
|---|---|---|
| (0,0) | $0.1 \times (2 \times 0 + 2) = 0.2$ | $0.1 \times (2 \times 0 + 8) = 0.8$ |
| (-0.2,-0.8) | $0.1 \times (2 \times -0.2 + 2) = 0.16$ | $0.1 \times (2 \times -0.8 + 8) = 0.64$ |
| (-0.36,-1.44) | $0.1 \times (2 \times -0.36 + 2) = 0.128$ | $0.1 \times (2 \times -1.44 + 8) = 0.512$ |
| (-0.488,-1.952) | ... | ... |

La méthode de résolution numérique pour fonctions strictement convexes de plusieurs variables se résume par un algorithme dit de descente de gradient qui est explicité en Algorithme 10. Le critère d'arrêt est en général un test de convergence de la forme $f(\mathbf{w}_{i+1}) \approx f(\mathbf{w}_i)$. On peut également donner une borne sur le nombre maximal d'itérations si la précision de la solution n'est pas essentielle.

Un des points difficile concerne le choix du pas de gradient $\alpha$. Trop petit, l'algorithme progresse trop lentement vers la solution, trop grand, l'algorithme risque de diverger. Il existe un très grand nombre de méthodes de descente de gradient qui traitent les problèmes mentionnés ici et qui

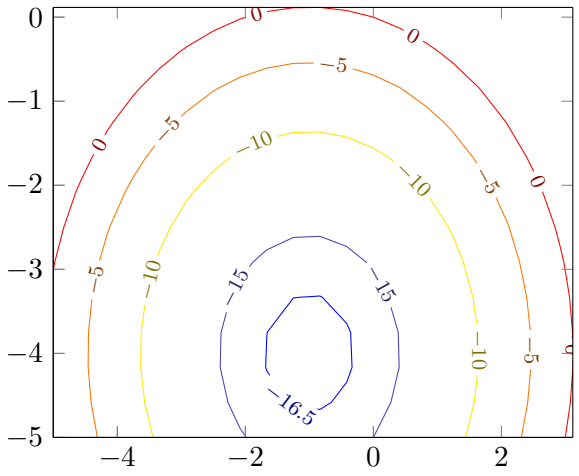



Figure 2.2: Contour de la fonction $f(w_1, w_2) = w_1^2 + w_2^2 + 2w_1 + 8w_2$

**Algorithm 10** Algorithme de descente de gradient

**function** GRADIENTDESCENT($\alpha$,$f(\mathbf{w})$)
  $\mathbf{w} \leftarrow \mathbf{0}$
  **while** non convergence **do**
    $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla f(\mathbf{w})$
  **end while**
  **return w**
**end function**

ont fait l'objet d'études considérables en calcul numérique et en optimisation. Les méthodes utilisées en TAL, et abordées dans ce cours restent en général particulièrement simples et approximatives. Celles-ci sont adaptées au traitement de jeux de données (1) de très grande dimensionalité et (2) comportant un très grand nombre d'exemples. Ceci dit, dans certains cas, il peut être utile de s'appuyer sur des librairies de calcul numérique comme `scipy.optimize`.

**Exercice 2.1 (Montée de gradient)** *La fonction* $f(w_1, w_2) = -(w_1^2 + w_2^2 + 2w_1 + 8w_2)$ *est strictement concave. Calculer son gradient et définissez un algorithme de montée de gradient qui permet de déterminer numériquement son maximum.*

## 2.2 Régression logistique

Les modèles logistiques sont des modèles qui permettent de prédire une variable binaire $Y = \{0, 1\}$ à partir d'un certain nombre de prédicteurs notés $\mathbf{x} \in \mathbb{R}^d$ et de paramètres $\mathbf{w} \in \mathbb{R}^d$. Il s'agit d'une brique de base pour plusieurs modèles de classification plus complexes.

**Modèle** Supposons que l'on ait observé les résultats de deux tests d'un étudiant au contrôle continu. Celui-ci obtient la note $x_1 = 6$ et la note $x_2 = 7$, ce qui constitue le vecteur d'observations $\mathbf{x}$. Un modèle logistique permet par exemple de donner une probabilité de succès (de réussite à l'examen) $P(Y = 1|\mathbf{x}; \mathbf{w})$ à l'aide de la formule suivante :

$$P(Y = 1|\mathbf{x}; \mathbf{w}) = \frac{e^{\mathbf{w}^T\mathbf{x}}}{1 + e^{\mathbf{w}^T\mathbf{x}}} \tag{2.2}$$

où le vecteur de poids $\mathbf{w}$ est supposé estimé depuis un jeu de données. Comme $Y = \{0, 1\}$, on peut remarquer que :

$$P(Y = 0|\mathbf{x}; \mathbf{w}) = 1 - P(Y = 1|\mathbf{x}; \mathbf{w})$$

ce qu'il est utile de développer analytiquement pour la suite :

$$P(1|\mathbf{x};\mathbf{w}) = \frac{\exp(\mathbf{w}^T\mathbf{x})}{1+\exp(\mathbf{w}^T\mathbf{x})} \tag{2.3}$$

$$P(0|\mathbf{x};\mathbf{w}) = 1 - \frac{\exp(\mathbf{w}^T\mathbf{x})}{1+\exp(\mathbf{w}^T\mathbf{x})} \tag{2.4}$$

$$= \frac{1+\exp(\mathbf{w}^T\mathbf{x})}{1+\exp(\mathbf{w}^T\mathbf{x})} - \frac{\exp(\mathbf{w}^T\mathbf{x})}{1+\exp(\mathbf{w}^T\mathbf{x})} \tag{2.5}$$

$$= \frac{1}{1+\exp(\mathbf{w}^T\mathbf{x})} \tag{2.6}$$

**Estimation des paramètres** Un mini jeu de données pour estimer des paramètres d'un modèle de régression logistique aura par exemple l'allure suivante :

| $y$ | $x_0$ | $x_1$ | $x_2$ |
|---|---|---|---|
| 1 | 1 | 7 | 9 |
| 1 | 1 | 6 | 8 |
| 0 | 1 | 4 | 5 |
| 0 | 1 | 3 | 4 |
| 1 | 1 | 9 | 6 |

où $y$ dénote le résultat binaire observé (comme la réussite à l'examen pour un étudiant donné), la colonne $x_0$ dénote le biais et chacune des autres colonnes $x_i$ dénotera par exemple le résultat de chaque étudiant aux tests intermédiaires. On note un jeu de données comme suit : $D = (\mathbf{x}_i, y_i)_{i=1}^N$ pour dire que le jeu de données comporte $N$ lignes qui apparient chacune un vecteur de données $\mathbf{x}_i$ et une prédiction observée $y_i$.

L'objectif de la méthode d'estimation des paramètres par **maximum de vraisemblance** consiste à trouver la valeur du vecteur $\mathbf{w}$ telle que :

$$\mathbf{w} = \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmax}} \prod_{i=1}^{N} P(y_i|\mathbf{x}_i;\mathbf{w})$$

où la fonction de vraisemblance $f(\mathbf{w}) = \prod_{i=1}^N P(y_i|\mathbf{x}_i;\mathbf{w})$ est ce qu'on appelle la **fonction objective** du problème d'optimisation. Le facteur $P(y_i|\mathbf{x}_i;\mathbf{w})$ dénote la probabilité que le modèle donne à la ligne de donnée $i$ avec une certaine valeur de paramètres $\mathbf{w}$. Cette probabilité prendra la forme $P(1|\mathbf{x}_i;\mathbf{w})$ si $y_i$ vaut 1 et la forme $P(0|\mathbf{x}_i;\mathbf{w})$ si $y_i$ vaut 0. Pour éviter

de manipuler les deux formules (2.3) et (2.6) dans ce qui suit, on utilise la formule synthétique (et équivalente) suivante :

$$P(y_i|\mathbf{x}_i;\mathbf{w}) = \frac{\exp(y_i(\mathbf{w}^T\mathbf{x}_i))}{1+\exp(\mathbf{w}^T\mathbf{x}_i)}$$

où $y_i \in \{1,0\}$ prend la valeur de la ligne qui lui correspond dans le jeu de données.

Comme la fonction de (log-)vraisemblance est une fonction strictement concave, il ne reste plus qu'à obtenir une forme analytique du gradient de cette fonction pour réaliser la maximisation par montée de gradient. Pour simplifier les calculs, on commence par utiliser la version logarithmique de la fonction objective :

$$\begin{aligned}
\hat{\mathbf{w}} &= \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmax}} \prod_{i=1}^{N} P(y_i|\mathbf{x}_i;\mathbf{w}) && (2.7)\\
&= \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmax}} \sum_{i=1}^{N} \log P(y_i|\mathbf{x}_i;\mathbf{w}) && (2.8)\\
&= \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmax}} \sum_{i=1}^{N} y_i(\mathbf{w}^T\mathbf{x}_i) - \log(1+e^{\mathbf{w}^T\mathbf{x}_i}) && (2.9)
\end{aligned}$$

Dont chacune des dérivées partielles $\frac{\partial f}{\partial w_j}(\mathbf{w})$ au point $\mathbf{w}$ s'obtient comme suit :

$$\begin{aligned}
\frac{\partial f(\mathbf{w})}{\partial w_j} &= \frac{\partial \sum_{i=1}^{N} y_i(\mathbf{w}^T\mathbf{x}_i) - \log(1+e^{\mathbf{w}^T\mathbf{x}_i})}{\partial w_j} && (2.10)\\
&= \sum_{i=1}^{N} y_i x_{ij} - \frac{\partial \log[1+e^{\mathbf{w}^T\mathbf{x}_i}]}{\partial w_j} && (2.11)\\
&= \sum_{i=1}^{N} y_i x_{ij} - \frac{\frac{\partial e^{\mathbf{w}^T\mathbf{x}_i}}{\partial w_j}}{1+e^{\mathbf{w}^T\mathbf{x}_i}} && (2.12)\\
&= \sum_{i=1}^{N} y_i x_{ij} - \frac{e^{\mathbf{w}^T\mathbf{x}_i} x_{ij}}{1+e^{\mathbf{w}^T\mathbf{x}_i}} && (2.13)\\
&= \sum_{i=1}^{N} y_i x_{ij} - [P(1|\mathbf{x}_i,\mathbf{w})\, x_{ij}] && (2.14)\\
&= \sum_{i=1}^{N} x_{ij}\,[y_i - P(1|\mathbf{x}_i,\mathbf{w})] && (2.15)
\end{aligned}$$

Autrement dit, le gradient $\nabla f(\mathbf{w})$ au point $\mathbf{w}$ correspond au vecteur $(\frac{\partial f}{\partial w_1}(\mathbf{w}) \dots \frac{\partial f}{\partial w_d}(\mathbf{w}))$ qui prend la forme suivante :

$$\nabla f(\mathbf{w}) = \sum_{i=1}^{n} \mathbf{x}_i \left[ y_i - P(1|\mathbf{x}_i, \mathbf{w}) \right]$$

Dans ce contexte l'algorithme de descente de gradient (ici de montée) prend la forme donnée en algorithme 11. Le cas de la régression logistique n'est

**Algorithm 11** Algorithme de montée de gradient (batch)

**function** BATCHGRADIENTASCENT($\alpha, (\mathbf{x}_i, y_i)_{i=1}^{N}$)
  $\mathbf{w} \leftarrow \mathbf{0}$
  **while** non convergence **do**
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \sum_{i=1}^{N} \ell(y_i, \mathbf{x}_i; \mathbf{w})$
  **end while**
  **return w**
**end function**

pas isolé. En fait dans la plupart des modèles d'apprentissage, le gradient prend la forme générale suivante :

$$\nabla f(\mathbf{w}) = \sum_{i=1}^{n} \ell(y_i, \mathbf{x}_i, \mathbf{w}) \tag{2.16}$$

où $\ell(y_i, \mathbf{x}_i, \mathbf{w})$ représente une fonction qui compare la prédiction du modèle étant donné la valeur courante de $\mathbf{w}$ pour un exemple $i$ à la valeur de référence observée dans les données $y_i$.

### 2.2.1 Descente de gradient stochastique

Dans un contexte d'apprentissage artificiel pour le TAL, la taille $N$ du jeu de données est en général considérable. De telle sorte que l'algorithme de descente (ou de montée) de gradient peut devenir un processus très coûteux en temps de par la nécessité à chaque itération de parcourir tout le jeu de données pour évaluer le gradient.

La méthode de **descente de gradient stochastique** (SGD) cherche à corriger ce problème en posant l'hypothèse simplificatrice suivante :

$$\nabla f(\mathbf{w}) \approx \ell(y_i, \mathbf{x}_i; \mathbf{w}) \qquad (i \sim \text{UNIFORM}(1, N))$$

Celle-ci consiste à approximer le gradient, en principe calculé à partir de tout le jeu de données par un gradient tiré au sort sur un seul exemple.

L'idée est qu'après un nombre suffisant d'itérations, la très grande majorité des exemples du jeu de données auront été tirés au sort. On donne la variante SGD de l'algorithme de descente[1] de gradient en algorithme 12. Vu

**Algorithm 12** Algorithme de descente de gradient stochastique

**function** StochasticGradientDescent($\alpha, (\mathbf{x}_i, y_i)_{i=1}^N$)
  $\mathbf{w} \leftarrow \mathbf{0}$
  **while** non convergence **do**
    $i \sim$ Uniform(1,N)
    $\mathbf{w} \leftarrow \mathbf{w} - \alpha\, \ell(y_i, \mathbf{x}_i; \mathbf{w})$
  **end while**
  **return w**
**end function**

que les mises à jour du gradient sont dans ce nouveau contexte beaucoup plus fréquentes, on espère que le processus global convergera plus rapidement vers une solution. Notons également, que dans la pratique, on trouve couramment la variante donnée en algorithme 13.

**Algorithm 13** Variante de l'algorithme de descente de gradient stochastique

**function** StochasticGradientDescent($\alpha, (\mathbf{x}_i, y_i)_{i=1}^N$)
  $\mathbf{w} \leftarrow \mathbf{0}$
  **while** non convergence **do**
    Shuffle($(\mathbf{x}_i, y_i)_{i=1}^N$)
    **for** $1 \leqslant i \leqslant N$ **do**
      $\mathbf{w} \leftarrow \mathbf{w} - \alpha\, \ell(y_i, \mathbf{x}_i; \mathbf{w})$
    **end for**
  **end while**
  **return w**
**end function**

Cet algorithme, très utilisé en TAL, connaît de nombreuses variantes, une des plus importantes est la variante dite en minibatch. Celle-ci consiste à sélectionner aléatoirement un nombre d'exemple $k$ ($1 < k \ll N$) pour évaluer le gradient. Cette variante est particulièrement utilisée pour paralléliser les calculs sur plusieurs processeurs. L'idée est que chaque processeur calcule

[1] Pour la régression logistique il faut utiliser la montée.

indépendamment un gradient approximé pour le minibatch qui lui est assigné. La mise à jour des poids est ensuite réalisée séquentiellement.

La force de la méthode SGD est aussi son problème principal. Le problème essentiel de la méthode c'est l'aspect aléatoire. Il est possible que certains exemples aberrants dans les données créent des estimations approximatives de gradients bruitées, ce qui peut perturber l'estimation des poids, surtout si ces exemples sont tirés en fin de procédure.

Pour corriger ce problème, on utilise couramment la version moyennée qui est appelée **descente de gradient stochastique moyennée** (ASGD). L'idée de ce dernier algorithme est de maintenir une somme cumulée des différents vecteurs $\mathbf{w}$ obtenus au cours des itérations et de renvoyer la moyenne. Ce type de méthode peut être vue comme une application de la loi des grands nombres dans le contexte de la descente de gradient (Algorithme 14).

**Algorithm 14** Algorithme de descente de gradient moyennée (ASGD)

**function** AVERAGEDSTOCHASTICGRADIENTDESCENT($\alpha, (\mathbf{x}_i, y_i)_{i=1}^N$)
  $\mathbf{w} \leftarrow \mathbf{0}$
  $\bar{\mathbf{w}} \leftarrow \mathbf{0}$
  $C \leftarrow 0$
  **while** non convergence **do**
    $i \sim$ UNIFORM(1,N)
    $\mathbf{w} \leftarrow \mathbf{w} - \alpha\, \ell(y_i, \mathbf{x}_i; \mathbf{w})$
    $\bar{\mathbf{w}} \leftarrow \bar{\mathbf{w}} + \mathbf{w}$
    $C \leftarrow C + 1$
  **end while**
  **return** $\bar{\mathbf{w}}/C$
**end function**

### 2.2.2 Instabilités numériques

$\exp(x)$ peut créer un overflow si $x$ trop grand. Or on a l'équivalence suivante

$$\frac{e^x}{1+e^x} = \frac{e^x}{1+e^x}\frac{e^{-x}}{e^{-x}} \tag{2.17}$$

$$= \frac{1}{(1+e^x)e^{-x}} \tag{2.18}$$

$$= \frac{1}{e^{-x}+1} \tag{2.19}$$

Utiliser (2.19) lorsque $x$ est positif permet d'éviter l'overflow. Utiliser $\frac{e^x}{1+e^x}$ lorsque $x$ est négatif permet d'éviter l'overflow (dans l'autre sens).

## 2.3 Régression logistique multinomiale

Le modèle de régression logistique multinomiale est une généralisation du modèle de régression logistique au cas où $Y$ est un ensemble de deux valeurs discrètes ou plus. Un tel modèle prédit une probabilité $P(y|\mathbf{x};\mathbf{w})$ pour chaque classe $y \in Y$ étant donné un vecteur d'observations $\mathbf{x} \in \mathbb{R}^d$ et une matrice de poids $\mathbf{w} \in \mathbb{R}^{d \times |Y|}$.

$$\begin{bmatrix} z_{y_1} \\ z_{y_2} \\ z_{y_3} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

Figure 2.3: Représentation schématique d'un modèle de régression logistique multinomiale ($\mathbf{w} \in \mathbb{R}^{5\times3}$, $\mathbf{x} \in \mathbb{R}^5$)

En première approximation, on peut considérer qu' un vecteur de poids $\mathbf{w}_y \in \mathbb{R}^d$ (une ligne de la matrice) correspond à chaque classe $y \in Y$. Le score $z_y$ de la classe $y$ se calcule par le produit scalaire $\mathbf{w}^T\mathbf{x}$. On peut ainsi calculer le score de l'ensemble des classes par le produit de la matrice $\mathbf{w}$ et du vecteur $\mathbf{x}$, ce qu'on illustre schématiquement en figure 2.3.

Pour obtenir des probabilités à partir du vecteur de scores – positifs ou négatifs – on les transforme d'abord en nombres positifs à l'aide de la fonction exponentielle (exp) puis en divisant chacun de ces nombres par le total des scores ($\sum_y z_y$), ce qui donne le modèle de régression logistique

multinomiale[2] :

$$P(y|\mathbf{x};\mathbf{w}) = \frac{\exp(\mathbf{w}_y^T\mathbf{x})}{\sum_{y'\in Y}\exp(\mathbf{w}_{y'}^T\mathbf{x})} \tag{2.20}$$

**Codage des symboles par des vecteurs creux** Pour utiliser un tel modèle en TAL, il reste un problème à traiter : le modèle suppose naturellement des données $\mathbf{x}$ réelles alors que les données langagières sont le plus souvent représentées par des symboles discrets (à l'exception des données audio).

Supposons que $\mathbf{x}$ est un vecteur de symboles discrets de dimension $k$ comme par exemple les familles du jeu de cartes : $\{\diamondsuit, \spadesuit, \heartsuit, \clubsuit\}$. La manière classique de coder numériquement ce type de symboles consiste à les coder sur des vecteurs à $|F|$ dimensions, tel que chaque symbole value à 1 une dimension et à 0 les autres. Par exemple pour les familles du jeu de cartes :

$$\begin{aligned}\diamondsuit &= [1,0,0,0]\\ \spadesuit &= [0,1,0,0]\\ \heartsuit &= [0,0,1,0]\\ \clubsuit &= [0,0,0,1]\end{aligned}$$

On peut formaliser cette méthode de codage, appelée **one hot coding**, à l'aide de **fonctions features**. À chaque dimension $i$ du vecteur codé $\mathbf{\Phi}(x) = \phi_1(x), \phi_2(x), \phi_3(x), \phi_4(x)$ on fait correspondre une fonction feature $\phi_i(x)$ à valuation booléenne, par exemple si on a tiré une carte $x$, on pourra la coder à l'aide des features suivantes :

$$\phi_1(x) = \begin{cases} 1 & \text{si } x = \diamondsuit \\ 0 & \text{sinon}\end{cases} \quad \phi_2(x) = \begin{cases} 1 & \text{si } x = \spadesuit \\ 0 & \text{sinon}\end{cases} \quad \phi_3(x) = \begin{cases} 1 & \text{si } x = \heartsuit \\ 0 & \text{sinon}\end{cases} \quad \phi_4(x) = \begin{cases} 1 & \text{si } x = \clubsuit \\ 0 & \text{sinon}\end{cases}$$

Supposons maintenant qu'on ait tiré un vecteur $\mathbf{x} = x_1, x_2$ de deux cartes. On peut toujours utiliser un vecteur de features $\mathbf{\Phi}(\mathbf{x})$ pour coder la séquence $\mathbf{x}$ de cartes tirées :

$$\phi_1(\mathbf{x}) = \begin{cases} 1 & \text{si } x_1 = \diamondsuit \\ 0 & \text{sinon}\end{cases} \quad \phi_2(\mathbf{x}) = \begin{cases} 1 & \text{si } x_1 = \spadesuit \\ 0 & \text{sinon}\end{cases} \quad \phi_3(\mathbf{x}) = \begin{cases} 1 & \text{si } x_1 = \heartsuit \\ 0 & \text{sinon}\end{cases} \quad \phi_4(\mathbf{x}) = \begin{cases} 1 & \text{si } x_1 = \clubsuit \\ 0 & \text{sinon}\end{cases}$$

$$\phi_5(\mathbf{x}) = \begin{cases} 1 & \text{si } x_2 = \diamondsuit \\ 0 & \text{sinon}\end{cases} \quad \phi_6(\mathbf{x}) = \begin{cases} 1 & \text{si } x_2 = \spadesuit \\ 0 & \text{sinon}\end{cases} \quad \phi_7(\mathbf{x}) = \begin{cases} 1 & \text{si } x_2 = \heartsuit \\ 0 & \text{sinon}\end{cases} \quad \phi_8(\mathbf{x}) = \begin{cases} 1 & \text{si } x_2 = \clubsuit \\ 0 & \text{sinon}\end{cases}$$

[2]Cette fonction de normalisation de scores en probabilités est également appelée fonction softmax.

mais on peut aussi envisager coder les information qui portent sur les **interactions** entre variables. Si on veut indiquer explicitement qu'on a tiré deux cartes de la famille pique, on pourra utiliser par exemple une feature telle que :

$$\phi_9(\mathbf{x}) = \begin{cases} 1 & \text{si } x_1 = \spadesuit \quad \& \quad x_2 = \spadesuit \\ 0 & \text{sinon} \end{cases}$$

de telle sorte que les vecteurs de features $\mathbf{\Phi}(\mathbf{x})$ ont généralement une dimensionnalité considérable en pratique. Notons que cette première méthode de codage permet d'utiliser le modèle logistique multinomial avec des symboles discrets comme données. En explicitant le codage, le modèle (2.20) prend ainsi la forme suivante :

$$P(y|\mathbf{x};\mathbf{w}) = \frac{\exp(\mathbf{w}_y^T \mathbf{\Phi}(\mathbf{x}))}{\sum_{y' \in Y} \exp(\mathbf{w}_{y'}^T \mathbf{\Phi}(\mathbf{x}))} \tag{2.21}$$

Un tel modèle fait encore l'hypothèse implicite que les poids sont organisés en matrice : chaque ligne $\mathbf{w}_y$ de la matrice de poids correspond aux poids destinés à scorer la classe $y \in Y$. Lorsque le nombre de classes est grand (voire infini) ou lorsqu'on ne souhaite pas utiliser une matrice de poids, on peut généraliser la méthode de codage par interactions en spécialisant les features à une classe donnée, c'est-à-dire en spécifiant des features qui ont la forme générale suivante :

$$\phi_i(\mathbf{x}, y) = \begin{cases} 1 & \text{si } Y = y \quad \& \quad x_k = \text{valeur} \quad (\& \, x_l = \text{valeur}) * \\ 0 & \text{sinon} \end{cases}$$

Cette représentation alternative – qui est le plus souvent utilisée dans ce cours – permet de représenter les poids comme un simple vecteur $\mathbf{w}$. Le score d'une classe est obtenu par un produit scalaire $z_y = \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}, y)$ qui ne fait intervenir que les features qui sont pertinentes pour scorer cette classe. Lorsqu'on utilise ce codage, on notera que le modèle (2.21) prend alors la forme suivante :

$$P(y|\mathbf{x};\mathbf{w}) = \frac{\exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}, y))}{\sum_{y' \in Y} \exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}, y'))} \tag{2.22}$$

**Estimation des paramètres** L'entraînement d'un modèle de régression logistique multinomiale est la procédure qui consiste à estimer un vecteur $\mathbf{w}$ de poids à partir de données annotées. Un jeu de données aura en général une allure du type :

| $y$ | $x_0$ | $x_1$ | $x_2$ |
|---|---|---|---|
| Animé | la | souris | grise |
| Inanimé | la | souris | apple |
| Inanimé | une | souris | cassée |
| Animé | une | souris | affamée |
| Groupe | un | troupeau | errant |

Cet exemple pourrait constituer le début d'un jeu de données destiné à classer les occurrences de noms en classes d'animacité ($Y = \{$Animé,Inanimé,Groupe$\}$) en fonction de leur contexte.

Dans ce qui suit, on note le jeu de données $(\mathbf{x}_i, y_i)_{i=1}^N$, et comme dans le cas de la régression logistique, l'objectif est de trouver une valeur des paramètres qui maximise l'objectif de maximum de vraisemblance :

$$f(\mathbf{w}) = \prod_{i=1}^{N} P(y_i|\mathbf{x}_i; \mathbf{w}) \tag{2.23}$$

La fonction de log-vraisemblance est en général celle qui est maximisée car elle est strictement concave et la version logarithmique facilite les calculs :

$$\begin{aligned}
\hat{\mathbf{w}} &= \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmax}} \prod_{i=1}^{N} P(y_i|\mathbf{x}_i; \mathbf{w}) \\
&= \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmax}} \sum_{i=1}^{N} \log P(y_i|\mathbf{x}_i; \mathbf{w}) \\
&= \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmax}} \sum_{i=1}^{N} \log \frac{\exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y_i))}{\sum_{y'\in Y} \exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y'))} \\
&= \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmax}} \sum_{i=1}^{N} \left[ \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y_i) - \log \left( \sum_{y'\in Y} \exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y')) \right) \right]
\end{aligned} \tag{2.24}$$

On obtient les dérivées partielles de manière analogue au cas de la régression

logistique :

$$\begin{aligned}
\frac{\partial f(\mathbf{w})}{\partial w_j} &= \frac{\partial f}{\partial w_j} \sum_{i=1}^{N} \left[ \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y_i) - \log \left( \sum_{y' \in Y} \exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y')) \right) \right] \\
&= \sum_{i=1}^{N} \left[ \phi_j(\mathbf{x}_i, y_i) - \frac{\partial f}{\partial w_j} \log \left( \sum_{y' \in Y} \exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y')) \right) \right] \\
&= \sum_{i=1}^{N} \left[ \phi_j(\mathbf{x}_i, y_i) - \frac{\frac{\partial f}{\partial w_j} \sum_{y' \in Y} \exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y'))}{\sum_{y' \in Y} \exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y'))} \right] \\
&= \sum_{i=1}^{N} \left[ \phi_j(\mathbf{x}_i, y_i) - \frac{\sum_{y' \in Y} \exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y')) \phi_j(\mathbf{x}_i, y')}{\sum_{y' \in Y} \exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y'))} \right] \qquad (2.25) \\
&= \sum_{i=1}^{N} (\phi_j(\mathbf{x}_i, y_i) - \sum_{y' \in Y} P(y' | \mathbf{x}_i; \mathbf{w}) \phi_j(\mathbf{x}_i, y')) \qquad (2.26)
\end{aligned}$$

La dérivation qui précède applique essentiellement des règles de dérivation classiques (log et exp). Par contre il faut voir que le facteur $\phi_j(\mathbf{x}, y')$ qui apparaît en (2.25) n'est valué à 1 que pour le seul $y'$ où la feature est effectivement valuée à 1. Intuitivement, la mise à jour consiste à donner un bonus à $\phi_j$ lorsqu'elle contribue à prédire la bonne hypothèse et un malus lorsqu'elle contribue à prédire une mauvaise hypothèse. Le bonus sera d'autant plus important que le modèle aura mal prédit la bonne solution, et le malus est proportionnel à la probabilité que le modèle donne à cette mauvais hypothèse.

Autrement dit, la montée de gradient consiste à compter le nombre d'occurrences de la feature observées dans les données (premier terme) et à lui soustraire un pseudo-compte d'occurrences de la feature tel que prédit par le modèle (second terme). Le gradient sera nul lorsque ces deux nombres seront égaux pour toutes les features (et la procédure d'estimation aura convergé).

Pour le calcul de l'ensemble du gradient, on peut ainsi formuler une contrepartie de la fonction $\ell(y_i, \mathbf{x}_i; \mathbf{w})$, donnée en équation 2.16, pour le cas multinomial :

$$\ell(y_i, \mathbf{x}_i, \mathbf{w}) = \mathbf{\Phi}(\mathbf{x}_i, y) - \sum_{y' \in Y} P(y' | \mathbf{x}_i; \mathbf{w}) \mathbf{\Phi}(\mathbf{x}_i, y')$$

On peut ainsi réutiliser des algorithmes de montée de gradient standard ou stochastiques pour réaliser l'entrainement. On donne un exemple d'algorithme de montée de gradient en batch en Algorithme 15.

**Algorithm 15** Algorithme de montée de gradient pour la régression logistique multinomiale

**function** MULTINOMIALLOGISTICGRADIENTASCENT($\alpha, (\mathbf{x}_i, y_i)_{i=1}^N$)
  $\mathbf{w} \leftarrow \mathbf{0}$
  **while** non convergence **do**
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left( \sum_{i=1}^N \mathbf{\Phi}(\mathbf{x}_i, y_i) - \sum_{y' \in Y} P(y'|\mathbf{x}_i; \mathbf{w}) \mathbf{\Phi}(\mathbf{x}_i, y') \right)$
  **end while**
  **return w**
**end function**

**Exercice 2.2 (Régularisation)** *On utilise parfois une version régularisée du modèle logistique multinomial. La fonction objective de ce modèle s'exprime comme suit :*

$$f(\mathbf{w}) = \frac{\lambda}{2} ||\mathbf{w}||_2^2 + \sum_{i=1}^N \log P(y_i|\mathbf{x}_i; \mathbf{w}) \tag{2.27}$$

*où $\lambda \in \mathbb{R}^+$ est un paramètre fixé par l'utilisateur. Donner une explication intuitive de l'intérêt que peut avoir le terme de régularisation. Donner le gradient de la version régularisée du modèle logistique (pas besoin de refaire tous les calculs). Quel effet a $\lambda$ ?*

**Exercice 2.3 (SGD)** *Donner une reformulation de l'algorithme 15 qui utilise* SGD

**Exercice 2.4 (SGD)** *Donner une reformulation de l'algorithme 15 pour la fonction objective régularisée (2.27) et en utilisant* SGD.

## 2.4 Large marge multiclasse

Si les modèles logistiques cherchent à maximiser la probabilité que le modèle donne aux données, les modèles à large marge cherchent à produire un modèle qui non seulement minimise les erreurs sur les données d'entrainement mais également essaye de donner une marge de confiance aux décisions qu'il va prendre. Deux modèles très connus font partie de cette famille : l'algorithme du perceptron et le modèle des machines à vecteurs support linéaires (SVM).

Les modèles à large marge, présentés ici, sont des modèles linéaires, chaque classe $y \in Y$ est associée à un vecteur de poids qui lui correspond et

le score d'une classe est déterminé par un produit scalaire :

$$\text{score}(y, \mathbf{x}; \mathbf{w}) = \mathbf{w}_y^T \, \mathbf{\Phi}(\mathbf{x}) \tag{2.28}$$

On peut utiliser les modèles à large marge avec le même système de codage que les modèles logistiques. On utilisera ici la version du modèle avec le codage en $\phi(\mathbf{x}, y)$, c'est-à-dire :

$$\text{score}(y, \mathbf{x}; \mathbf{w}) = \mathbf{w}^T \, \mathbf{\Phi}(\mathbf{x}, y) \tag{2.29}$$

Contrairement aux modèles logistiques, les modèles à large marge produisent des scores qui couvrent toute l'intervalle réelle. La procédure de décision sous-jacente consiste à choisir la classe de score maximal. Donc plus un score est élevé, plus la classe a de chances d'être choisie par la procédure de décision.

On voit donc que les modèles à large marge ont un fonctionnement général qui est très similaire aux modèles logistiques. La différence principale provient du fait que les prédictions ne s'interprètent pas comme des probabilités.

**Estimation des paramètres** L'originalité des modèles à large marge tient dans le procédé d'apprentissage des poids. Les données utilisées ont la même allure que les données utilisées par les modèles logistiques. Ainsi on note un jeu de données $D = (\mathbf{x}_i, y_i)_{i=1}^N$.

La fonction objective d'un modèle à large marge multiclasse est la suivante :

$$f(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \max(0, 1 + \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y_i^*) - \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y_i))) \tag{2.30}$$

où $y^* = \text{argmax}_{y' \in Y \setminus \{y\}} \, \mathbf{w}^T \, \mathbf{\Phi}(\mathbf{x}, y')$ représente l'hypothèse incorrecte qui a le meilleur le score (le plus sérieux concurrent de l'hypothèse correcte). La fonction $\max(0, 1 + \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y_i^*) - \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y_i)))$ est appelée **hinge loss multiclasse**, on peut remarquer que plus le score du plus sérieux concurrent est élevé plus la valeur de la fonction de coût augmente. Par conséquent, l'objectif d'un modèle à large marge est de minimiser la fonction objective (on utilisera cette fois l'algorithme de descente de gradient). Pour le dire encore autrement, il faut voir que pour un exemple donné, le coût augmente lorsque :

$$\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y_i) - \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y_i^*) \leqslant 1$$

c'est-à-dire tant que le score de la prédiction de référence n'est pas supérieur à la somme score du meilleur concurrent avec une marge de 1. La marge a en général la valeur conventionnelle 1. Certains modèles utilisent une marge nulle, comme typiquement l'algorithme du perceptron.

**Notion de sous-gradient d'une fonction convexe** La procédure d'estimation des paramètres consiste à minimiser (2.30). Bien que convexe, cette fonction n'est pas différentiable sur l'intégralité de son domaine car elle fait intervenir un maximum. On illustre cet aspect en figure 2.4 où la fonction $f(x) = \max(0, 1 - x)$ n'est pas différenciable en $x_0 = 1$ : visuellement on peut constater qu'un nombre infini de droites sont tangentes à la fonction en $x_0 = 1$.

On appelle **sous-dérivée** d'une fonction convexe $f(x)$ en $x_0$ tout nombre réel $s \in \mathbb{R}$ tel que :

$$f(x) \geqslant f(x_0) + s(x - x_0)$$

intuitivement $s$ est l'ensemble des coefficients de pente des droites qui passent en $(x_0, f(x_0))$ et qui sont situées sous la fonction $f(x)$. Sur l'exemple en figure 2.4, $s \in [-1, 0]$. L'ensemble $S$ de toutes les sous-dérivées de $f(x)$ en $x_0$ est appelé **sous-différentiel** de $f$ en $x_0$. Le calcul des bornes de $S$ est donné dans le cas de la fonction max par les dérivées des fonctions linéaires $f(x) = 0$ et $f(x) = 1 - x$.

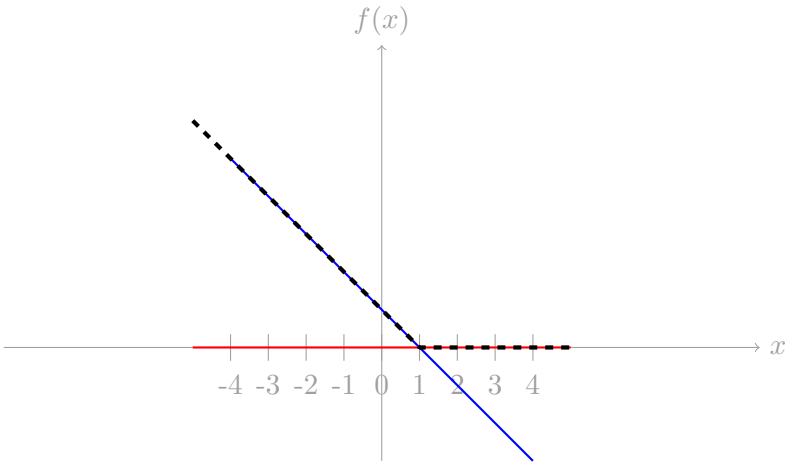



Figure 2.4: Maximum de deux fonctions $f(x) = \max(0, 1 - x)$

De manière générale la dérivée d'une fonction $f(x) = \max(f_1(x), f_2(x))$

s'obtient en divisant le domaine en deux sous-intervalles réparties autour du point de non différentiabilité ($x_0 = 1$ sur l'exemple). On utilisera la dérivée correspondant à l'intervalle dans lequel on se trouve. On peut déterminer facilement cet intervalle en comparant $f_1$ et $f_2$ de telle sorte que :

$$\frac{\partial f}{\partial x}\max(f_1(x), f_2(x)) = \begin{cases} \frac{\partial f_1}{\partial x}(x) & \text{si } f_1(x) > f_2(x) \\ \\ \frac{\partial f_2}{\partial x}(x) & \text{si } f_2(x) > f_1(x) \end{cases}$$

Lors de la descente de sous-gradient, au point non différentiable, on peut choisir tout $s \in S$ comme valeur de la dérivée. En pratique on prendra arbitrairement la dérivée de $f_1$ ou de $f_2$.

Ce qui précède se généralise aux cas des fonctions de plusieurs variables. En ce qui concerne la fonction objective des modèles à large marge (2.30), on obtient les dérivées partielles suivantes :

$$\hat{\mathbf{w}} = \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmin}} \sum_{i=1}^{N} \max(0, 1 + \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i^*) - \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i))) \tag{2.31}$$

$$\begin{aligned} \frac{\partial f}{\partial w_j} &= \sum_{i=1}^{N} \frac{\partial}{\partial w_j}\max(0, 1 + \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i^*) - \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i))) \\ &= \sum_{i=1}^{N} \begin{cases} \phi_j(\mathbf{x}_i, y*) - \phi_j(\mathbf{x}_i, y_i) & \text{si } 1 + \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i^*) - \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i) \geqslant 0 \\ 0 & \text{sinon} \end{cases} \\ &= \sum_{i=1}^{N} \begin{cases} \phi_j(\mathbf{x}_i, y*) - \phi_j(\mathbf{x}_i, y_i) & \text{si } \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i) - \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i^*) \leqslant 1 \\ 0 & \text{sinon} \end{cases} \end{aligned} \tag{2.32}$$

Au final, on peut formuler une fonction $\ell$ qui représente la contribution de chaque exemple dans les données au gradient :

$$\ell(y_i, \mathbf{x}_i, \mathbf{w}) = \begin{cases} \mathbf{\Phi}(\mathbf{x}_i, y_i^*) - \mathbf{\Phi}(\mathbf{x}_i, y_i) & \text{si } [\text{score}(y_i) - \text{score}(y_i^*)] \leqslant 1 \\ \mathbf{0} & \text{sinon} \end{cases}$$

L'algorithme de descente de gradient s'instancie de manière habituelle, ce qui donne la version présentée en algorithme 16. On peut noter qu'on a procédé à une légère réorganisation des signes, ce qui facilite notamment la comparaison avec les algorithmes d'optimisation pour le cas logistique. On peut remarquer que l'algorithme d'optimisation met à jour les poids uniquement lorsque la classe correcte n'est pas prédite avec une marge supérieure à 1 sur sa plus sérieuse concurrente. Dans ce cas l'algorithme donne un bonus au poids associés aux features de la référence et un malus aux poids associés aux features de la prédiction concurrente.

**Algorithm 16** Descente de gradient pour modèle à large marge linéaire

**function** BATCHLARGEMARGE($\alpha, (\mathbf{x}_i, y_i)_{i=1}^N$)
  $\mathbf{w} \leftarrow \mathbf{0}$
  **while** non convergence **do**
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left( \frac{1}{N} \sum_{i=1}^N \begin{cases} \mathbf{\Phi}(\mathbf{x}_i, y_i) - \mathbf{\Phi}(\mathbf{x}_i, y_i^*) & \text{si } [\text{score}(y_i) - \text{score}(y_i^*)] \leqslant 1 \\ \mathbf{0} & \text{sinon} \end{cases} \right)$
  **end while**
  **return w**
**end function**

L'algorithme de descente de gradient à large marge peut être utilisé tel quel, mais on va maintenant montrer que deux algorithmes très connus peuvent être vus comme des cas particuliers de la famille des modèles à large marge : l'algorithme du perceptron et les SVM linéaires.

### 2.4.1 Algorithme du perceptron

L'algorithme du **perceptron** est un algorithme très populaire en TAL car il est particulièrement simple à implémenter. On peut le voir comme un cas particulier des modèles à large marge dont la procédure d'optimisation est SGD. Ainsi en posant que la marge est nulle, l'objectif à large marge devient :

$$f(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \max(0, \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y_i^*) - \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, y_i))) \tag{2.33}$$

et le sous-gradient devient :

$$\frac{\partial f(\mathbf{w})}{\partial w_j} = \sum_{i=1}^{N} \begin{cases} \phi_j(\mathbf{x}_i, y_i^*) - \phi_j(\mathbf{x}_i, y_i) & \text{si } [\text{score}(y_i) - \text{score}(y_i^*)] \leqslant 0 \\ 0 & \text{sinon} \end{cases} \tag{2.34}$$

Si l'algorithme utilisé pour réaliser la descente de gradient est SGD structuré en itérations sur les données (Algorithme 13), on peut observer que ce cas d'utilisation correspond exactement à l'algorithme du **perceptron multiclasse** traditionnel que l'on reproduit en algorithme 17.

**Exercice 2.5 (Vérification)** *Pour vous convaincre que l'algorithme 17 est bien une variante notationnelle d'un modèle à large marge. Ecrivez l'algorithme de descente de gradient stochastique qui utilise directement (2.34) et procédez à la comparaison.*

**Algorithm 17** Algorithme du perceptron

```
function PERCEPTRON(α, (x_i, y_i)_{i=1}^N)
    w ← 0
    while non convergence do
        SHUFFLE((x_i, y_i)_{i=1}^N)
        for 1 ⩽ i ⩽ N do
            ŷ ← argmax_{y∈Y} w^T Φ(x_i, y)
            if ŷ ≠ y_i then
                w ← w + α (φ_j(x_i, y_i) − φ_j(x_i, ŷ))
            end if
        end for
    end while
    return w
end function
```

Notons finalement que l'algorithme du perceptron est très utilisé dans sa version moyennée ce qui permet de lui donner une certaine stabilité et de réduire les effets de surentrainement (Algorithme 18)

### 2.4.2 SVM linéaire multiclasse

Si maintenant on ajoute un terme de régularisation à l'objectif (2.30), celui-ci prend la forme suivante :

$$f(\mathbf{w}) = \frac{\lambda}{2}||\mathbf{w}||_2^2 + \frac{1}{N}\sum_{i=1}^{N}\max(0, 1 + \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i^*) - \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i))) \quad (2.35)$$

Cet objectif correspond à celui d'une **machine à vecteurs de support** (SVM) linéaire. On peut indiquer que les modèles à large marge ont pour origine les travaux sur SVM, même si ceux-ci ont été initialement formulés de manière très différente (en apparence).

On peut optimiser cet objectif avec les algorithmes de descente de gradient et de descente de gradient stochastique moyennée présentés dans ce chapitre. Ainsi le problème d'optimisation consiste à minimiser :

$$\hat{\mathbf{w}} = \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmin}} \frac{\lambda}{2}||\mathbf{w}||_2^2 + \sum_{i=1}^{N}\max(0, 1 + \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i^*) - \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i))) \quad (2.36)$$

dont on obtient les dérivées partielles suivantes en procédant à un développement

**Algorithm 18** Algorithme du perceptron moyenné

**function** AVERAGEDPERCEPTRON($\alpha, (\mathbf{x}_i, y_i)_{i=1}^N$)
  $\mathbf{w} \leftarrow \mathbf{0}$
  $\bar{\mathbf{w}} \leftarrow \mathbf{0}$
  $e \leftarrow 0$
  **while** non convergence **do**
    SHUFFLE($(\mathbf{x}_i, y_i)_{i=1}^N$)
    **for** $1 \leqslant i \leqslant N$ **do**
      $\hat{y} \leftarrow \text{argmax}_{y \in Y}\, \mathbf{w}^T\, \mathbf{\Phi}(\mathbf{x}_i, y)$
      **if** $\hat{y} \neq y_i$ **then**
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\left(\phi_j(\mathbf{x}_i, y_i) - \phi_j(\mathbf{x}_i, \hat{y})\right)$
      **end if**
      $\bar{\mathbf{w}} \leftarrow \bar{\mathbf{w}} + \mathbf{w}$
    **end for**
    $e \leftarrow e + 1$
  **end while**
  **return** $\bar{\mathbf{w}}/(N \times e)$
**end function**

**Algorithm 19** Descente de gradient de SVM linéaire (batch)

**function** BATCHSVM($\alpha, \lambda, (\mathbf{x}_i, y_i)_{i=1}^N$)
  $\mathbf{w} \leftarrow \mathbf{0}$
  **while** non convergence **do**
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\left(\frac{1}{N}\sum_{i=1}^N \begin{cases} \mathbf{\Phi}(\mathbf{x}_i, y_i) - \mathbf{\Phi}(\mathbf{x}_i, y_i^*) & \text{si } [\text{score}(y_i) - \text{score}(y_i^*)] \leqslant 1 \\ \mathbf{0} & \text{sinon} \end{cases} - \lambda\mathbf{w}\right)$
  **end while**
  **return w**
**end function**

analogue aux précédents :

$$\frac{\partial f}{\partial w_j} = \lambda w_j + \sum_{i=1}^{N} \begin{cases} \phi_j(\mathbf{x}_i, y*) - \phi_j(\mathbf{x}_i, y_i) & \text{si } \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i) - \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i^*) \leqslant 1 \\ 0 & \text{sinon} \end{cases} \tag{2.37}$$

Contrairement au perceptron, la résolution du problème d'optimisation de SVM en forme primale est habituellement présenté comme un algorithme de descente de gradient en batch (Algorithme 19). Mais rappelons que SVM est habituellement présenté à partir de sa forme duale. Celle-ci est toutefois inadaptée aux problèmes de TAL.

| MODÈLE | OBJECTIF | GRADIENT |
|---|---|---|
| Régression softmax | $\prod_{i=1}^{N} P(y_i\|\mathbf{x};\mathbf{w})$ | $\sum_{i=1}^{N} \phi_j(\mathbf{x_i}, y_i) - \sum_{y' \in Y} P(y'\|\mathbf{x_i}, \mathbf{w})\phi_j(\mathbf{x_i}, y_i)$ |
| Large marge | $\sum_{i=1}^{N} \max(0, 1 + \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i^*) - \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i)))$ | $\sum_{i=1}^{N} \begin{cases} \phi_j(\mathbf{x}_i, y*) - \phi_j(\mathbf{x}_i, y_i) & \text{si } \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i) - \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i, y_i^*) \leqslant 1 \\ 0 & \text{sinon} \end{cases}$ |

## 2.5 Réseaux de neurones

On propose d'introduire l'usage de réseaux de neurones en TAL comme une généralisation des modèles de régression logistique multinomiale.

La difficulté pratique quand on utilise un modèle de régression logistique multinomiale consiste à définir les fonctions features et notamment leurs interactions. Deux problèmes se posent :

- Le nombre de features et notamment d'interactions est en général considérable **(feature engineering)**
- Quelles features (interactions) inclure dans le modèle ? **(feature selection)**

En TAL on utilise des réseaux de neurones pour construire de meilleures représentations numériques de symboles discrets, comme par exemple des représentations des mots appelées plongements lexicaux ou *word embeddings*).

### 2.5.1 Réseau à Propagation avant

On peut voir le modèle de régression logistique multinomiale (ou modèle de régression softmax) tel que présenté en équation (2.20) et réécrite ici sous forme plus compacte à l'aide de la notation softmax :

$$\mathbf{y} = \text{softmax}(\mathbf{W}_1\mathbf{x}) \tag{2.38}$$

Pour exprimer une contrepartie aux interactions et à la réduction de dimensionnalité (feature selection) des modèles traditionnels, un réseau de neurones à propagation avant introduit une couche cachée $\mathbf{h}$ supplémentaire à (2.20).

$$\mathbf{h} = g(\mathbf{W}_2\mathbf{x}) \tag{2.39}$$
$$\mathbf{y} = \text{softmax}(\mathbf{W}_1\mathbf{h}) \tag{2.40}$$

où $g$ dénote une fonction non linéaire de la forme $\mathbb{R}^d \mapsto \mathbb{R}^d$ dite fonction d'activation. Celle-ci est typiquement choisie parmi la fonction logistique, la fonction tangente hyperbolique ou la fonction relu (Figure 2.5).

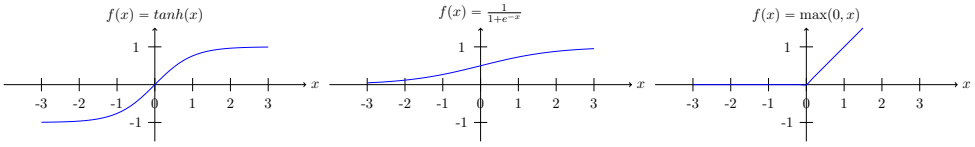



Figure 2.5: Fonctions d'activation

On illustre en figure 2.6 l'allure schématique d'un réseau de ce type. En général la couche cachée a une dimension qui est choisie plus petite que la dimension du vecteur d'entrée de telle sorte qu'elle encode une représentation des données en dimension réduite.

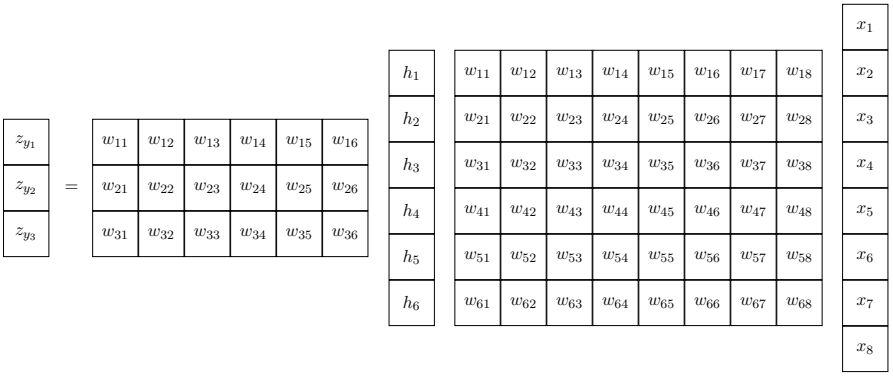



Figure 2.6: $\mathbf{y} = \text{softmax}(\mathbf{W}_1 g(\mathbf{W}_2\mathbf{x}))$

**Estimation des paramètres** L'estimation des paramètres consiste généralement à maximiser la vraisemblance (on parle également de minimisation de l'entropie croisée) d'un jeu de données. La méthode d'optimisation utilisée est en général la descente de gradient stochastique ( SGD ). Si le principe reste le même que pour les cas précédents, le calcul du gradient à chaque itération est plus complexe. Il est réalisé à l'aide d'un algorithme appelé rétropropagation du gradient. Par ailleurs la fonction objective d'un réseau de neurone n'est en général pas convexe. Il existe en général un grand nombre de minima locaux et il y a peu de garanties de trouver un minimum global.

TODO (backprop) `http://colah.github.io/posts/2015-08-Backprop/`

Notons pour terminer que l'utilisation d'une couche de sortie softmax n'a rien d'obligatoire, on peut utiliser des couches de sorties à large marge ou du perceptron. Il reste que la couche de sortie softmax est en pratique très utilisée.

**Plongements lexicaux** Reste la question de l'interface d'un réseau de neurones avec des données langagières. On a implicitement supposé que le vecteur de données $\mathbf{x}$ est un vecteur de réels. Or en TAL les données sont des symboles discrets (des mots).

L'interface est réalisée par une couche de dictionnaire (dite parfois couche de lookup ou couche de plongement) qui consiste à encoder un dictionnaire qui envoie des mots sur des vecteurs.

Dans ce contexte, on suppose que les données du réseau sont des symboles codés par la méthode one-hot. La couche de plongement est alors une simple matrice $\mathbf{E}$ dont la multiplication avec chaque vecteur one-hot renvoie l'embedding $\mathbf{e}$ correspondant. Le vecteur d'entrée $\mathbf{x}$ est la concaténation des vecteurs d'embeddings correspondant à chacun de ces symboles, comme illustré en Figure 2.7.

Comme la matrice $\mathbf{E}$ est une matrice de paramètres, ceux-ci sont mis à jour, au même titre que tous les autres paramètres, lors de la descente de gradient.

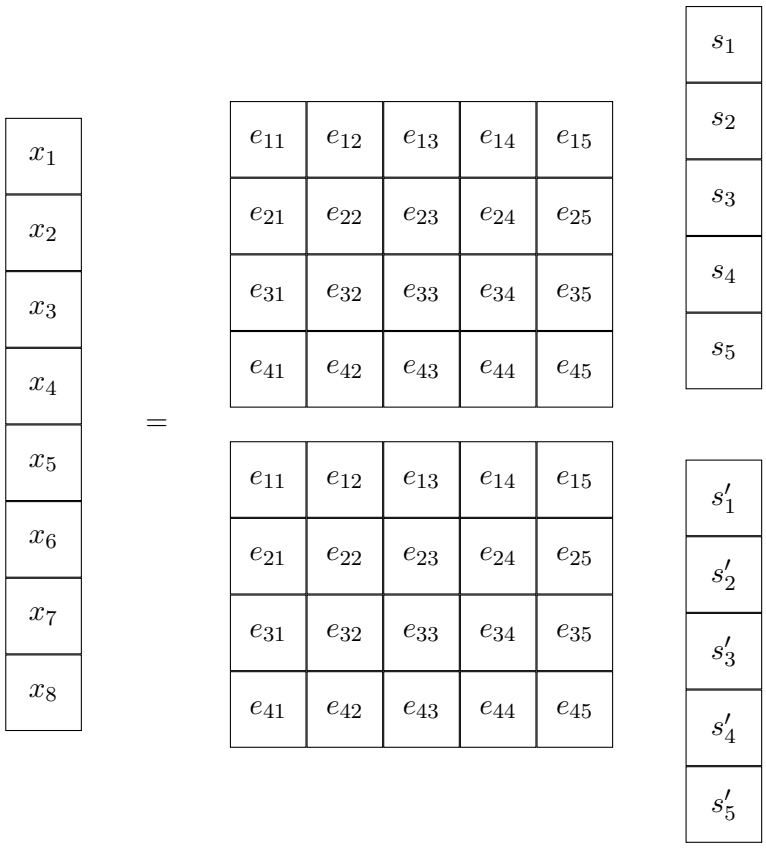



Figure 2.7: Couche de plongements

# Chapter 3

# Perceptron structuré

Le modèle du perceptron structuré est initialement dû à [**?**].

## 3.1 Généralisation du modèle du perceptron

Le perceptron structuré est un modèle qui généralise le modèle du perceptron au cas des données structurées. Dans ce chapitre, on donne une présentation pour la modélisation de séquences. On généralisera ce cas à la prédiction d'arbres dans les chapitres suivants.

Un perceptron multiclasse est un modèle statistique qui prédit le score $\psi(y)$ d'une donnée $y \in Y$ de manière linéaire :

$$\psi(y) = \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}, y) \tag{3.1}$$

Comme $\mathbf{w} \in \mathbb{R}^d$ et $\mathbf{\Phi}(\mathbf{x}, y) \in \{0,1\}^d$, on a que $\psi(y) \in \mathbb{R}$. Ce modèle est habituellement utilisé dans un contexte de prise de décision, il s'agit de sélectionner parmi un ensemble $Y$ d'hypothèses, l'hypothèse $y \in Y$ de score maximal :

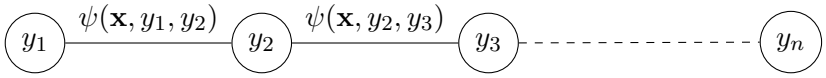
$$\hat{y} = \underset{y \in Y}{\operatorname{argmax}}\, \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}, y) \tag{3.2}$$

Dans le cas où $Y$ est un ensemble de structures, comme un ensemble de séquences, la taille de cet ensemble croît exponentiellement en fonction de la longueur de la phrase. L'idée d'un modèle de perceptron structuré est de permettre la décomposition du calcul du score des séquences de telle sorte qu'il soit possible de partager des sous-calculs entre des sous-séquences communes. Ainsi une séquence de tags $\mathbf{y} = y_1, y_2 \dots y_m \dots$ sera évaluée par

la fonction de score suivante :

$$\Psi(\mathbf{y}) = \sum_{i=1}^{m} \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}, y_{i-1}, y_i) \tag{3.3}$$

Autrement dit, le score d'une séquence est la somme des scores $\psi_i(\mathbf{x}, y_{i-1}, y_i) = \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}, y_{i-1}, y_i)$ attribués à tous les couples de tags qui la composent. Lorsqu'il s'agit de donner un score à un ensemble de séquences, il est ainsi possible de représenter le problème dans un DAG de programmation dynamique et de réutiliser l'algorithmique décrite dans les chapitres précédents.




## 3.2 Recherche de la meilleure analyse

### 3.2.1 Fonctions features

**Exercice 3.1 (Algorithme de Dijkstra ?)** *Est-il possible d'adapter l'algorithme de Dijkstra pour prédire la meilleure séquence de tags avec un modèle de perceptron structuré ? si oui comment ? si non, justifiez.*

## 3.3 Estimation des paramètres

On suppose qu'un corpus annoté $C = (\mathbf{x}_i, \mathbf{y}_i)_{i=1}^N$ est un exemplaire de $N$ phrases. Chaque exemple annoté est un couple $(x_i, y_i)$ qui représente une séquence de mots $x = x_1 \dots x_m$ et une séquence de tags de référence $y = y_1 \dots y_m$ de même longueur.

## 3.4 Estimation des paramètres et approximations

TODO: cette section, introduire la décomposition du score partout, y compris dans le pseudo code.

Pour certains types de problèmes ou de représentations, il n'est pas possible d'évaluer un DAG de programmation dynamique exhaustivement. En pratique, les temps de calcul deviennent prohibitifs. Pour cette raison, on peut souhaiter utiliser une méthode de recherche approximative en faisceau, y compris lors de l'apprentissage.

Dans le contexte d'un problème d'estimation des paramètres, l'utilisation d'un faisceau pose un problème au niveau de la procédure de prédiction

**Algorithm 20** Estimation des paramètres d'un perceptron

**function** TRAIN-PERCEPTRON($(\mathbf{x}_i, \mathbf{y}_i)_{i=1}^N$,$E$)
  $\mathbf{w} \leftarrow \bar{\mathbf{0}}$
  **for** $1 \leqslant e \leqslant E$ **do**
    **for** $1 \leqslant i \leqslant N$ **do**
      $\hat{\mathbf{y}} \leftarrow \text{argmax}_{\mathbf{y} \in \mathbf{Y}}\, \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, \mathbf{y})$ ▷ Tagging
      **if** $\hat{\mathbf{y}} \neq \mathbf{y}_i$ **then**
        $\mathbf{w} \leftarrow \mathbf{w} + [\mathbf{\Phi}(\mathbf{x}_i, \mathbf{y}_i) - \mathbf{\Phi}(\mathbf{x}_i, \hat{\mathbf{y}})]$
      **end if**
    **end for**
  **end for**
  **return w**
**end function**

**Algorithm 21** Estimation des paramètres d'un perceptron moyenné

**function** TRAIN-PERCEPTRON($(\mathbf{x}_i, \mathbf{y}_i)_{i=1}^N$,$E$)
  $\mathbf{w} \leftarrow \mathbf{0}$
  $\bar{\mathbf{w}} \leftarrow \mathbf{0}$
  **for** $1 \leqslant e \leqslant E$ **do**
    **for** $1 \leqslant i \leqslant N$ **do**
      $\hat{\mathbf{y}} \leftarrow \text{argmax}_{\mathbf{y} \in \mathbf{Y}}\, \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, \mathbf{y})$ ▷ Tagging
      **if** $\hat{\mathbf{y}} \neq \mathbf{y}_i$ **then**
        $\mathbf{w} \leftarrow \mathbf{w} + [\mathbf{\Phi}(\mathbf{x}_i, \mathbf{y}_i) - \mathbf{\Phi}(\mathbf{x}_i, \hat{\mathbf{y}})]$
      **end if**
      $\bar{\mathbf{w}} \leftarrow \bar{\mathbf{w}} + \mathbf{w}$
    **end for**
  **end for**
  **return** $\bar{\mathbf{w}}/N$
**end function**

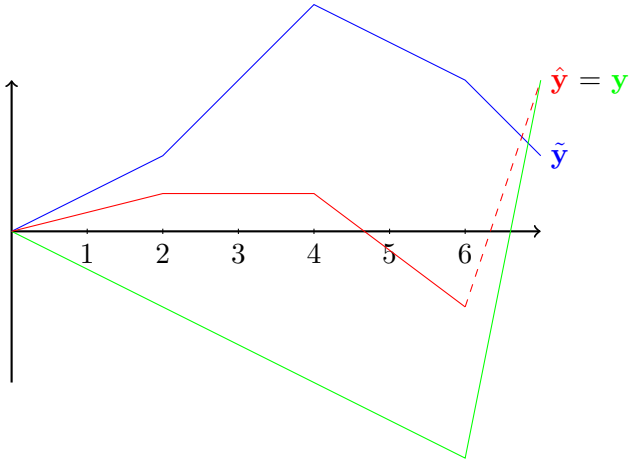
(Algorithme 20, ligne 5) Le problème posé par ce type de méthode est que les méthodes en faisceau renvoient un *pseudo-argmax* qui a pour origine l'inexactitude de l'algorithme de recherche de solutions. Cela signifie que $\hat{\mathbf{y}}$ est potentiellement sous-optimal, ce qui fausse la mise à jour (ligne 7) et peut causer la divergence de la procédure d'estimation.

**Mise à jour d'un modèle à large marge** Pour comprendre le problème, on commence par reformuler la procédure de mise à jour dans le cas exact (sans utiliser un faisceau). Notons $\mathbf{y}^*$ la séquence incorrecte à laquelle le modèle attribue le meilleur score, c'est-à-dire :

$$\mathbf{y}^* = \underset{\mathbf{y} \in \mathbf{Y} \setminus \{\mathbf{y}\}}{\operatorname{argmax}} \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}, \mathbf{y})$$

Il y a nécessairement mise à jour lorsque le score de la meilleure séquence incorrecte est supérieur au score de la séquence correcte : $\Psi(\mathbf{y}^*) \geqslant \Psi(\mathbf{y})$. Lorsque $\Psi(\mathbf{y}^*) < \Psi(\mathbf{y})$ il n'y a pas de mise à jour. Cette reformulation se jsutifie par le fait que le perceptron peut être vu comme un cas particulier de modèle à large marge (Chapitre XXX).

**Généralisation aux méthodes en faisceau** Dans un contexte où on utilise un faisceau ($\mathbf{Y}' \subsetneqq \mathbf{Y}$), la meilleure prédiction dans le beam $\tilde{\mathbf{y}} = \operatorname{argmax}_{\mathbf{y} \in \mathbf{Y}'} \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}, \mathbf{y})$ ne correspond pas nécessairement à la meilleure séquence optimale $\hat{\mathbf{y}} = \operatorname{argmax}_{\mathbf{y} \in \mathbf{Y}} \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}, \mathbf{y})$ qui a pu être écartée du faisceau prématurément. Le cas suivant devient désormais possible $\Psi(\tilde{\mathbf{y}}) < \Psi(\mathbf{y})$ alors que $\hat{\mathbf{y}} = \mathbf{y}$




*On doit garantir que score($\tilde{\mathbf{y}}$) > score($\mathbf{y}$) pour que l'update soit valide*

Une telle configuration entraine la mise à jour des paramètres du perceptron. Dans ce cas on parle de mise à jour invalide. Il a été démontré [**?**] qu'une mise à jour valide respecte nécessairement la condition suivante :

$$\Psi(\tilde{\mathbf{y}}^*_{1\ldots k}) > \Psi(\mathbf{y}_{1\ldots k})$$

où $\tilde{\mathbf{y}}^*_{1\ldots k}$ dénote la sous séquence préfixe de tags incorrecte de meilleur score dans le beam et $\mathbf{y}_{1\ldots k}$ dénote une sous séquence préfixe de tags de référence. Si cette condition est respectée, la convergence de la procédure d'estimation des paramètres est garantie. On peut remarquer que cette condition revient à généraliser le déclenchement de la mise à jour pour les modèles à large marge au cas des sous-séquences.

Dans la pratique, il est courant depuis [**?**] de réaliser la mise à jour dès que la séquence de référence sort du beam pendant l'étape de parsing. C'est ce qu'on appelle la mise à jour rapide (**early update**). D'autres méthodes sont possibles. Par exemple, en définissant la marge d'erreur $\Delta_k = \Psi(\tilde{\mathbf{y}}^*_{1\ldots k}) - \Psi(\mathbf{y}_{1\ldots k})$, et en sélectionnant l'indice $k = \text{argmax}_{1 \leqslant k \leqslant m} \Delta_k$ on obtient la mise à jour dite à **violation maximale**.

# Chapter 4

# Champs conditionnels aléatoires

## 4.1 Généralisation des modèles logistiques

Les modèles de champs conditionnels aléatoires CRF sont des modèle qui généralisent les modèles de régression logistique multinomiale au cas des données structurées. Dans ce cours, on donne une présentation pour la modélisation de séquences, ce qui est le cas d'usage le plus courant.

On se rappelle qu'un modèle de régression logistique multinomiale prédit une donnée $y$ à partir de variables observées $\mathbf{x}$ de la manière suivante :

$$P(Y = y \,|\, \mathbf{x}, \mathbf{w}) = \frac{\exp(\mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y))}{\sum_{y' \in Y} \exp(\mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y'))} \tag{4.1}$$

Le numérateur représente un score $\psi(y) = \exp(\mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y))$ tel que $\psi(y) \geqslant 0$ : la fonction exponentielle transforme toute valeur réelle en réel positif.

Comme chaque classe $y \in Y$ reçoit un score $\psi(y)$, le dénominateur ne dit rien d'autre qu'un score se transforme en probabilité en le divisant par la somme totale des scores pour toutes les classes. Le dénominateur est parfois appelé constante de normalisation.

Un CRF linéaire n'est rien d'autre qu'un modèle de régression logistique multinomiale dont la variable $Y$ à prédire est un ensemble de séquences. Dans ce contexte, la difficulté est que le nombre de séquences possibles de tags (ou plus généralement de symboles) croit exponentiellement en fonction de la longueur de la séquence à prédire. Ainsi il devient rapidement très difficile de réutiliser naïvement le modèle de régression logistique multinomiale

(équation 4.1), car le calcul de la constante de normalisation devient très vite ingérable.

Toute l'idée de CRF c'est de permettre la décomposition du calcul du score d'une séquence de telle sorte que l'on puisse réutiliser les méthodes de programmation dynamique notamment introduites dans les cours précédents pour réaliser les calculs dans des temps raisonnables (complexité polynomiale). Ainsi une séquence de tags $\mathbf{y} = y_1 \dots y_m$ sera prédite par un CRF à l'aide de la formule suivante :

$$P(\mathbf{Y} = \mathbf{y} \,|\, \mathbf{x}, \mathbf{w}) = \frac{\exp\left(\sum_{i=1}^{m} \mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y_{i-1}, y_i)\right)}{\sum_{\mathbf{y}' \in \mathbf{Y}} \exp\left(\sum_{i=1}^{m} \mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y'_{i-1}, y'_i)\right)} \tag{4.2}$$

On voit que dans cette formulation le score global d'une séquence se décompose en une somme de scores locaux[1]. Il ne s'agit de rien d'autre que d'imposer une décomposition du calcul d'un produit scalaire destiné à évaluer une séquence (un produit scalaire est une grosse addition de toute façon).

On va évidemment développer et exploiter cette propriété pour exprimer des algorithmes qui partagent des sous parties de calculs efficacement... On traite en priorité l'algorithmique des deux problèmes d'inférence suivants :

- Prédire la séquence de tags la plus probable $\hat{\mathbf{y}}$ pour une séquence de mots $\mathbf{x}$, c'est-à-dire résoudre le problème suivant :

$$\hat{\mathbf{y}} = \underset{\mathbf{y} \in \mathbf{Y}}{\operatorname{argmax}}\ P(\mathbf{Y} = \mathbf{y} | \mathbf{x}, \mathbf{w}) \tag{4.3}$$

- Estimation par maximum de vraisemblance conditionnelle des paramètres d'un modèle dans un contexte d'apprentissage supervisé. La résolution de ce problème nous fera résoudre par effet de bord le sous-problème de calcul de la probabilité d'une assignation de tags $P(\mathbf{Y} = \mathbf{y} | \mathbf{x}, \mathbf{w})$.

**Exercice 4.1 (fonction exponentielle)** *Faire un graphique avec la librairie graphique de votre choix de la fonction $x \mapsto e^x$ sur l'intervalle réelle $]-\infty, +\infty[$*

## 4.2 Recherche de la séquence de tags la plus probable

**La solution du paresseux** La solution la plus efficace (et à utiliser en pratique) au problème (4.3) est celle du paresseux. Il suffit de remarquer que

[1] Le tout est encore argument d'une fonction exponentielle. Le traitement est détaillé en section suivante.

les fonctions exponentielles utilisées en (4.2) n'ont pas d'autre fonction que de normaliser les scores de produits scalaires pour obtenir des probabilités et que la normalisation ne change pas la valeur du maximum. Par conséquent, chercher la solution de :

$$\hat{\mathbf{y}} = \underset{\mathbf{y}\in\mathbf{Y}}{\operatorname{argmax}} \sum_{i=1}^{m} \mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y_{i-1}, y_i) \tag{4.4}$$

à la place de (4.3) résoud le problème. La solution du paresseux revient donc à réutiliser la méthode de résolution déjà présentée pour le modèle du perceptron global (algorithme de Viterbi).

**La solution par décomposition explicite du score** Le détail de l'autre solution présentée ici (à ne pas utiliser en pratique) permet de mieux comprendre comment le score d'un CRF se décompose. L'idée est de reformuler le score non normalisé d'une séquence par un produit comme suit :

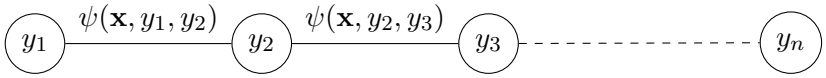
$$\exp\left(\sum_{i=1}^{m} \mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y_{i-1}, y_i)\right)$$

devient :

$$\prod_{i=1}^{m} \exp\left(\mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y_{i-1}, y_i)\right)$$

On peut donc remarquer qu'on peut reformuler le score d'une séquence par un produit de réels strictement positifs appelés potentiels (et notés $\psi(\mathbf{x}, y_{i-1}, y_i) = \exp\left(\mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y_{i-1}, y_i)\right)$).

On peut également donner une représentation graphique au calcul réalisé sous forme de chemin dans un graphe :



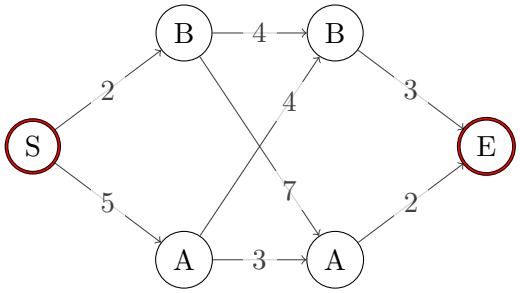

La conséquence est que l'algorithme de Viterbi pour CRF a exactement la même forme que pour HMM : les scores de séquences sont des produits de potentiels et les séquences sont comparées par la fonction maximum. Plus généralement, la reformulation de (4.2) par

$$P(\mathbf{Y} = \mathbf{y} \,|\, \mathbf{x}, \mathbf{w}) = \frac{\prod_{i=1}^{m} \exp\left(\mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y_{i-1}, y_i)\right)}{\sum_{\mathbf{y}'\in\mathbf{Y}} \prod_{i=1}^{m} \exp\left(\mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y'_{i-1}, y'_i)\right)} \tag{4.5}$$

permet de tirer parti de toute l'algorithmique de programmation dynamique déjà développée pour HMM.

## 4.3 À la découverte des DAGs de programmation dynamique

On se propose dans cette section d'étudier par l'exemple quelques propriétés des DAGS de programmation dynamique qui seront déterminantes pour résoudre le problème d'estimation des paramètres d'un CRF.




Le DAG ci-dessus va nous servir d'exemple de départ. Il encode un problème qui correspond à probabiliser toutes les séquences possibles de 2 tags, pris dans le jeu de tags $Y = \{A, B\}$, pour une séquence de deux mots. On ajoute un état source unique $S$ et un état de but à atteindre $E$. La pondération des arcs correspond à des valeurs possibles pour des potentiels $\psi(\mathbf{x}, y_{i-1}, y_i)$ strictement positifs.

En termes de notations, on notera $s_i$ un noeud du DAG où $s \in Y$ est un tag et $i$ sa position. Chaque chemin $\pi = s_1, s_2 \dots s_{k-1}, s_k$ dans le DAG a un score noté $\sigma(\pi)$.

**Algorithm 22** Algorithme de Viterbi pour CRF

```
function ViterbiS,V,s)
    TriTopologique(S, V, s)
    δ(s) ← 1
    for all s ∈ S (suivant ordre topologique) do
        δ(s) ← 0
        for all (s', s) ∈ AE(s) do
            δ(s) ← Max(δ(s), δ(s') × ψ(x, s', s))
        end for
    end for
end function
```

**Exercice 4.2 (Viterbi)** *Simuler l'exécution de l'algorithme de Viterbi (Algorithme 22) sur l'exemple illustratif. Quel est le poids du meilleur chemin*

*? Quel est ce chemin ?*

**Exercice 4.3 (Viterbi)** *Dans la section précédente, on suggère une "solution du paresseux" pour trouver le meilleur chemin dans un* CRF*. L'algorithme 22 est-il directement utilisable dans ce contexte ? Si la réponse est négative, quelles modifications faudrait-il lui apporter ?*

### Algorithme avant

Quelle est la somme des scores qui mènent à un état $s_i$ depuis la source ? C'est la question à laquelle répond l'algorithme avant. Notons $\alpha(s_i)$ la quantité qui correspond à la somme des scores de tous les chemins qui mènent à $s_i$ depuis la source. Par exemple $\alpha(B_2) = (2 \times 4) + (5 \times 4) = 28$.

**Algorithm 23** Algorithme Avant

```
function FORWARD(S,V,s)
    TRITOPOLOGIQUE(S, V, s)
    α(s) ← 1
    for all s ∈ S (suivant ordre topologique) do
        α(s) ← 0
        for all (s', s) ∈ AE(s) do
            α(s) ← α(s) + (α(s') × ψ(x, s', s))
        end for
    end for
end function
```

On peut automatiser ce type de calcul à l'aide de l'algorithme avant qui est une variante, à demi-anneau près, de l'algorithme de Viterbi. Celle-ci est donnée en algorithme 23. La récurrence de cet algorithme est la suivante :

$$\alpha(s_i) = \begin{cases} 1 & \text{si } i = 0 \\ \sum_{(s',s_i) \in AE(s_i)} \alpha(s') \times \psi(\mathbf{x}, s', s_i) & \text{sinon} \end{cases} \tag{4.6}$$

On peut remarquer que la quantité $\alpha(E)$ correspond au facteur de normalisation $Z$, c'est-à-dire la somme des scores de tous les chemins qui mènent de la source jusqu'à la destination (toutes les séquences de tags possibles). Autrement dit, l'algorithme avant permet de résoudre en temps polynomial le problème de sommation de scores pour un nombre exponentiel de séquences de tags. Formellement on a donc que :

$$\alpha(s_i) = \sum_{y_1 \ldots y_i \in \mathbf{Y^i}} \prod_{j=1}^{j=i} \psi(\mathbf{x}, y_{j-1}, y_j)$$

et pour la cas spécifique des séquences complètes :

$$\alpha(s_{m+1}) = Z = \sum_{\mathbf{y}\in\mathbf{Y}} \prod_{j=1}^{j=m} \psi(\mathbf{x}, y_{j-1}, y_j)$$

**Exercice 4.4 (Probabilité d'une séquence)** *Utiliser les propriétés de l'algorithme avant pour calculer $P(S_0, B_1, A_2, E_3)$ la probabilité de la séquence $S, B, A, E$ dans le* DAG *exemple.*

**Exercice 4.5 (Probabilité de toutes les séquences)** *Calculer la probabilité de tous les chemins qui mènent de la source à la destination dans le* DAG *exemple. Vérifier que ces probabilités somment à 1 et que la séquence qui a la plus haute probabilité correspond à celle que vous avez trouvé avec l'algorithme de Viterbi.*

### Algorithme arrière

Quelle est la somme des scores qui mènent à un état $s_i$ depuis la destination ? C'est la question à laquelle répond l'algorithme arrière. On notera $\beta(s_i)$ la quantité qui correspond à la somme des scores de tous les chemins qui mènent à $s_i$ depuis la destination. Par exemple $\beta(B_1) = (2\times3)+(3\times4) = 18$. Il s'agit essentiellement d'une variante miroir de l'algorithme avant. La récurrence est la suivante :

$$\beta(s_i) = \begin{cases} 1 & \text{si } i = m+1 \\ \sum_{(s_i,s')\in AS(s_i)} \beta(s') \times \psi(\mathbf{x}, s_i, s') & \text{sinon} \end{cases} \tag{4.7}$$

En tant que tel cet algorithme arrière n'a pas beaucoup d'intérêt. C'est en combinaison avec l'algorithme avant que l'on peut faire émerger son utilité.

**Exercice 4.6 (Pseudo-code)** *Donner un pseudo code pour l'algorithme arrière.*

### Combiner les quantités avant et arrière

Utilisées en combinaison, les quantités $\alpha(s_i)$ et $\beta(s_i)$ permettent de donner des probabilités à des sous-chemins dans un DAG.

L'exemple donné en figure 4.1 illustre les quantités $\alpha(s_i)$, notées en bleu sur chaque noeud, et les quantités $\beta(s_i)$ sont notées en rouge. On peut commencer par observer que la quantité $\alpha(E_4) = 573$ correspond au facteur

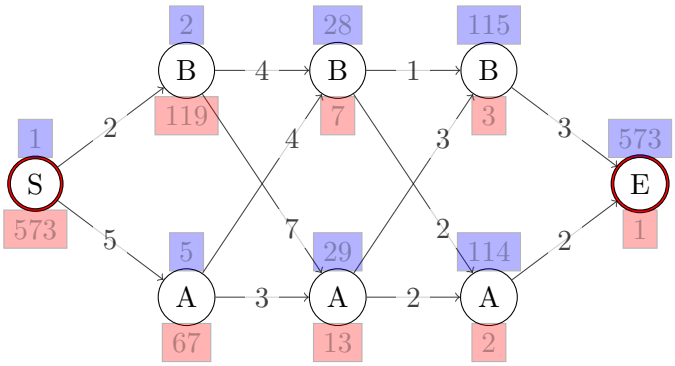



Figure 4.1: Treillis de programmation dynamique annoté par $\alpha$ et $\beta$

$Z$. On obtient la probabilité d'un chemin (d'une séquence de tags) comme par exemple le chemin $S_0, A_1, B_2, B_3, E_4$ en réalisant la division suivante :

$$P(S_0, A_1, B_2, B_3, E_4) = \frac{5 \times 4 \times 1 \times 3}{573} = \frac{60}{573}$$

En utilisant la quantité $\alpha(s_i)$, on peut déterminer la probabilité d'un séquence de tags qui termine par $B, B, E$ de la manière suivante :

$$P(\triangleleft, B_2, B_3, E_4) = \frac{\alpha(B_2) \times \psi(\mathbf{x}, B_2, B_3) \times \psi(\mathbf{x}, B_3, E_4)}{Z} = \frac{28 \times 1 \times 3}{573} = \frac{84}{573}$$

En continuant le raisonnement on peut calculer la probabilité de suivre une transition (par exemple $A_1, B_2$ représente la probabilité de tagguer le premier mot $A$ et le second mot $B$) en utilisant les quantités $\alpha(s_i)$ et $\beta(s_i)$ :

$$P(\triangleleft, A_1, B_2, \triangleright) = \frac{\alpha(A_1) \times \psi(\mathbf{x}, A_1, B_2) \times \beta(B_2)}{Z} = \frac{5 \times 4 \times 7}{543} = \frac{140}{543}$$

En résumé, et comme illustré dans les exercices qui suivent, on peut en réalité transformer un graphe de programmation dynamique en calculette.

Ce dernier exemple introduit une quantité clé qui est réutilisée par la méthode d'estimation des paramètres par descente de gradient qui est :

$$P(\triangleleft, s_i, s_{i+1}, \triangleright | \mathbf{x}; \mathbf{w}) = \frac{\alpha(s_i) \times \psi(\mathbf{x}, s_i, s_{i+1}) \times \beta(s_{i+1})}{Z} \tag{4.8}$$

Celle-ci correspond à la probabilité de suivre une transition entre deux tags consécutifs. Utiliser les quantités avant et arrière pour calculer ces proba-

bilités de transition, c'est utiliser l'**algorithme avant/arrière**. Formellement, on peut remarquer que $P(\lhd, s_i, s_{i+1}, \rhd|\mathbf{x})$ correspond à :

$$\begin{aligned} P(\lhd, s_i, s_{i+1}, \rhd|\mathbf{x}) &= \frac{1}{Z} \sum_{y_1 \dots y_{i-1}} \prod_{j=1}^{j=i-1} \psi(\mathbf{x}, y_{i-1}, y_i) && (4.9) \\ &\times \quad \psi(\mathbf{x}, y_{i-1}, y_i) && (4.10) \\ &\times \sum_{y_i \dots y_m} \prod_{j=i+1}^{j=m} \psi(\mathbf{x}, y_{j-1}, y_j) && (4.11) \end{aligned}$$

**Exercice 4.7** *À partir de la figure 4.1, donner la probabilité que le second mot soit taggué B, c'est-à-dire,* $P(\lhd, B_2, \rhd)$.

**Exercice 4.8** *Observer que* $P(\lhd, B_2, \rhd) + P(\lhd, A_2, \rhd) = 1$ *à partir de la figure 4.1. Expliquer informellement pourquoi.*

## 4.4 Estimation des paramètres

L'estimation des paramètres d'un CRF consiste à déterminer les valeurs du vecteur de paramètres $\mathbf{w}$ à partir d'un corpus annoté.

On suppose qu'un corpus annoté $C = (\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ est un exemplaire de $N$ phrases. Chaque exemple annoté est un couple $(\mathbf{x}_i, \mathbf{y}_i)$ qui représente une séquence de mots $\mathbf{x} = x_1 \dots x_m$ et une séquence de tags de référence $\mathbf{y} = y_1 \dots y_m$ de même longueur.

Comme pour la régression logistique multinomiale, on présente ici l'estimation des paramètres par maximum de (log-) vraisemblance conditionnelle.

$$L(\mathbf{w}) = \sum_{i=1}^{N} \log P(\mathbf{y}_i|\mathbf{x}_i; \mathbf{w}) \tag{4.12}$$

Autrement dit on cherche à résoudre le problème d'optimisation suivant :

$$\hat{\mathbf{w}} = \underset{\mathbf{w} \in \mathbb{R}^d}{\operatorname{argmax}} \sum_{i=1}^{N} \log P(\mathbf{y}_i|\mathbf{x}_i; \mathbf{w}) \tag{4.13}$$

En substituant la probabilité $P(\mathbf{y}_i|\mathbf{x}_i; \mathbf{w})$ par sa définition en équation (4.2), on optimise :

$$\hat{\mathbf{w}} = \underset{\mathbf{w} \in \mathbb{R}^d}{\operatorname{argmax}} \sum_{i=1}^{N} \log \frac{\exp\left(\sum_{i=1}^{m} \mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y_{i-1}, y_i)\right)}{\sum_{\mathbf{y}' \in \mathbf{Y}} \exp\left(\sum_{i=1}^{m} \mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}, y'_{i-1}, y'_i)\right)} \tag{4.14}$$

Les dérivées partielles ont la même allure que pour la régression logistique multinomiale :

$$\frac{\partial L(\mathbf{w})}{\partial w_k} = \sum_{i=1}^{N} C_k(\mathbf{x}_i, \mathbf{y}_i) - \sum_{i=1}^{N} \sum_{\mathbf{y} \in \mathbf{Y}} P(\mathbf{y}|\mathbf{x}_i, \mathbf{w}) C_k(\mathbf{x}_i, \mathbf{y}) \tag{4.15}$$

où l'abbréviation $C_k(\mathbf{x}, \mathbf{y})$ représente le comptage de la feature $\phi_k$ dans une séquence de référence, c'est-à-dire :

$$C_k(\mathbf{x}_i, \mathbf{y}_i) = \sum_{j=1}^{m} \phi_k(\mathbf{x}_i, y_{j-1}^i, y_j^i)$$

Le calcul du premier terme en (4.15) consiste à compter les occurrences de $\phi_k$ dans l'ensemble des séquences du corpus de référence.

En ce qui concerne le second terme en (4.15), il faut compter les occurrences de $\phi_k$ – dans toutes les séquence de tags possibles – en pondérant chaque occurrence par la probabilité de la séquence de tags dans laquelle elle apparait. Naïvement, ce calcul demande d'énumérer l'ensemble exponentiel $\mathbf{Y}$ de toutes les séquences de tags $\mathbf{y} \in \mathbf{Y}$ possibles pour chaque exemple du jeu de données.

Le développement qui suit consiste à montrer comment utiliser les techniques de programmation dynamique présentées précédemment pour réaliser ce calcul de manière efficace. Le second terme de la formule (4.15) nous dit que pour chaque exemple dans les données il faut calculer :

$$\sum_{\mathbf{y} \in \mathbf{Y}} P(\mathbf{y}|\mathbf{x}, \mathbf{w}) \sum_{j=1}^{m} \phi_k(\mathbf{x}, y_{j-1}, y_j) \tag{4.16}$$

$$= \sum_{\mathbf{y} \in \mathbf{Y}} \sum_{j=1}^{m} P(\mathbf{y}|\mathbf{x}, \mathbf{w}) \phi_k(\mathbf{x}, y_{j-1}, y_j) \tag{4.17}$$

$$= \sum_{j=1}^{m} \sum_{\mathbf{y} \in \mathbf{Y}} P(\mathbf{y}|\mathbf{x}, \mathbf{w}) \phi_k(\mathbf{x}, y_{j-1}, y_j) \tag{4.18}$$

$$= \sum_{j=1}^{m} \sum_{y_{j-1}} \sum_{y_j} P(\triangleleft, y_{j-1}, y_j, \triangleright|\mathbf{x}, \mathbf{w}) \phi_k(\mathbf{x}, y_{j-1}, y_j) \tag{4.19}$$

Au terme de cette reformulation, l'évaluation du second terme revient à compter les occurrences de la feature $\phi_k$ – pondérées par les probabilités de transitions où elles apparaissent – dans le DAG de programmation dynamique, et ce pour l'ensemble des exemples du jeu de données.

# Chapter 5

# Modèles locaux et modèles récurrents

## 5.1 Modèles markoviens à maximum d'entropie

Des modèles d'étiquetage de la famille CRF supposent d'assigner une séquence de tags $\mathbf{y}$ à une séquence de mots $\mathbf{x}$, ce qui est résumé par la formule (4.3) répétée ici :

$$\hat{\mathbf{y}} = \underset{\mathbf{y} \in \mathbf{Y}}{\operatorname{argmax}}\ P(\mathbf{Y} = \mathbf{y} | \mathbf{x}; \mathbf{w}) \tag{5.1}$$

On a vu que le problème d'optimisation de CRF fait intervenir des algorithmes coûteux en temps de calcul. La difficulté vient du fait que l'ensemble $\mathbf{Y}$ de toutes les séquences de tags possibles croit exponentiellement en fonction de la longueur de la phrase.

La motivation des modèles locaux est de simplifier la procédure d'estimation des paramètres en reformulant (4.3) par l'approximation suivante :

$$P(\mathbf{Y} = \mathbf{y} | \mathbf{x}; \mathbf{w}) \approx \prod_{i=2}^{n} P(y_i | y_{i-1}, \mathbf{x}; \mathbf{w}) \tag{5.2}$$

ce qui est une réutilisation de l'approximation markovienne classique

$$P(\mathbf{y} = y_1 \dots y_n | \mathbf{x}; \mathbf{w}) \approx \prod_{i=2}^{n} P(y_i | y_{i-1}, \mathbf{x}; \mathbf{w})$$

L'équation (5.2) permet de réduire le problème d'apprentissage structuré à un cas d'apprentissage non structuré où la tâche de prédiction consiste à prédire la probabilité du tag $y_i$ d'un mot donné sachant la séquence de mots

$\mathbf{x}$ et le tag du mot précédent $y_{i-1}$. Un tel modèle se réduit donc à un modèle de régression logistique multinomiale (ou modèle à maximum d'entropie) où:

$$P(y_i|y_{i-1}, \mathbf{x}; \mathbf{w}) = \frac{\exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}, y_{i-1}, y_i))}{\sum_{y' \in Y} \exp(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}, y_{i-1}, y'))}$$

Les paramètres du modèle étant connus, on peut en théorie réaliser la prédiction avec tout algorithme de recherche de solutions approprié (Viterbi, Dijkstra ou toute approximation en faisceau).

**Estimation des paramètres** Un jeu de données pour un modèle local est un ensemble de couples $(\mathbf{x}_i y_{i-1}, y_i)_{i=1}^N$ où $\mathbf{x}_i$ est une séquence de mots (et le tag $y_{i-1}$ dans le corpus) et $y_i$ est une catégorie morphosyntaxique. L'estimation des paramètres du modèle se fait donc en optimisant la log-vraisemblance des données en utilisant par exemple un algorithme de descente de gradient stochastique ou tout algorithme d'optimisation approprié.

**Error propagation** On peut remarquer que contrairement à un CRF, un modèle markovien à maximum d'entropie ne visite pas tous les états. Il ne voit que les états supposés "gold", c'est-à-dire que $y_{i-1}$ sera toujours supposé correct dans les données d'entrainement. Par conséquent ce type de modèle est en théorie exposé à des effets de propagation d'erreur : le modèle n'est pas naturellement entraîné pour traiter des états où il est est arrivé par erreur.

**Label bias** Le problème du *label bias* est lié à la prise de décision locale, par exemple :

- The robot wheels Fred round
- The robot wheels are round

Si la tâche est de prédire la catégorie morphologique des mots, la décision sera identique dans les deux cas avec un modèle local (dont les facteurs sont de la forme $P(y_i|y_{i-1}, \mathbf{x})$) pour étiqueter *wheels*.

## 5.2 Modèles neuronaux

**Modèles de langage neuronaux pour le tagging** Une autre famille importante de modèles locaux sont constitués par des réseaux de neurones à propagation avant dont la couche de sortie est de type softmax de telle

sorte que les sorties du réseau s'interprètent comme des probabilités de la forme $P(y_t|y_{t-1}, \mathbf{x}; \mathbf{w})$.

L'archétype de ce type de modèle est le modèle de langage neuronal (NNLM Bengio et al 2003) qui est un réseau à propagation avant qui prend la forme suivante :

$$\mathbf{y} = \text{softmax}(\mathbf{W}_1\mathbf{h}) \tag{5.3}$$

$$\mathbf{h} = g(\mathbf{W}_2\mathbf{e}) \tag{5.4}$$

$$\mathbf{e} = \begin{bmatrix} \mathbf{E}\mathbf{x}_1 \\ \vdots \\ \mathbf{E}\mathbf{x}_k \end{bmatrix} \tag{5.5}$$

où $\mathbf{x}_1 \dots \mathbf{x}_k$ sont des vecteurs one-hot qui codent les symboles prédicteurs. Ces vecteurs représentent par exemple les quelques mots qui précèdent ou les quelques mots qui suivent le mot $w_t$ (à tagguer par $y_t$) ainsi que le tag du mot précédent ($y_{t-1}$).

La couche de sortie softmax de ce modèle s'interprète comme une probabilité conditionnelle de la forme $P(y_t|y_{t-1}, \mathbf{x})$. Autrement dit, ce modèle est une variante de MEMM.

Les paramètres du modèle étant connus, on peut réaliser la prédiction avec tout algorithme de recherche de solutions approprié (Viterbi, Dijkstra ou toute approximation en faisceau).

L'estimation des paramètres se fait de manière analogue au cas des MEMM en optimisant la log-vraisemblance des données en utilisant par exemple un algorithme de descente de gradient stochastique ou tout algorithme d'optimisation approprié.

**Réseaux de neurones récurrents pour le tagging** Les réseaux de neurones récurrents constituent de nos jours une alternative très populaire aux MEMM et à leurs variantes depuis l'introduction des modèles de langages à réseaux récurrents (RNNLM) par Mikolov 2010.

Les réseaux de neurones récurrents ont pour propriété intéressante de permettre de mémoriser l'historique de la phrase dans une cellule (vecteur noté $\mathbf{h}$) qui s'interprète comme une mémoire.

Le problème de prédiction de séquence est vu comme une succession d'étapes de prédiction temporelles qui à chaque étape mettent à jour la mémoire du système en fonction de l'état courant de la mémoire et du symbole courant. La forme primitive d'un réseau de ce type est la suivante. Soit une séquence de vecteurs $\mathbf{x}_t$ ($1 \leqslant t \leqslant n$) qui est donnée, le réseau prédit la

séquence de vecteurs $\mathbf{y}_t$ comme suit:

$$\begin{aligned}
\mathbf{y}_t &= \text{softmax}(\mathbf{W}_0\mathbf{h}_t) && (1 \leqslant t \leqslant n)\\
\mathbf{h}_t &= g(\mathbf{W}_1\mathbf{x}_t + \mathbf{W}_2\mathbf{h}_{t-1} + \mathbf{b}) && (1 \leqslant t \leqslant n)
\end{aligned}$$

Il faut remarquer que la mémoire $\mathbf{h}$ est mise à jour à chaque étape temporelle. Pour la modélisation du langage, on utilisera en général la forme où les données sont des embeddings, c'est-à-dire un réseau de la forme:

$$\begin{aligned}
\mathbf{y}_t &= \text{softmax}(\mathbf{W}_0\mathbf{h}_t) && (1 \leqslant t \leqslant n)\\
\mathbf{h}_t &= g(\mathbf{W}_1\mathbf{e}_t + \mathbf{W}_2\mathbf{h}_{t-1} + \mathbf{b}) && (1 \leqslant t \leqslant n)\\
\mathbf{e}_t &= \mathbf{E}\mathbf{x}_t && (1 \leqslant t \leqslant n)
\end{aligned}$$

où on suppose que chaque $\mathbf{x}_t$ est un vecteur one hot qui code le mot $w_t$ pris dans la séquence $w_1 \dots w_n$.

Dans le cas du tagging, on observe que la probabilité de sortie du modèle décrit ci-dessus s'interprète comme la probabilité $P(y_t|w_1 \dots w_t)$ car la mémoire du RNN n'est pas bornée. Et c'est cet aspect non borné qui lui donne son originalité principale par contraste avec les autres modèles présentés dans ce cours qui posent tous des hypothèses simplificatrices de type markovienne.

La prédiction avec un tel modèle est en général réalisée de manière gloutonne. À chaque étape temporelle le tag prédit est celui dont le score maximise la couche de sortie:

$$\hat{y}_t = \underset{y_t \in Y}{\text{argmax}}\, \mathbf{y}_t$$

**Réseaux récurrents empilés** Parmi les nombreuses variantes de réseaux récurrents, signalons le cas des réseaux empilés. Il s'agit de réseaux de la forme:

$$\begin{aligned}
\mathbf{y}_t &= \text{softmax}(\mathbf{W}_0\mathbf{h}_t^j)\\
\mathbf{h}_t^j &= g(\mathbf{W}_1^j\mathbf{h}_t^{j-1} + \mathbf{W}_2^j\mathbf{h}_{t-1}^j + \mathbf{b}^j)\\
&\vdots\\
\mathbf{h}_t^1 &= g(\mathbf{W}_1^1\mathbf{x}_t + \mathbf{W}_2^1\mathbf{h}_{t-1}^1 + \mathbf{b}^1)
\end{aligned}$$

Parmi ceux-ci, les réseaux bi-récurrents constituent un cas particulier emblématique. Il s'agit de réseaux empilés à deux couches dont la forme

élémentaire est la suivante:

$$\mathbf{y}_t = \text{softmax}(\mathbf{W}_0 \begin{bmatrix} \mathbf{h}_t^b \\ \mathbf{h}_t^f \end{bmatrix})$$
$$\mathbf{h}_t^b = g(\mathbf{W}_1^b \mathbf{x}_t + \mathbf{W}_2^b \mathbf{h}_{t+1}^b + \mathbf{b}^b)$$
$$\mathbf{h}_t^f = g(\mathbf{W}_1^f \mathbf{x}_t + \mathbf{W}_2^f \mathbf{h}_{t-1}^f + \mathbf{b}^f)$$

Ces réseaux bi-récurrents possèdent deux mémoires: $\mathbf{h}_t^b$ et $\mathbf{h}_t^f$ qui représentent respectivement les contextes non bornés droite et gauche du mot en position $t$.

Cette dernière propriété permet de voir un bi-RNN comme un procédé d'encodage de mots sur des vecteurs qui tient compte du contexte dans lequel apparaît chaque mot. Ainsi les vecteurs $\mathbf{h}_t^b$ et $\mathbf{h}_t^f$ peuvent être utilisés à la place de word embeddings classiques comme représentation des mots pour des tâches additionnelles comme par exemple une tâche d'analyse syntaxique.

**Estimation des paramètres** L'estimation des paramètres pour des réseaux récurrents se fait par maximum de vraisemblance à l'aide d'une descente de gradient classique comme SGD. À chaque évaluation du gradient, pour l'ensemble des données, l'algorithme de rétropropagation du gradient compare la prédiction du modèle pour chaque mot à l'annotation de référence et met à jour les poids de manière appropriée.

L'évaluation du gradient pose en général des problèmes formels non négligeables pour les RNN standards (disparition ou explosion du gradient). Pour cette raison on utilise en pratique des variantes appelées LSTM (long short term memory network) ou GRU (gated recurrent unit) qui corrigent le problème mais dont les formulations explicites sont plus complexes.

**Aspects pratiques** Les jeux de données pour les modèles récurrents ont la forme $\{(\mathbf{w}_i, \mathbf{t}_i)\}_{i=1}^N$ où $\mathbf{w}_i$ représente une séquence de mots et $\mathbf{t}_i$ une séquence de tags.

La descente de gradient de modèles récurrents peut, selon les cas, demander un calcul long et intensif. Pour des raisons d'efficacité tant quant à l'utilisation de CPU que de GPU, les bibliothèques appliquent une procédure de descente de gradient organisée en mini-batches. Les exemples d'un même mini batch sont traités en parallèle pour rendre la descente de gradient plus efficace.

Cette méthode impose que les séquences d'un même mini-batch ont toutes exactement la même longueur. Lorsque cela n'est pas possible il est

d'usage d'ajouter des éléments factices au début ou à la fin de séquences trop courtes (*padding*) et de tronquer des séquences trop longues (*truncation*).

**Réseaux de neurones récurrents pour la classification de phrases** À côté des réseaux récurrents transductifs comme ceux utilisés pour le cas de l'étiquetage morphosyntaxique, on utilise également des réseaux agglomérants dont le but est de prédire une étiquette $y$ pour une séquence d'observables $x_1 \dots x_n$ comme des mots ou des caractères.

Les cas d'utilisations sont par exemple : catégoriser un mot inconnu à partir de la séquence de caractères qui le composent ou catégoriser une phrase par une catégorie de sentiment. La forme d'un tel réseau récurrent est la suivante:

$$\begin{aligned} \mathbf{y}_t &= \text{softmax}(\mathbf{W}_0\mathbf{h}_t) && (t = n) \\ \mathbf{h}_t &= g(\mathbf{W}_1\mathbf{x}_t + \mathbf{W}_2\mathbf{h}_{t-1} + \mathbf{b}) && (1 \leqslant t \leqslant n) \end{aligned}$$

Lorsque les $\mathbf{x}_t$ sont produits par des embeddings de mots ou de caractères, le réseau prend alors la forme suivante:

$$\begin{aligned} \mathbf{y} &= \text{softmax}(\mathbf{W}_0\mathbf{h}_t) && (t = n) \\ \mathbf{h}_t &= g(\mathbf{W}_1\mathbf{e}_t + \mathbf{W}_2\mathbf{h}_{t-1} + \mathbf{b}) && (1 \leqslant t \leqslant n) \\ \mathbf{e}_t &= \mathbf{E}\mathbf{x}_t && (1 \leqslant t \leqslant n) \end{aligned}$$

Il faut remarquer que $\mathbf{y}$ n'est calculé q'une seule fois lors de la dernière étape temporelle. Dans ce contexte, on interprète la couche cachée de la dernière étape temporelle $\mathbf{h}_n$ comme une représentation vectorielle de la séquence tout entière (un embedding de séquence).

Ce type de modèle peut être utilisé tel quel dans différents contextes : comme par exemple pour catégoriser des phrases en analyse de sentiments. On peut également utiliser un tel réseau sur des séquences de caractères pour prédire la catégorie d'un mot inconnu ou encore utiliser $\mathbf{h}_n$ comme embedding 'morphologique' de mot et le donner comme embedding d'entrée à un modèle d'étiquetage morphosyntaxique ou d'analyse syntaxique.

Dans la pratique, et pour les mêmes raisons que pour les modèles transductifs de séquences, les RNN sont instanciés par des LSTM ou des GRU.

## 5.3 Différenciation automatique et graphe de calcul

Les réseaux de neurones s'analysent comme une composition, en général complexe, de fonctions à paramètres et à valeurs vectorielles. Celles-ci sont

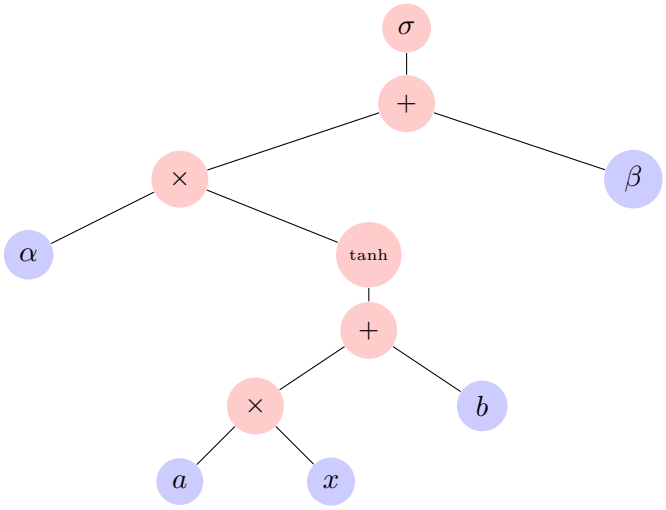
fréquemment non linéaires.

L'optimisation de telles fonctions se résoud en général par descente de gradient. Le calcul du vecteur de dérivées partielles n'est en général pas réalisé analytiquement mais par une méthode de différentiation automatique qui s'appuie sur la notion de **graphe de calcul**.

Un graphe de calcul est un graphe acyclique orienté dont les feuilles représentent soit des paramètres soit des données et dont les noeuds représentent des fonctions (ou des opérateurs). Le graphe de calcul est une manière commode de représenter informatiquement une fonction. Pour illustrer, prenons l'exemple de la fonction :

$$f(x) = \sigma(\alpha \tanh(ax + b) + \beta)$$

dont la donnée est le réel $x$ et les paramètres sont les réels $a, b, \alpha, \beta$. Le graphe de calcul est le suivant :

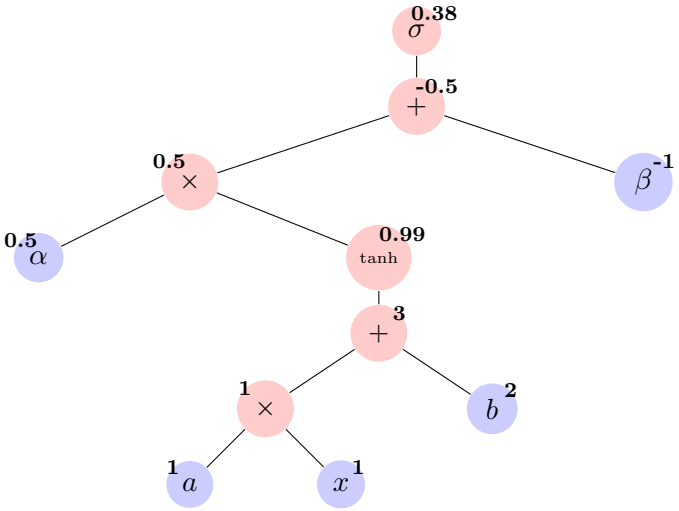



**Propagation avant** Un graphe de calcul peut être utilisé pour valuer une fonction si on connait les valeur des paramètres et qu'on reçoit une donnée. Évaluer une fonction à l'aide d'un graphe de calcul c'est utiliser l'algorithme de propagation avant.

Sur l'exemple précédent, en supposant la valuation des paramètres suivante :

| | |
|---|---|
| $a$ | 1 |
| $b$ | 2 |
| $\alpha$ | 0.5 |
| $\beta$ | -1 |
| $x$ | 1 |

on obtient le graphe de calcul valué suivant :




**Propagation arrière** L'algorithme de descente de gradient demande à chaque itération de réévaluer le gradient de la fonction, c'est-à-dire le vecteur de dérivées partielles pour chacune des variables de la fonction objective. Le graphe de calcul permet également d'automatiser cette opération.

Pour en comprendre le fonctionnement, rappelons nous la règle de dérivation des fonctions composées :

$$(f \circ g)'(x) = f'(g(x))g'(x)$$

Celle-ci peut s'exprimer également en notation de Leibniz en donnant un nom à $g(x)$, comme par exemple $y = g(x)$ :

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta y}\frac{\delta g}{\delta x}$$

Ce rappel effectué, on propose maintenant de continuer notre exemple en dérivant d'abord analytiquement le gradient de la fonction par rapport à chacun des paramètres $a, b, \alpha, \beta$ en utilisant la règle de composition. Pour

simplifier la notation, on propose d'abord de nommer les différentes sous-fonctions :

$$\begin{aligned} g &= \alpha \tanh(ax+b) + \beta \\ h &= \alpha \tanh(ax+b) \\ i &= \tanh(ax+b) \\ j &= ax+b \\ k &= ax \end{aligned}$$

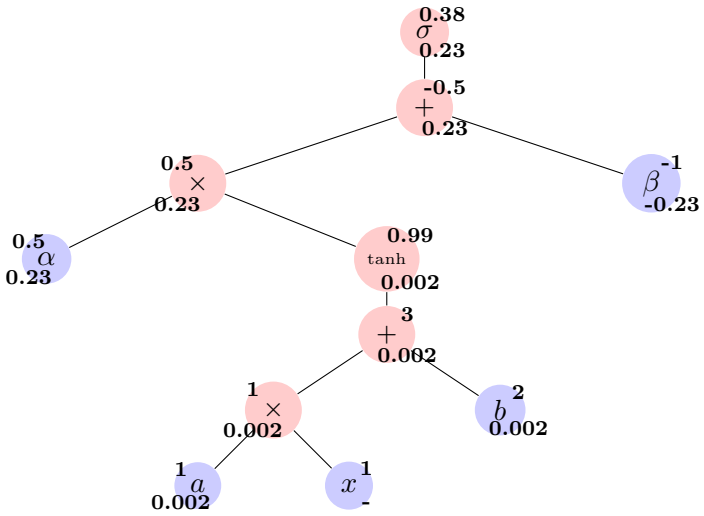
On peut ainsi exprimer aisément les dérivées partielles pour les différents paramètres[1] :

$$\begin{aligned} \frac{\partial f}{\partial \beta} = \frac{\partial \sigma}{\partial g}\frac{\partial +}{\partial \beta} &= \sigma(g)(1-\sigma(g))(0-1) \\ &= \sigma(-0.5)(1-\sigma(-0.5))(-1) \\ &= -0.23 \\ \frac{\partial f}{\partial \alpha} = \frac{\partial \sigma}{\partial g}\frac{\partial +}{\partial h}\frac{\partial \times}{\partial \alpha} &= \sigma(g)(1-\sigma(g))i \\ &= \sigma(-0.5)(1-\sigma(-0.5))0.99 \\ &= 0.23 \\ \frac{\partial f}{\partial b} = \frac{\partial \sigma}{\partial g}\frac{\partial +}{\partial h}\frac{\partial \times}{\partial i}\frac{\partial \tanh}{\partial j}\frac{\partial +}{\partial b} &= \sigma(g)(1-\sigma(g))(1-\tanh^2(j))1 \\ &= \sigma(-0.5)(1-\sigma(-0.5))(1-\tanh^2(3)) \\ &= 0.002 \\ \frac{\partial f}{\partial a} = \frac{\partial \sigma}{\partial g}\frac{\partial +}{\partial h}\frac{\partial \times}{\partial i}\frac{\partial \tanh}{\partial j}\frac{\partial +}{\partial k}\frac{\partial \times}{\partial a} &= \sigma(g)(1-\sigma(g))(1-\tanh^2(j))x \\ &= \sigma(-0.5)(1-\sigma(-0.5))(1-\tanh^2(3))1 \\ &= 0.002 \end{aligned}$$

On voit que le développement analytique, qui repose sur l'utilisation de la règle de dérivation des fonctions composées produit pour chaque paramètre une multiplication qui correspond au produit des dérivées des fonctions que l'on trouve sur le chemin qui mène de la racine de l'arbre à la feuille qui représente ce paramètre.

[1] Rappelons nous que la dérivée de la fonction sigmoide est la fonction $\sigma'(x) = \sigma(x)(1-\sigma(x))$ et que la dérivée de la fonction tangent hyperbolique est la fonction $tanh'(x) = 1 - tanh(x)^2$

Une implémentation informatique de ce graphe de calcul pour calculer le vecteur gradient d'une fonction objective est une application de l'**algorithme de rétropropagation du gradient**. En général l'exécution de l'algorithme garde en mémoire les valeurs déjà calculées et organise le calcul en suivant une structure de graphe acyclique orienté pour éviter les reduplications, à la manière des algorithmes de programmation dynamique.




On généralise ce premier exemple au cas d'un réseau de neurones quelconque en remplaçant les entrées et les sorties par des valeurs vectorielles plutôt que scalaires. Les paramètres prennent alors en général la forme de matrices (ou de vecteurs lorsqu'il s'agit des biais). Cette généralisation ne change rien au principe de l'algorithme de rétropropagation exposé ci-dessus.

L'exemple dévéloppé ici se réexprime sous forme vectorielle par un réseau du type perceptron multi-couche classique :

$$f(\mathbf{x}) = \sigma(\mathbf{A}_1 \tanh(\mathbf{A}_2\mathbf{x} + \mathbf{b}_2) + \mathbf{b}_1)$$

**Graphes de calculs dynamiques en NLP** ...

## 5.4 Calculs massivement parallèles sur GPU

Les modèles de Deep Learning contemporains se caractérisent par une certaine lenteur à l'apprentissage qui est notamment liée à la quantité considérable de paramètres dans les modèles.

**Observation de départ** On observe toutefois que les modèles de Deep Learning réalisent essentiellement des opérations sur des matrices (ou des Tenseurs[2]) et que ces opérations se parallélisent très facilement.

Pour illustrer rappelons nous qu'un modèle linéaire est un modèle de la forme

$$y = \mathbf{w}^T \mathbf{x}$$

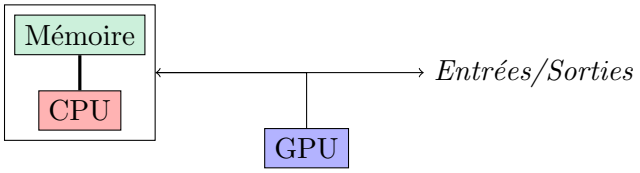
c'est-à-dire un produit scalaire d'un vecteur ligne $\mathbf{w}$ qui représente les paramètres et d'un vecteur colonne $\mathbf{x}$ qui représente un exemple dans un jeu de données. Si on organise le jeu de données de $N$ exemples comme une matrice $\mathbf{X}$ de $N$ colonnes, le modèle linéaire réalise l'opération :

$$\mathbf{y} = \mathbf{w}^T \mathbf{X}$$

ce qui a pour effet de réaliser toutes les opérations de prédiction d'un jeu de données en une seule opération. Conceptuellement cette opération matricielle se parallélise très simplement car elle correspond à $N$ opérations indépendantes de multiplication de $\mathbf{w}$ par les vecteurs $\mathbf{x}_1 \dots \mathbf{x}_N$.

C'est très exactement le type d'opérations qui sont réalisées très efficacement et en parallèle par un GPU moderne (en plus des opérations purement graphiques).

**Architecture matérielle et usage du GPU** L'usage du GPU pose toutefois un problème pratique dont la cause provient de l'architecture même des ordinateurs. Schématiquement les ordinateurs suivent une architecture de von Neumann :




On observe que suivant cette architecture le processeur est lié à la mémoire par un bus qui permet une communication rapide avec le CPU alors que le GPU fait partie des dispositifs d'entrée sortie (au même titre que la souris, le clavier ou la carte son pour ne citer que les principaux). Le bus qui connecte les appareils d'entrée/sortie à la mémoire ne permet par contre qu'une communication lente.

[2]Un tenseur est une généralisation de vecteurs ou de matrices à un ordre supérieur.

La conséquence pratique est que le GPU autorise de réaliser des opérations matricielles très efficacement mais que le transfert de données depuis (et vers) la mémoire centrale est lent.

Par conséquent la méthode de programmation avec GPU consiste à minimiser le transfert de données entre le GPU et l'unité centrale. Cela se fait en réalisant peu d'opérations impliquant de gros calculs plutôt qu'en multipliant des petits calculs qui répètent le transfert de données sur un bus lent.

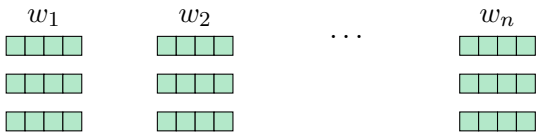
**La méthode du (mini-)batch en deep learning** Comme la mémoire sur un GPU est limitée, on stockera typiquement le jeu de données dans la mémoire centrale de la machine et on transfère successivement des **batchs** d'exemples pour lesquels on va réaliser les calculs de prédiction[3].

Dans le cas du modèle linéaire, plutôt que de répéter successivement des opérations de la forme $\mathbf{w}^T\mathbf{x}_i$ pour chaque vecteur $\mathbf{x}_i$ d'un jeu de données on préfère réaliser des opérations de la forme $\mathbf{w}^T\mathbf{X}$ où $\mathbf{X}$ est un mini-batch d'exemples $\mathbf{x}_i \dots \mathbf{x}_{i+B}$ ($B$ est la taille du mini-batch).

Dans un contexte de deep learning, l'utilisation de mini-batchs généralise le cas d'exemple donné pour le modèle linéaire à des modèles d'apprentissage profond de structure beaucoup plus complexe. Toutefois, on ne peut en général pas utiliser des batchs comportant un trop grand nombre d'exemples (comme la totalité du jeu de données) pour éviter de saturer la mémoire du GPU.

**Batcher des données structurées** Pour la modélisation du langage, le cas classique est que la donnée du problème de prédiction ne se réduit pas à un simple vecteur $\mathbf{x}$. Une ligne de données est une séquence de vecteurs d'embeddings $\mathbf{X} = \mathbf{x}_1 \dots \mathbf{x}_n$, un par mot. Par exemple, pour le tagging le jeu de données est typiquement une liste d'exemples $D = (\mathbf{X}_i, \mathbf{Y}_i)_{i=1}^N$ où $\mathbf{X}_i$ et $\mathbf{Y}_i$ représentent des séquences d'embeddings de mots et des séquences d'embeddings de catégories

Batcher des telles données fonctionnerait simplement si tous les exemples avaient exactement le même nombre de mots :




[3]Avec la plupart des librairies modernes, les paramètres du modèle résident en permanence sur le GPU, ce sont les vecteurs $\mathbf{x}$ de données et $\mathbf{y}$ de prédictions qui sont transférés.

Or la réalité est différente : les textes ou les phrases sont de **longueur variable**. De telle sorte que certains éléments du batch (plus courts) ont des vecteurs manquants (par comparaison avec les éléments les plus longs) :


Pour contourner ce problème, il est d'usage :

- D'ajouter des vecteurs factices (en général nuls) dans les positions manquantes. (Technique de **padding**)
- De tronquer les exemples du batch de telle sorte que toutes les séquences de mots aient la même longueur. Les mots supplémentaires sont simplement ignorés. (Technique de **troncation**)

Plus généralement, le calcul par batchs est facile à mettre en oeuvre si la structure du réseau est identique (statique) pour tous les exemples du jeu de données. La mise en oeuvre du calcul par batchs est plus complexe lorsque la structure du réseau varie d'exemple à exemple (on parle de modèle dynamique). C'est ce dernier cas qui est classique pour la modélisation du langage.

Lorsque la structure de données est plus complexe qu'une simple séquence, le problème de batching n'a pas de solution généralement admise par la communauté. Par exemple si il s'agit d'arbres, une méthode consiste à coder l'arbre par une séquence d'actions de shift et de reduce (cf chapitre de Parsing) ce qui permet de se ramener au cas séquentiel.

## 5.5 Préparation des données

On remarque que modéliser des données langagières écrites demande de déployer un certain nombre de traitements sujets à erreur, commme par exemple le codage du vocabulaire sur des entiers, les traitements par batchs.... On donne ici quelques indications sur le workflow typique lié au prétraitement des données d'un modèle pour le TAL ainsi que quelques points de repères pour éviter les erreurs.

Il est commode de voir un jeu de données comme un fichier tabulaire de type `csv`. Chaque colonne correspond à une variable (ou champ) et chaque ligne à une observation dans les données, comme par exemple :

| Source | Phrase | Article cible |
|---|---|---|
| Wikipedia | Le chat mange la souris | chat |
| Wikipedia | Trump est président des USA | Trump |
| ... | ... | ... |

À partir d'un jeu de données, on peut identifier différentes étapes de traitement qui reviennent régulièrement :

- Pour les colonnes qui contiennent du texte libre, il faut d'abord penser à donner une segmentation à ce texte.
- Le codage de chaque colonne sur des entiers. Il s'agit d'identifier la totalité du vocabulaire utilisé dans la colonne et de définir une fonction qui envoie les symboles sur des entiers. Il faut se poser la question des symboles inconnus (mots du vocabulaire d'une langue mais non vus dans les données) et éventuellement prévoir des symboles artificiels pour coder les mots inconnus, des symboles de début de phrase, de fin de phrase...
- Il faut le cas échéant charger un dictionnaire d'embeddings qui permet d'associer les mots à des vecteurs de type embedding
- Il faut le cas échéant penser à structurer le jeu de données en batchs. Si les données sont des phrases de longueur variables, il faut également réaliser des opérations de padding et de troncation

L'ensemble de ces opérations peut soit se programmer manuellement soit s'appuyer sur des librairies comme `torchtext`, ce qui permet d'éviter d'introduire des erreurs potentiellement difficiles à détecter lors de ces étapes de préparation des données

## 5.6 Deep Learning sur le cloud

Comme les modèles de deep learning mobilisent des moyens de calcul importants, on réalise fréquemment les calculs sur le "cloud", c'est-à-dire sur des serveurs de calcul distants qui comportent un bon nombre de CPU, une mémoire suffisante et parfois des GPU.

Parmi les plus connus on citera par exemple Amazon Web Services (`https://aws.amazon.com`) et Google Colab (`https://colab.research.google.com`). Dans le cadre de ce cours on utilisera Kaggle (`https://www.kaggle.com`) qui permet également d'évaluer des projets. Une compétition

KAGGLE est organisée autour d'un jeu de données sur lequel il faut résoudre un problème d'apprentissage. Dans le cadre de ce cours, la procédure pour participer est la suivante :

- On reçoit une invitation à la compétition par email. Cette invitation renseigne le lien de la compétition.
- Depuis la page d'accueil de la compétition, il faut créer une machine virtuelle (appelée **Kernel**). Dans le cadre de ce cours, on créera toujours un notebook python.
  - Le kernel est un notebook python. On peut entrer du code dans la partie notebook. On dispose d'une console qui simule un shell et on peut monitorer ou configurer l'usage de la machine virtuelle en consultant les onglets appropriés.
  - Comme le code est entré dans un navigateur web, il y a risque de le perdre lorsqu'on ferme la fenêtre. Il est recommandé de cliquer régulièrement sur le bouton commit (en haut à droite) pour sauvegarder.
  - Lorsqu'on utilise un kernel dans une compétition, les données liées à la compétition sont préinstallées dans un répertoire dédié (sous `../input/`). Il s'agit de données d'entrainement et de données de test. Ces dernières sont vierges d'annotations.
  - Le code réalisé dans un kernel sera destiné essentiellement à créer un fichier `csv` qui représente les prédictions du modèle d'apprentissage. Ce fichier aura en général la forme suivante :

| idx | prédiction |
|---|---|
| 1 | A |
| 2 | B |
| 3 | A |
| ... | |

- Lorsque le noyau a produit un fichier `csv` satisfaisant, on peut décider de soumettre le résultat à la compétition. Il faut pour cela :
  - Effectuer un commit
  - se reconnecter à la page d'accueil[4], suivre l'onglet *Kernels*, cliquer sur le nom du kernel dont vous souhaitez soumettre les prédictions et enfin cliquer sur le bouton de soumission dans la page résumant ce kernel (onglet ouptut data).

[4]Depuis l'éditeur de code, cliquer sur le **K** bleu en haut à gauche.

- Il est possible d'observer facilement la structure des jeux de données utilisés. Depuis la page d'accueil de la compétition, suivre l'onglet *Data* et explorer.

Remarquons que les calculs exécutés dans un kernel sont limités. Un kernel doit terminer toute exécution en moins de six heures et la quantité de données qu'il est possible d'écrire dans le kernel est de l'ordre de quelques Gigabytes (5Gb). Il est également possible de choisir le processeur utilisé : on peut utiliser soit 4 CPUs soit 1 GPU pour réaliser les calculs.

# Chapter 6

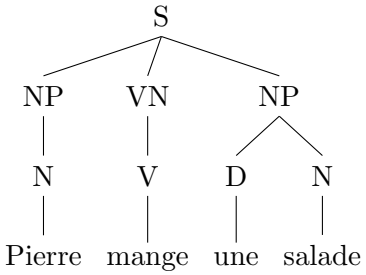
# Représentations

En TAL, on fait de l'analyse syntaxique et de la syntaxe non pas pour décider de la grammaticalité des phrases mais pour leur donner une structure destinée à en construire le sens.

La problématique d'analyse syntaxique en TAL diffère de la problématique d'analyse syntaxique en compilation où seuls les programmes syntaxiquement corrects sont analysés sémantiquement. En TAL il y a volonté de robustesse, c'est-à-dire que tout énoncé doit recevoir une interprétation sémantique.

Deux types de représentations syntaxiques sont utilisées : les représentations dites en dépendances et les représentations dites en constituants.

Les représentations en constituants sont utilisées traditionellement en linguistique pour représenter la strucuration des phrases en groupes de mots :




Elles sont aussi utilisées explicitement pour aider à la construction de la représentation sémantique en indiquant un parenthésage qui permet de décider comment associer les sous-phrases pour calculer compositionnellement le sens (6.1)

Les arbres en dépendances sont en général utilisés directement (en tous cas les relations qu'ils représentent) pour extraire un contenu sémantique de

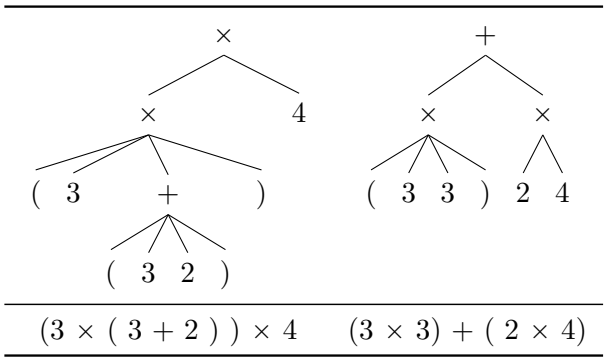
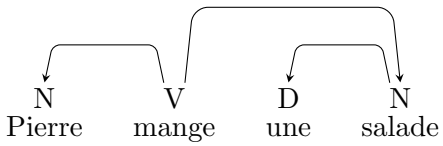



Figure 6.1: Le parenthésage permet de changer l'interprétation de la phrase

la phrase, comme par exemple pour interfacer avec une base de données.




**Arbres en constituants** Les arbres en constituants sont généralement vus comme des traces de dérivation de grammaires CFG. En général la grammaire n'est pas définie explicitement car pour un ensemble de symboles non terminaux $N$ et un ensemble de mots $W$ on autorise toute règle de la forme :

$$\begin{aligned} X_i &\rightarrow X_j X_k \qquad \{X_i, X_j, X_k\} \in N^3 \\ X_i &\rightarrow X_k X_j \qquad \{X_i, X_j, X_k\} \in N^3 \\ X_i &\rightarrow w \qquad w \in W \end{aligned}$$

Dans ce contexte le problème d'analyse syntaxique consiste à déterminer le (ou les) arbres d'analyse les plus plausibles parmi l'ensemble exponentiel des parenthésages possibles de la phrase.

Voir les arbres en constituants comme des traces de dérivations d'une grammaire libre de contexte (même laxiste) permet de préserver les définitions valables pour PCFG, qui sert de référentiel commun utilisé pour ordonner les différentes arbres d'analyse est PCFG à partir duquel sont dérivés les modèles d'apprentissage alternatifs.

## 6.1 Arbres de dépendances

# Chapter 7

# Analyse syntaxique en constituants

L'analyse en constituants est une représentation de la structure des phrases relativement ancienne en linguistique formelle : les constituants sont un parenthésage des mots de la phrase qui peut servir à en calculer sa sémantique à l'aide de méthodes compositionnelles.

Traditionnellement les représentations en constituants sont vues comme des traces des opérations de réécriture successives réalisées par une grammaire générative indépendante du contexte. Une grammaire formelle définit un langage, c'est-à-dire un ensemble de phrases dites grammaticales qu'il est possible de dériver de l'axiome par réécritures successives. Les séquences de mots qu'il n'est pas possible d'engendrer à l'aide de la grammaire sont dites non grammaticales.

Dans ce qui suit, on s'intéresse au problème d'analyse dit robuste. C'est-à-dire que la notion de grammaticalité est délaissée au profit d'un mécanisme d'analyse statistique capable de donner une structure en arbre pour toute séquence de mots.

De plus on sait que le problème d'ambiguité est massif pour l'analyse syntaxique, les causes proviennent essentiellement de problèmes liés à l'attachement de syntagmes prépositionnels et de résolution de portée de la coordination. C'est pour apporter une méthode de déterminisation (ou de choix) des analyses que l'on s'intéresse à ajouter une composante statistique destinée à la désambiguisation.

On introduit ici quelques algorithmes d'analyse syntaxique augmentés de pondérations statistiques calculées par des modèles discriminants. Ces algorithmes supposent que les arbres à manipuler sont sous-forme binaire

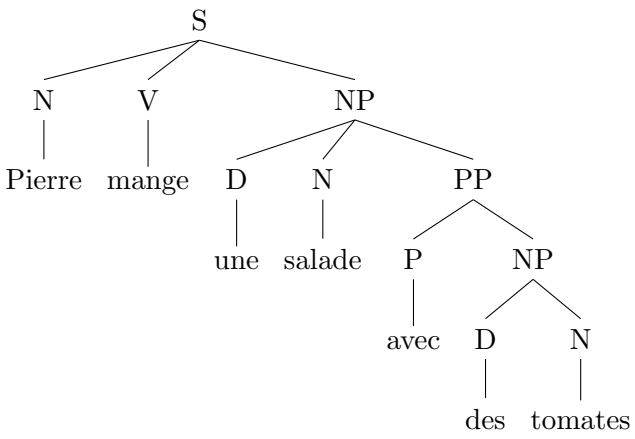



Figure 7.1: Un arbre de constituants

(en forme normale de Chomsky). On commence donc par introduire ce type de transformation avant de présenter l'algorithme de Cocke Kasami Younger comme une extension de l'algorithme de Viterbi, l'algorithme de Knuth comme une extension de l'algorithme de Dijkstra et finalement les méthodes d'analyse en constituants par transitions.

## 7.1 Forme normale

Dans ce qui suit, on suppose manipuler des arbres de constituants engendrés par une grammaire en forme normale de Chomsky (CNF) : les arbres du treebank original sont transformés en arbres binaires et les résultats internes d'analyse sont débinarisés en dernière étape. L'usage de cette transformation permet de simplifier la conception des algorithmes d'analyse. Il ne s'agit que de construire des arbres binaires.

Une grammaire en forme normale de Chomsky est une grammaire dont les règles ont nécessairement une des formes suivantes :

$$A \rightarrow B \quad C \tag{7.1}$$

$$A \rightarrow a \tag{7.2}$$

où $A, B, C \in N$ et $a \in \Sigma$.

**Longueur des dérivations** On peut remarquer que le nombre d'étapes de dérivation pour construire une séquence de terminaux de longueur $n \leqslant 1$

à partir de l'axiome vaut $\eta = 2n - 1$ pour une grammaire en forme normale de Chomsky. On montre cela en considérant deux cas. Dans le cas où $n = 1$, la seule opération de réécriture à partir de l'axiome fait intervenir la règle (7.2) $\eta = 1 = 2-1$. Dans le cas où $n > 1$, la première opération de réécriture à partir de l'axiome fait intervenir la règle (7.1), ce qui crée une séquence de symboles de longeur 2. Toute opération de réécriture qui fait intervenir à nouveau la règle (7.1) augmente la longueur de la séquence de symboles de 1. Par conséquent il faut $n-1$ étapes de dérivation pour produire une séquence de symboles non terminaux de longueur $n$. Comme il faut nécessairement utiliser $n$ fois la règle (7.2) pour produire les terminaux, on a au total $2n-1$ étapes de dérivation pour produire la séquence à l'aide d'une grammaire en forme normale de Chomsky.

Cette propriété sur la longueur des dérivations est régulièrement réutilisée (parfois de manière très indirecte) pour borner le nombre d'étapes de dérivation des algorithmes d'analyse syntaxique présentés dans ce document.

**Transformation des arbres** Une des première étapes de traitement lors de la mise au point d'un analyseur statistique en constituants consiste à transformer le treebank original en un treebank dont les arbres représentent des dérivations d'une grammaire en forme normale de Chomsky. Pour ce faire, on suppose en général que les arbres du treebank ont tous le même symbole racine et que ce symbole n'est pas utilisé pour catégoriser des noeuds autres que la racine.

Les deux opérations de transformation qu'il faut réaliser consistent (a) à supprimer les branches unaires qui n'introduisent pas de non terminaux et (b) à réduire les branchements $n$-aires ($n > 2$).

Soit la branche unaire qui est la séquence de symboles $X_1, X_2 \dots X_d$ telle que $X_i \in N$, l'algorithme de fermeture des symboles unaires consiste à remplacer cette branche par le nouveau symbole non terminal $X_1 + X_2 + \dots + X_n$ (où + représente la concaténation). On illustre le procédé sur l'exemple donné en Figure 7.2.

Lorsque l'arbre en constituants présente des sous-arbres qui instancient des règles de la forme $r_0 = X \rightarrow A_1, A_2 \dots A_n$ avec $n > 2$, l'algorithme de binarisation consiste à (1) remplacer $r_0$ par la règle $r_1 = X \rightarrow A_1 A_2 + \dots + A_n$ en créant le nouveau non terminal $A_2 + \dots + A_n$ et à (2) introduire la règle $r_2 = A_2 + \dots + A_n \rightarrow A_2, \dots, A_n$ sous ce nouveau non terminal. La transformation est récursive et s'applique à nouveau sur $r_2$ puis sur toute nouvelle $r_i$ nouvellement introduite dans l'arbre d'arité supérieure à 2. Cette méthode s'appelle la binarisation droite. On illustre le procédé sur l'exemple

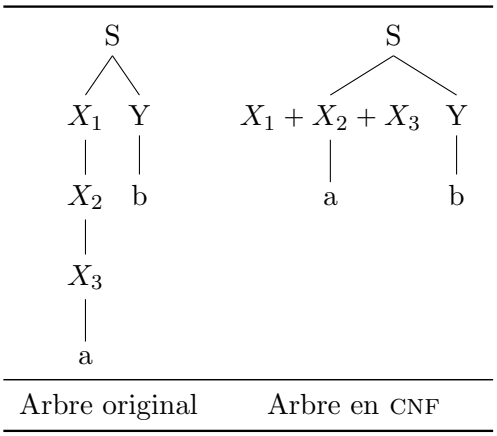



Figure 7.2: Exemple de réduction des règles unaires.

donné en Figure 7.3.

Alternativement, on peut binariser par la gauche. Dans ce cas on remplace $r_0 = X \to A_1, \ldots, A_{n-1}, A_n$ par une règle de la forme $r_1 = X \to A_1 + \ldots + A_{n-1}, A_n$ et en introduisant la règle $r_2 = A_1 + \ldots + A_{n-1} \to A_1 \ldots A_{n-1}$ et ce recursivement jusqu'à ce que la règle $r_i$ nouvellement introduite ait une arité de 2. Cette méthode s'appelle la binarisation gauche et on illustre le procédé sur l'exemple donné en Figure 7.3.

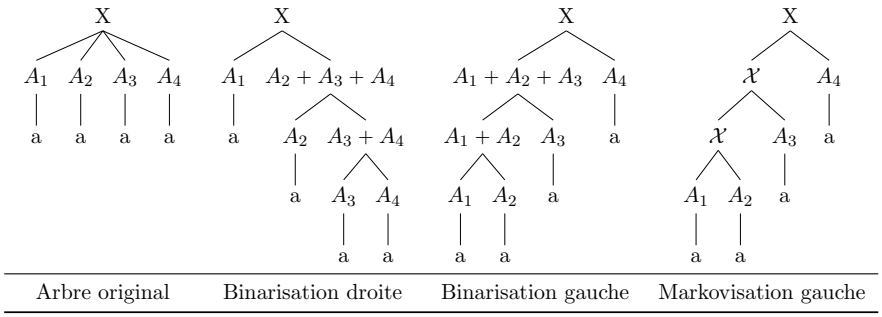



Figure 7.3: Exemples de binarisations

Des variantes pratiques de binarisation sont utilisées sous le nom de markovisation des arbres. Celles-ci consistent à réduire la quantité de symboles non terminaux artificiels créés par la procédure de binarisation. On peut par exemple décider d'utiliser un symbole artificiel $\mathcal{X}$ correspondant à

la racine du sous arbre binarisé (c'est-à-dire au symbole en partie gauche de la règle $r_0$) en lieu et place des concaténations de symboles réalisées par la procédure exacte. On illustre le procédé sur l'exemple donné en Figure 7.3.

## 7.2 Algorithmes de programmation dynamique et forêt d'analyse

**Représentation** La représentation supposée par les algorithmes ascendants de Knuth et CKY est fondée sur la notion d'empan (*span*). Une phrase $w_1 \dots w_n$ est représentée par une séquence d'états $0 \dots n$ reliés par des transitions $(i, i+1)$ étiquetées par le mot $w_{i+1}$. On notera une telle transition par le triplet $\langle i, i+1, w_{i+1} \rangle$.

Par exemple, on représente la phrase *le chat dort* comme suit :

$$0 \quad \text{le} \quad 1 \quad \text{chat} \quad 2 \quad \text{dort} \quad 3$$

et on représente que le mot *chat* couvre l'empan $(1, 2)$ par le triplet $\langle 1, 2, \mathit{chat} \rangle$. La notion de triplet (ou d'item d'analyse) se généralise également pour représenter des empans couverts par des symboles non terminaux lors d'une analyse. Par exemple on notera $\langle 0, 2, NP \rangle$ pour indiquer qu'un groupe nominal couvre l'empan $(0, 2)$. De manière générale un item d'analyse a la forme :

$$\langle i, j, X \rangle$$

où $i, j$ représentent l'empan de l'item et $X$ est un mot ($X \in \Sigma$) ou un symbole non terminal ($X \in N$).

Les algorithmes asecendants combinent successivement des items contigus à l'aide d'une règle de réduction unique :

$$\frac{\langle i, k, Y_1 \rangle \quad \langle k, j, Y_2 \rangle}{\langle i, j, X \rangle} \qquad X \in N \tag{7.3}$$

qui indique que l'arbre de racine $Y_1$ qui couvre l'empan $(i, k)$ se combine avec l'arbre de racine $Y_2$ qui couvre l'empan contigu $(k, j)$ pour former l'arbre de racine $X$ qui couvre l'empan $(i, j)$. On peut résumer explicitement les opérations réalisées par ce type d'algorithme par le système déductif donné en figure 7.4.

On peut remarquer que contrairement à la présentation classique de CKY, la règle de réduction donnée en (7.3) ne suppose pas de règle de grammaire explicite (comme $X \to Y_1 Y_2$). Au contraire tout symbole $X \in N$ peut servir de symbole racine pour le nouvel arbre ainsi créé.

$$\begin{array}{lll} \textbf{init} & \langle i, i+1, w_{i+1}\rangle & 0 \leqslant i < n \\ \textbf{reduce} & \frac{\langle i,k,Y_1\rangle \quad \langle k,j,Y_2\rangle}{\langle i,j,X\rangle} & X \in N \\ \textbf{but} & \langle 0, n, X\rangle & \end{array}$$

On suppose une phrase : $w_1 \dots w_n$

Figure 7.4: Système de déduction pour l'analyse ascendante

**Représenter l'ambiguité** Une telle procédure d'analyse crée de l'ambiguité. Pour une même séquence de mots $w_0 \dots w_n$, on s'attend en théorie à un nombre exponentiel d'arbres d'analyse. La **forêt** d'analyse est une structure de données qui permet de représenter formellement un tel ensemble d'analyses de manière compacte. On illustre en Figure 7.5 une forêt (à droite) qui encode les deux analyses données à gauche. Formellement, une forêt correspond à un hypergraphe $H = \langle V, E\rangle$ dont les noeuds sont les items d'analyse $\langle i, j, X\rangle$ et dont les hyperarcs sont des couples $(h, v_1 \dots v_k)$ où $h \in V$ et où $v_i \in V$. Ainsi en figure 7.5, la forêt comporte six noeuds

$$V = \{\langle 0,3,X\rangle, \langle 0,2,X\rangle, \langle 1,3,X\rangle, \langle 0,1,w_1\rangle, \langle 1,2,w_2\rangle, \langle 2,3,w_3\rangle\}$$

et les hyperarcs :

$$\begin{array}{rcl} E & = & \{(\langle 0,3,X\rangle, \langle 0,2,X\rangle\langle 2,3,w_3\rangle), \\ & & (\langle 0,3,X\rangle, \langle 0,1,w_1\rangle\langle 1,3,X\rangle), \\ & & (\langle 0,2,X\rangle, \langle 0,1,w_1\rangle\langle 1,2,w_2\rangle), \\ & & (\langle 1,3,X\rangle, \langle 1,2,w_2\rangle\langle 2,3,w_3\rangle), \\ & & (\langle 0,1,w_1\rangle, \epsilon), \\ & & (\langle 1,2,w_2\rangle, \epsilon), \\ & & (\langle 2,3,w_3\rangle, \epsilon)\} \end{array}$$

On va voir que les algorithmes de Knuth et CKY sont deux méthodes de construction d'une forêt d'analyse (ou d'hypergraphe pondéré) qui utilisent le système de déduction donné en figure 7.4) mais qui construisent les noeuds et les hyperarcs en utilisant un ordre différent. Pour celà on commence par définir formellement la notion d'hypergraphe telle qu'elle est utilisée en analyse syntaxique [**?**, **?**].

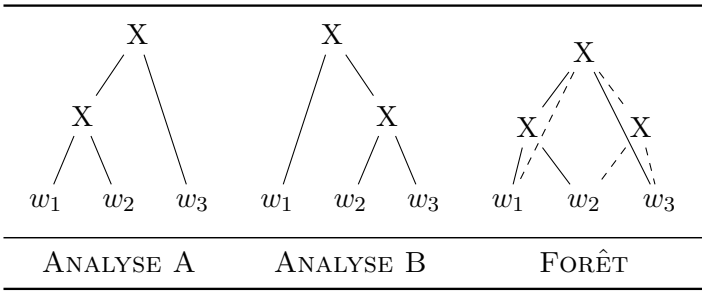



Figure 7.5: Représentation de l'ambiguité

### 7.2.1 Notion d'hypergraphe (et de forêt)

**Définition 7.1 (Hypergraphe)** *Un hypergraphe (orienté) $H = \langle V, E \rangle$ est un couple où $V$ est un ensemble de noeuds et $E$ est un ensemble d'hyperarcs. Un hyperarc $e \in E$ est un couple $e = \langle h, v_1 \dots v_k \rangle$ où $h \in V$ est la tête et $v_1 \dots v_k \in V^k$ est la queue. Dans le cas où l'hypergraphe est pondéré on définit une fonction de score $s : E \mapsto \mathbb{R}$ qui donne un score à chaque hyperarc. De plus on note $head(e)$ la fonction qui renvoie la tête $h$ d'un hyperarc $e$ et $tail(e)$ la fonction qui renvoie la queue $v_1 \dots v_k$ d'un hyperarc $e$. On note $|e| = k$ l'arité d'un hyperarc $e = \langle h, v_1 \dots v_k \rangle$.*

**Définition 7.2 (Arcs entrants)** *L'ensemble $AE(v)$ des arcs entrants sur un noeud $v$ est l'ensemble $\{e \in E | v = head(e)\}$.*

**Définition 7.3 (Arcs sortants)** *L'ensemble $AS(v)$ des arcs sortants du noeud $v$ est l'ensemble $\{e \in E | v \in tail(e)\}$.*

On peut également définir une extension de la notion de chemin pour un graphe qui est appelée ici dérivation dans le cas des hypergraphes. Cette notion correspond également à la notion d'arbre syntaxique.

**Définition 7.4 (Dérivation)** *Une dérivation $D(v)$ enracinée au noeud $v \in V$ se définit récursivement comme suit :*

- **Base:** *Si $e \in AE(v)$ et que $|e| = 0$ alors $D(v) = \langle v, \epsilon \rangle$ est une dérivation de $v$.*
- **Récurrence:** *Si $e \in AE(v)$ et $|e| = k > 0$ alors $D(v) = \langle v, D(v_1) \dots D(v_k) \rangle$ est une dérivation de $v$ $(1 \leqslant i \leqslant k)$.*

Comme plusieurs dérivations sont possibles pour un même noeud $v$ (cas d'ambiguité), on notera $\mathcal{D}(v)$ l'ensemble des dérivations d'un noeud $v$. Voyons maintenant comment pondérer les dérivations.

**Définition 7.5 (Fonction de pondération)** *La fonction de pondération $s : E \mapsto \mathbb{R}$ est une fonction qui attribue un score sous forme de valeur réelle à un hyperarc $e$.*

Dans la pratique pour un hyperarc $e = \langle h, v_1 \dots v_k \rangle$, la fonction de pondération attribue un score à un hyperarc en ayant accès à la valeur de chacun des noeuds. Autrement dit, elle prend la forme $s(h, e_1, \dots e_k)$.

Notons que dans le cas d'une PCFG, $s(e)$ sera une probabilité de la forme $P(\alpha|A)$ pour une règle $A \rightarrow \alpha$ alors que pour un modèle discriminant $s(e)$ se reformule avec un paramètre additionel $\mathbf{x}$ qui représente la séquence de mots. Dans ce dernier cas la fonction prend la forme $s(e, \mathbf{x}) = \mathbf{w}^T \mathbf{\Phi}(e, \mathbf{x})$ pour un perceptron structuré et $s(e, \mathbf{x}) = \psi(e, \mathbf{x}) = \exp(\mathbf{w}^T \mathbf{\Phi}(e, \mathbf{x}))$ pour un CRF.

**Définition 7.6 (Pondération d'une dérivation)** *Le poids $w(D(v))$ d'une dérivation $D(v)$ enracinée au noeud $v$ se définit récursivement comme suit :*

- **Base:** *Si $e \in AE(v)$ et que $|e| = 0$ alors $w(D(v)) = 1$.*
- **Récurrence:** *Si $e \in AE(v)$ et que $|e| = k > 0$ alors $w(D(v)) = s(e) \times \prod_{i=1}^{k} w(D(v_i))$ $(1 \leqslant i \leqslant k)$.*

où les scores des $D_i$ dérivations qui mènent à $e$ sont multipliées avec le score $s(e)$ de $e$[1].

**Définition 7.7 (Poids maximal d'un noeud)** *Le poids maximal d'un noeud $\delta(v)$ est le poids de la meilleure dérivation qui mène à $v$ :*

$$\delta(v) = \begin{cases} 1 & \textit{si } v \textit{ est un noeud source} \\ max_{D(v) \in \mathcal{D}(v)} w(D(v)) & \textit{sinon} \end{cases}$$

**Problème d'analyse syntaxique** Les deux algorithmes que nous présentons ci-dessous permettent de résoudre le problème d'analyse syntaxique d'une phrase $w_1 \dots w_n$. La résolution consiste à extraire la dérivation de poids maximal de tout item $\langle 0, n, X \rangle$ $(X \in N)$. En considérant l'ensemble

[1] Il s'agit en général d'une multiplication mais parfois d'une addition selon le semi-anneau considéré (cas du perceptron structuré par exemple).

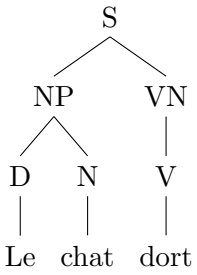
$\mathcal{D}(\mathbf{x}) = \bigcup_{X \in N} \mathcal{D}(\langle 0, n, X \rangle)$ de toutes les dérivations qui couvrent la phrase, les algorithmes qui suivent permettent de résoudre le problème :

$$\hat{D}(v) = \underset{D(v) \in \mathcal{D}(\mathbf{x})}{\operatorname{argmax}} \; w(D(v)) \tag{7.4}$$

On peut finalement remarquer que les définitions qui précèdent ont pour conséquence que le score d'une dérivation $D$ se décompose comme le produit des scores de ses hyperarcs :

$$w(D(v)) = \prod_{e \in D(v)} s(e) \tag{7.5}$$

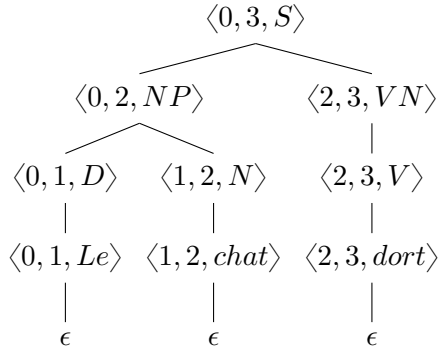
**Exemple de dérivation** Une dérivation est la contrepartie d'un arbre d'analyse (en général partiel) pour le traitement algorithmique. Ainsi pour l'arbre:




La dérivation sera :

$(\langle 0,3,S\rangle, (\langle 0,2,NP\rangle, (\langle 0,1,D\rangle, (\langle 0,1,Le\rangle, \epsilon))(\langle 1,2,N\rangle, (\langle 1,2,chat\rangle, \epsilon)))(\langle 2,3,VN\rangle, (\langle 2,3,V\rangle, (\langle 2,3,dort\rangle, \epsilon))))$

ce que l'on peut alternativement écrire :




La pondération d'une dérivation consiste à faire le produit (selon le demi-anneau) des hyperarcs qui connectent les noeuds. On peut voir à partir de l'exemple que le procédé est analogue à la pondération d'une dérivation PCFG.

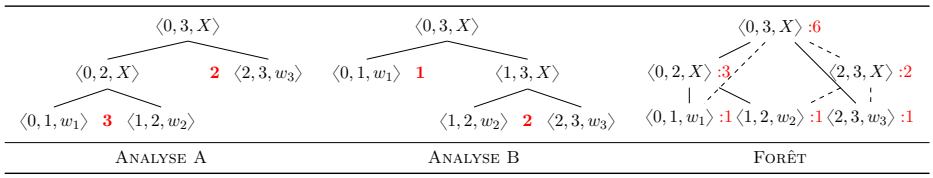



Figure 7.6: Calcul de $\delta(v)$ dans une forêt

**Exemple de calcul de $\delta(v)$** L'exemple donné en figure 7.6 illustre les interactions entre dérivations dans une même forêt notamment pour le calcul de $\delta(v)$. On donne en rouge, dans les deux figures de gauche, une pondération pour les différents hyperarcs. Celui-ci correspond à un score qui serait produit par une méthode d'apprentissage. La figure de droite, qui illustre une forêt à proprement parler, donne les $\delta(v)$ pour chacun des noeuds de cette forêt.

### 7.2.2 Algorithme CKY

L'algorithme CKY réalise les réductions en commençant par combiner les items qui couvrent les petits empans, en créeant itérativement des items qui couvrent des empans de plus en plus grands et ce jusqu'à créer un item qui couvre toute la phrase. Formellement cet algorithme crée la forêt en suivant un ordre topologique sur les noeuds de cette forêt. On définit l'ordre topologique sur un hypergraphe comme suit.

**Définition 7.8 (Graphe projeté)** *Le graphe projeté d'un hypergraphe orienté $H = \langle V, E\rangle$ est le graphe orienté $G = \langle V, E'\rangle$ tel que $E' = \{(u, v)|\exists e \in AE(v) \wedge u \in tail(v)\}$. Si $H$ est acyclique alors $G$ est acyclique.*

**Définition 7.9 (Ordre topologique)** *Un ordre topologique pour $H$ est tout ordonnancement total de ses noeuds qui est un ordre topologique pour son graphe projeté $G$.*

L'algorithme prend alors la forme générale donnée en algorithme 24. L'algorithme associe à chaque noeud $v$ le poids $\delta(v)$ de la meilleure dérivation qui mène à $v$.

L'algorithme est également souvent présenté en explicitant la table de programmation dynamique et en fixant un ordre topologique qui tire parti de la longueur des empans créés, comme illustré en algorithme 25. Cette dernière version permet de voir que la complexité de l'algorithme est en $\mathcal{O}(n^3)$ car il comporte trois boucles de longueur $n$ dans le pire des cas.

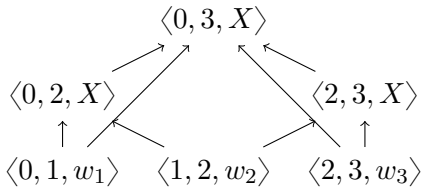



Figure 7.7: Graphe projeté correspondant à la forêt de Figure 7.6

**Algorithm 24** Algorithme Viterbi-CKY

**function** VITERBI-CKY($w_1 \dots w_n$)
  **for** $0 \leqslant i < n$ **do** ▷ Initialisation
    $v \leftarrow \langle i, i+1, w_{i+1} \rangle$
    $\delta(v) \leftarrow 1$
  **end for**
  **for all** $v \in V$ en suivant un ordre topologique **do** ▷ Récurrence
    $\delta(v) \leftarrow 0$
    **for** $e \in AE(v)$ **do**
      $\langle h, v_1 \dots v_k \rangle \leftarrow e$
      $\delta(v) \leftarrow \max\left(\delta(v), s(e) \times \prod_{i=1}^{k} \delta(v_i)\right)$
    **end for**
  **end for**
**end function**

**Algorithm 25** Algorithme Viterbi-CKY (version classique binaire)

**function** VITERBI-CKY($w_1 \dots w_n$)
  **for** $0 \leqslant i < n$ **do** ▷ Initialisation
    $v \leftarrow \langle i, i+1, w_{i+1} \rangle$
    $\delta(v) \leftarrow 1$
  **end for**
  **for** $1 < span \leqslant n$ **do** ▷ Récurrence
    **for** $0 \leqslant i \leqslant n - span$ **do**
      $j \leftarrow i + span$
      **for** $X \in N$ **do** ▷ Boucle sur non terminaux
        $v \leftarrow \langle i, j, X \rangle$
        $\delta(v) \leftarrow 0$
        **for** $i < k < j$ **do** ▷ Applications de la règle de réduction
          **for** $(Y_1, Y_2) \in N \times N$ **do**
            $v_1, v_2 \leftarrow \langle i, k, Y_1 \rangle, \langle k, j, Y_2 \rangle$
            $\delta(v) \leftarrow \max(\delta(v), \times(s(v, v_1, v_2), \delta(v_1), \delta(v_2))$
          **end for**
        **end for**
      **end for**
    **end for**
  **end for**
**end function**

**Exercice 7.1 (Historique)** *Donner une extension de l'algorithme 25 qui permet de renvoyer la dérivation de poids maximal.*

**Estimation de paramètres** L'algorithme CKY décompose les scores des dérivations comme suit :

$$w(D) = \prod_{e \in D} w(e) \tag{7.6}$$

ce type de décomposition permet d'entraîner des modèles structurés à large marge très facilement. On donne un exemple pour l'algorithme du perceptron en Algorithme 26 pour lequel (7.6) s'instancie comme suit :

$$w(\mathbf{x}, \mathbf{D}) = \sum_{e \in \mathbf{D}} \mathbf{w}^T \mathbf{\Phi}(e, \mathbf{x})$$

**Algorithm 26** Perceptron pour estimer les paramètres d'un système d'analyse CKY

**function** CKY-PERCEPTRON($(\mathbf{x}_i, \mathbf{D}_i)_{i=1}^N$,$E$)
  $\mathbf{w} \leftarrow \bar{\mathbf{0}}$
  **for** $1 \leqslant e \leqslant E$ **do**
    **for** $1 \leqslant i \leqslant N$ **do**
      $\hat{\mathbf{D}} \leftarrow \text{argmax}_{\mathbf{D} \in \mathcal{D}(\mathbf{x})} \sum_{e \in \mathbf{D}} \mathbf{w}^T \mathbf{\Phi}(e, \mathbf{x}_i)$ ▷ Parsing
      **if** $\hat{\mathbf{D}} \neq \mathbf{D}_i$ **then**
        $\mathbf{w} \leftarrow \mathbf{w} + \left[\sum_{e \in \mathbf{D}} \mathbf{\Phi}(e, \mathbf{x}_i) - \sum_{\hat{e} \in \hat{\mathbf{D}}} \mathbf{\Phi}(\hat{e}, \mathbf{x}_i)\right]$
      **end if**
    **end for**
  **end for**
  **return w**
**end function**

L'estimation des paramètres pour un modèle CRF est plus complexe. Cela demande de déployer une contrepartie de l'algorithme *forward backward* pour calculer le second terme du gradient. Il existe un algorithme appelé dedans-dehors (*inside-outside*) qui permet de calculer ce terme par programmation dynamique à partir de toute la forêt d'analyse. Cependant, il est en général peu utilisé tel quel car trop coûteux en temps de calcul pour des cas d'utilisation réels.

### 7.2.3 Algorithme de Knuth

On présente ici l'algorithme de Knuth en utilisant le même système déductif que précedemment (Figure 7.4). L'algorithme de Knuth, ou algorithme de recherche du meilleur d'abord peut être vu comme une reformulation de l'algorithme de Dijkstra pour la recherche de court chemin sur des graphes pondérés. On peut faire l'analogie entre cet algorithme et le problème traditionnel de recherche de court chemin : lorsqu'on combine deux chemins on s'attend à ce que le score du chemin global augmente.

**Algorithm 27** Algorithme de Knuth

```
function KNUTH-PARSER(w_1 ... w_n)
    for 0 ⩽ i < n do                                   ▷ Initialisation
        v ← ⟨i, i + 1, w_{i+1}⟩
        δ(v) ← 1̄
        RÉORDONNERCLÉ(Q,v)
    end for
    rest(e) ← |e|   (∀e ∈ E)                           ▷ Init arités hyperarcs
    while Q ≠ ∅ do
        v ← EXTRACT-MIN(Q)
        for all e ∈ AS(v) do
            h, (v_1, v_2) ← e
            rest(e) ← rest(e) − 1
            if rest(e) = 0 then
                δ(h) ← max(δ(h), ×(s(v, v_1, v_2), δ(v_1), δ(v_2)))
                RÉORDONNERCLÉ(Q,h)
            end if
        end for
    end while
end function
```

Informellement l'algorithme de Knuth suppose que le score d'une dérivation augmente lorsqu'on la combine avec une autre dérivation. Ce type de comportement est typiquement celui du demi-anneau tropical (et plus généralement des demi-anneaux qui satisfont la condition dite de supériorité) Ce sont des poids qui se dérivent naturellement de probabilités ($w = -\log(p)$) ou à des sorties de modèles de régression logistique multinomiale ou de réseaux de neurones à couche de sortie softmax.

Plus formellement, la fonction de pondération doit satisfaire les conditions dites de monotonie et de supériorité.

**Algorithm 28** Algorithme de Knuth (version binaire)

**function** KNUTH-PARSER($w_1 \dots w_n$)
  **for** $0 \leqslant i < n$ **do** ▷ Initialisation
    $v \leftarrow \langle i, i+1, w_{i+1} \rangle$
    $\delta(v) \leftarrow \bar{1}$
    RÉORDONNERCLÉ($Q$,$v$)
  **end for**
  $rest(e) \leftarrow |e| \quad (\forall e \in E)$ ▷ Init arités hyperarcs
  **while** $Q \neq \varnothing$ **do**
    $v \leftarrow$ EXTRACT-MIN($Q$)
    $\langle i, k, L \rangle \leftarrow v$
    **for** $k \leqslant j \leqslant n$ **do**
      **for all** $(X_1, X_2) \in N \times N$ **do**
        $e \leftarrow \langle i, j, X_1 \rangle, \langle i, k, L \rangle \langle k, j, X_2 \rangle$
        $rest(e) \leftarrow rest(e) - 1$
        **if** $rest(e) = 0$ **then**
          $\delta(h) \leftarrow \max(\delta(h), \times(s(v, v_1, v_2), \delta(v_1), \delta(v_2)))$
          RÉORDONNERCLÉ($Q$,$h$)
        **end if**
      **end for**
    **end for**
    $\langle k, j, L \rangle \leftarrow v$
    **for** $0 \leqslant i \leqslant k$ **do**
      **for all** $(X_1, X_2) \in N \times N$ **do**
        $e \leftarrow \langle i, j, X_1 \rangle, \langle i, k, X_2 \rangle \langle k, j, L \rangle$
        $rest(e) \leftarrow rest(e) - 1$
        **if** $rest(e) = 0$ **then**
          $\delta(h) \leftarrow \max(\delta(h), \times(s(v, v_1, v_2), \delta(v_1), \delta(v_2)))$
          RÉORDONNERCLÉ($Q$,$h$)
        **end if**
      **end for**
    **end for**
  **end while**
**end function**

**Définition 7.10 (Fonction de pondération monotone)** *Une fonction* $f : \mathbb{R}^d \mapsto \mathbb{R}$ *est monotone pour la relation d'ordre* $\preceq$ *si pour tout* $(x, x') \in \mathbb{R}^2$

$$x \preceq x' \qquad \Rightarrow \qquad f(\ldots, x, \ldots) \preceq f(\ldots, x', \ldots)$$

**Définition 7.11 (Fonction de pondération supérieure)** *Une fonction* $f : \mathbb{R}^d \mapsto \mathbb{R}$ *est supérieure si le résultat de la fonction est plus grand que chacun de ses arguments :*

$$x_i \preceq f(x_1 \ldots, x_i, \ldots x_m) \qquad (1 \leqslant i \leqslant m)$$

L'algorithme de Knuth (Algorithme 27) construit l'hypergraphe en expansant en priorité les noeuds les plus prometteurs, c'est-à-dire les noeuds de poids minimal. L'ordre de priorité est géré par une file de priorité (file de Fibonacci). Il faut remarquer que contrairement aux arcs d'un graphe, les hyperarcs ne peuvent être expansés si l'ensemble des noeuds de leur queue n'ont pas été engendrés. Pour cette raison l'algorithme maintient une table d'arité des hyperarcs (noté $rest(e)$) qui enregistre combien de noeuds de la queue ont déjà été engendrés.

L'algorithme de Knuth est une contrepartie de l'algorithme de Dijkstra pour le cas des hypergraphes. Il a été utilisé et étudié sous des noms divers et notamment *best first parsing* principalement pour mettre au point des heuristiques de recherche (exactes ou non) de la famille $A\star$ pour des modèles probabilistes génératifs.

**TODO Dire qq part qu'il faut contraindre légèrement: pas de root en temporaire, pas deux fils temporaires etc.**

## 7.3 Analyse par transitions

Les méthodes d'analyse par transition en constituants sont très anciennes et ont pour origine la théorie de la compilation avec les algorithmes à décalage réduction (algorithme d'analyse LR et GLR) ainsi que des formalisations d'algorithmes de planification (comme STRIPS) et de démonstration automatique.

L'ensemble de ces algorithmes travaillent avec une structure de donnée appelée configuration (ou état d'analyse) et qui est essentiellement un couple $\langle \mathbf{S}, \mathbf{B} \rangle$ fait d'une pile et d'une file (ou buffer).

Contrairement aux algorithmes présentés précédemment qui autorisent la programmation dynamique, ici l'ensemble des configurations possibles est en général très vaste de telle sorte que mener une procédure exacte de

recherche de solutions d'analyse est infaisable en pratique. Par conséquent les algorithmes présentés dans cette section sont en général associés à des méthodes de recherche inexactes de solution. Le caractère inexact de la méthode de recherche de solution est compensé par la richesse de l'information stockée dans les configurations, ce qui permet de prendre des décisions très bien informées localement.

On donne en figure 7.8 un système de transitions pour l'analyse en constituants qui suppose des arbres binarisés suivant les méthodes décrites dans les sections précédentes. L'algorithme commence avec un buffer rempli pas la séquence de mots à analyser et termine lorsque la pile ne contient plus qu'un seul élément et que la pile est vide. L'action de décalage insère en sommet de pile le tag du premier mot de la file (tagging) alors que les actions de réduction remplacent les deux non terminaux au sommet de la pile par un nouveu non terminal $X$. Notons qu'il y a autant d'actions de décalage qu'il y a de tags et autant d'actions de reduction qu'il y a de non terminaux ($x \in \Sigma$).

$$
\begin{array}{rcccl}
\textbf{init} & \langle \epsilon, w_0 \dots w_n \rangle & & & \\
\textbf{but} & \langle X, \epsilon \rangle & & & X \in N \\
\textbf{shift}(X) & \langle \mathbf{S}, w_i | \mathbf{B} \rangle & \Rightarrow & \langle \mathbf{S} | X, \mathbf{B} \rangle & X \in Tags \\
\textbf{reduce}(X) & \langle \mathbf{S} | X_i | X_j, \mathbf{B} \rangle & \Rightarrow & \langle \mathbf{S} | X, \mathbf{B} \rangle & X \in N
\end{array}
$$

Figure 7.8: Un système de transitions pour l'analyse en constituants

L'algorithme est fondamentalement non déterministe. Il est par conséquent habituel de l'augmenter (1) d'une méthode de pondération des hypothèses et (2) d'une méthode de recherche de solutions approximative.

**Dérivations et pondérations** Un pas de dérivation est le passage d'une configuration $c_i$ à une configuration $c_{i+1}$ en utilisant une transition (ou action) parmi les $|X|$ actions de décalage et les $|X|$ actions de réduction. On peut éventuellement ajouter une action **stop** qui indique la fin de l'analyse. Une dérivation est une séquence $\mathbf{d} = (c_0, t_0)(c_1, t_1) \dots (c_m, t_m)$ de couples associant des configurations à des actions à exécuter. Lorsque la grammaire est en CNF une dérivation pour le système de transitions décrit en figure 7.8 a une longueur $m = 2n - 1$. Le score global d'une dérivation se calcule en décomposant le score global par le produit des scores locaux :

$$\Psi(\mathbf{d}) = \sum_{i=0}^{m} \psi(c_i, t_i, \mathbf{x}) \tag{7.7}$$

où $\mathbf{x}$ représente la séquence de mots à analyser. Le problème d'analyse syntaxique consiste à résoudre :

$$\hat{\mathbf{d}} = \underset{\mathbf{d} \in \mathbf{D}(\mathbf{x})}{\operatorname{argmax}} \Psi(\mathbf{d}) \tag{7.8}$$

Les dérivations complètes du système de transitions ont comme propriété que leur longueur est de $2n-1$ pas de dérivation pour une phrase de $w_0 \dots w_n$ mots : pour obtenir une analyse complète, il faut réaliser $n$ décalages et $n-1$ opérations de réduction. Lorsqu'on ajoute une action **stop** la longueur de la dérivation vaut $2n$.

Cette propriété facilite l'usage du système de transitions en combinaison avec une méthode de recherche de solutions en faisceau. En effet, on observe empriquement que le score d'une dérivation tel que donné par un modèle discriminant est à peu près linéairement corrélé à sa longueur : plus une dérivation est longue, plus son score est élevé. Ici, comme toutes les dérivations concurrentes ont la même longueur, ce problème de biais lié à la longueur ne se manifeste pas (contrairement à la plupart des systèmes de transitions présentés par la suite).

**Algorithm 29** Algorithme d'analyse en constituants en faisceau

**function** ConstituantsBeamParse($\mathbf{x}$,K)
  $\mathcal{B} \leftarrow \langle \epsilon, w_0 \dots w_n, \varnothing \rangle$
  **for** $0 \leqslant i < 2n-1$ **do**
    **for** $c \in$ K-argmax$_{c \in \mathcal{B}}\, \delta(c)$ **do**
      **for** $t \in T$ **do** ▷ $T$ est l' ensemble des actions
        $c' \leftarrow t(c)$
        $\delta(c') \leftarrow \delta(c) + \psi(c, t, \mathbf{x})$
        $\mathcal{B}' \leftarrow \mathcal{B} \cup c'$
      **end for**
    **end for**
    $\mathcal{B} \leftarrow \mathcal{B}'$
  **end for**
  **return** argmax$_{c \in \mathcal{B}}\, \delta(c)$
**end function**

**Estimation des paramètres** Les données d'entrainement pour l'analyse en constituants se présentent naturellement sous la forme d'un treebank. La première étape de traitement des données consiste à mettre le treebank en forme normale de Chomsky. La seconde étape de traitement consiste

à transformer les arbres en dérivations de référence à l'aide d'une fonction oracle de telle sorte que le jeu de données ait la forme $(\mathbf{x}_i, \mathbf{d}_i)_{i=1}^N$ où $\mathbf{x}_i$ est une séquence de mots et $\mathbf{d}_i$ une dérivation pour l'arbre de référence.

La fonction oracle simule une analyse en ayant pour information supplémentaire $V$ l'ensemble des noeuds de l'hypergraphe qui correspond à l'arbre d'analyse de référence. À chaque étape de dérivation, l'action suivie par l'oracle est donnée par l'algorithme 30.

**Algorithm 30** Oracle statique pour l'analyse en constituants par transitions

```
function StaticOracle(S,B,V)
    if S|(i,k,X1)|(k,j,X2) then      ▷ Il y a au moins un élément dans S
        if (i,j,X0) ∈ V then
            return reduce(X0)
        end if
    end if
    if w_i|B ≠ ϵ then                ▷ Il y a au moins un élément w_i dans B
        if (i,i+1,T) ∈ V then
            return shift(T)
        end if
    end if
end function
```

L'estimation des paramètres à partir des données se fait en comparant les dérivations de référence produites par la fonction oracle aux prédictions de l'analyseur. L'estimation se fait en général avec un modèle à large marge ou un modèle local car un modèle CRF exige de calculer l'ensemble de dérivations pour obtenir le facteur de normalisation. Ce calcul est en général infaisable exhaustivement avec un système de transitions. On donne ici un exemple de procédure d'estimation avec l'algorithme du perceptron en algorithme 31.

**Problème d'approximation par recherche en faisceau** L' algorithme 34 est en général utilisé avec un algorithme de recherche en faisceau, ce qui a pour conséquence que la meilleure analyse prédite n'est pas nécessairement celle qui a le meilleur poids pour le modèle (celle-ci peut avoir été sortie prématurément du faisceau).

Dans ce cas de figure, il est donc possible que l'algorithme d'estimation réalise une mise à jour des poids alors que sans l'approximation introduite par le faisceau le modèle aurait prédit la bonne analyse. Ce comportement peut perturber la descente de gradient jusqu'à causer une divergence.

**Algorithm 31** Perceptron pour un système d'analyse en constituants par transitions

**function** CONSTITUANTS-PERCEPTRON($(\mathbf{x}_i, \mathbf{d}_i)_{i=1}^N$,$E$)
  $\mathbf{w} \leftarrow \bar{\mathbf{0}}$
  **for** $1 \leqslant e \leqslant E$ **do**
    **for** $1 \leqslant i \leqslant N$ **do**
      $\hat{\mathbf{d}} \leftarrow \text{argmax}_{\mathbf{d} \in \mathbf{D}(\mathbf{x})} \sum_{j=0}^{m} \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, c_j, t_j)$ ▷ Parsing
      **if** $\hat{\mathbf{d}} \neq \mathbf{d}_i$ **then**
        $\mathbf{w} \leftarrow \mathbf{w} + \left[\sum_{j=0}^{m} \mathbf{\Phi}(\mathbf{x}_i, c_j, t_j) - \sum_{j=0}^{m} \mathbf{\Phi}(\mathbf{x}_i, \hat{c}_j, \hat{t}_j)\right]$
      **end if**
    **end for**
  **end for**
  **return w**
**end function**

Pour garantir que la descente de gradient converge il faut garantir que la mise à jour a lieu lorsqu'il y a bien une violation effective de la marge et que celle-ci ne provient pas de l'approximation en faisceau.

Pour cette raison on s'autorise dans la pratique à réaliser la mise à jour sur des sous-séquences de dérivation pour lesquelles on a la garantie que la marge est effectivement violée indépendamment des approximations introduites par le faisceau. Autrement dit une mise à jour valide respecte la condition suivante[2] :

$$\Psi(\tilde{\mathbf{y}}^*_{0\ldots k}) > \Psi(\mathbf{y}_{0\ldots k}) \tag{7.9}$$

La méthode la plus utilisée en pratique est la mise à jour rapide (**early update**) où la mise à jour est réalisée sur des sous-dérivations dont le préfixe $0 \ldots k$ correspond à l'étape de dérivation où la dérivation de référence sort du faisceau. Une méthode alternative est la mise à jour à violation maximale de la marge (**max violation update**). Dans ce cas on choisit $k$ tel que

$$k = \underset{0 \leqslant k < 2n-1}{\text{argmax}} \; \Psi(\tilde{\mathbf{y}}^*_{0\ldots k}) - \Psi(\mathbf{y}_{0\ldots k})$$

## 7.4 Analyse lexicalisée

L'analyse en constituants est parfois utilisée dans un cadre d'analyse dite lexicalisée. Dans ce contexte les arbres sont augmentés d'informations qui

[2] La mise à jour invalide qui consiste à comparer la meilleure analyse complète dans le faisceau en fin d'analyse avec la référence ne vérifie pas cette condition dans tous les cas.

permettent d'identifier les têtes syntaxiques. On parle d'annotations en têtes et d'arbres annotés par les têtes.

La lexicalisation est utilisée pour donner des indices destinés à aider à la désambiguisation lors de l'analyse. On illustre l'idée en Figure 7.9. Celle-ci illustre un problème dit d'attachement de groupe prépositionnel. La structure illustrée en (a) suggère que le dépendant *tomates* est à mettre en relation avec *salade* (les tomates sont des ingrédients de la salade) alors que la structure (b) suggère que le dépendant *couvert* est à mettre en relation avec le verbe *manger* (les couverts sont des instruments pour manger).

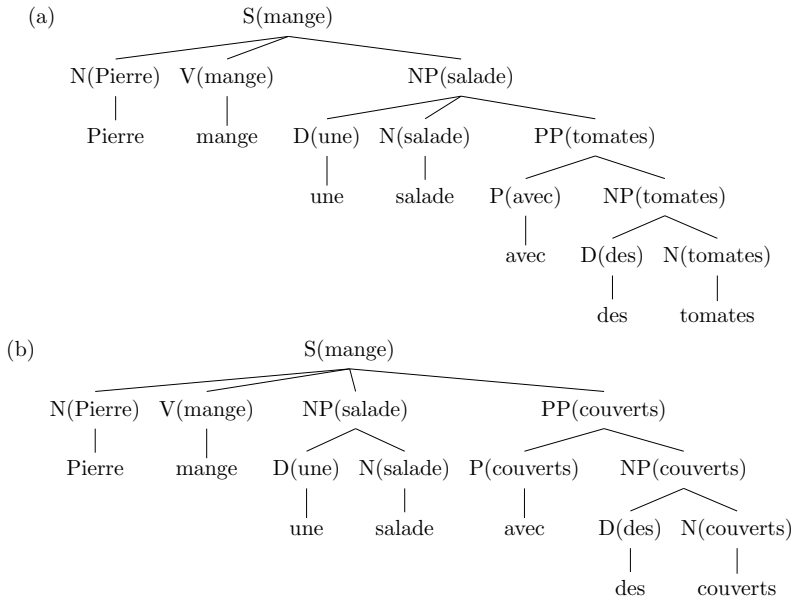



Figure 7.9: Arbres lexicalisés

D'un point de vue formel, les deux phrases peuvent (a) et (b) recevoir l'une ou l'autre analyse. La décision de structure se prend sur base de la relation entre le groupe prépositionnel et son gouverneur. Comme la représentation est lexicalisée on peut voir que les relations de dépendances critiques (salade, tomates) et (manger,couverts) sont explicitées, ce qui donne à l'algorithme d'analyse la possibilité de réaliser le choix approprié.

La plupart des treebanks existants ne donnent pas une telle annotation de têtes. Celle-ci est souvent ajoutée par une heuristique qui se formalise par une table de propagation des têtes. Et l'analyse lexicalisée est formalisée comme une variante des algorithmes qui précèdent en utilisant une

représentation binarisée de la grammaire qui est une variante de la forme normale de Chomsky, parfois appelée 2-LCFG et dont les règles sont de la forme suivante :

$$\begin{array}{lcll} A[h] & \rightarrow & B[h] & C[w] \\ A[h] & \rightarrow & B[w] & C[h] \\ A[h] & \rightarrow & h & \end{array}$$

Les algorithmes présentés précédemment se généralisent au cas lexicalisé (programmation dynamique et transitions). En substance les items d'analyse comportent un indice supplémentaire $\langle i, j, h, X \rangle$ qui indique la position de la tête dans cet item. De plus, commme il y a deux types de règles binaires : l'une assignant comme tête du nouveau syntagme la tête de son fils gauche, l'autre la tête de son fils droit. Par contre la conception naïve d'algorithmes pour ces systèmes à un indice supplémentaire mène à des algorithmes dont la complexité est en $\mathcal{O}(n^5)$.

Comme on peut le remarquer les arbres d'analyse en constituants lexicalisés encodent des arbres de dépendances projectifs. On propose donc d'illustrer ces différents aspects dans le chapitre consacré à l'analyse en dépendances.

# Chapter 8

# Analyse syntaxique en dépendances

## 8.1 Arbres de dépendances

Un graphe de dépendances est un graphe $G = \langle V, E \rangle$ où :

- $W$ est un ensemble de noeuds muni d'une relation d'ordre $\prec$
- $E \subseteq V \times V$ est un ensemble d'arcs

La relation d'ordre sur les noeuds est la relation d'ordre sur les entiers qui indicent les noeuds. $E^*$ dénote la fermeture réflexive transitive de $E$. Cette relation permet de capturer les relations de dominance entre noeuds qui sont indirectes.

Un arc $(i, j) \in E$ représente une relation de dépendance non typée entre deux noeuds. Celle-ci est également notée $i \rightarrow j$. On note $i \xrightarrow{*} j$ un élément de $E^*$

En pratique, les noeuds $w_i \in W$ sont étiquetés par des mots et les arcs $(i, j) \in E$ sont étiquetés par des types qui représentent les fonctions syntaxiques. Ces aspects pratiques sont secondaires pour les problèmes algorithmiques et seront largement ignorés dans la suite de ce chapitre.

**Propriétés** Les arbres de dépendances sont des graphes $G = \langle W, E \rangle$ qui satisfont les contraintes suivantes :

1. $G$ est connexe.

$$i \in W \quad \Rightarrow \quad \exists_j (i \rightarrow j) \in E \vee (j \rightarrow i) \in E$$

2. $G$ est acyclique.

$$i \rightarrow j \in E \quad \Rightarrow \quad j \xrightarrow{*} i \notin E^*$$

3. Chaque noeud de $G$ a au plus un arc entrant :

$$i \rightarrow j \in E \quad \Rightarrow \quad \neg\exists k (k \rightarrow j) \in E$$

4. $G$ est projectif :

$$i \rightarrow j \in E \quad \Rightarrow \quad \forall_{k:i<k<j} (i \xrightarrow{*} k)$$

La condition (4) est parfois relâchée pour traiter les langues à ordre des mots libre.

## 8.2 Analyse par transitions

Les méthodes d'analyse en dépendances par transitions sont dérivées des méthodes d'analyse par décalage réduction utilisées pour l'analyse en constituants et des formalisations d'algorithmes de planification et de démonstration automatique.

L'ensemble de ces algorithmes travaille avec une structure de donnée appelée **configuration** (ou état) et qui est essentiellement un couple $\langle \mathbf{S}, \mathbf{B} \rangle$ fait d'une pile et d'une file (ou buffer). L'ensemble des configurations possibles est en général très vaste de telle sorte que mener une procédure exacte de recherche de solutions d'analyse est en général infaisable. Par conséquent les algorithmes présentés dans cette section sont en général associés à des méthodes de recherche inexactes de solution. Le caractère inexact de la méthode de recherche de solution est compensé par la richesse de l'information stockée dans les configurations, ce qui permet de prendre des décisions très bien informées localement.

### 8.2.1 Système Arc-standard

Le système arc-standard est un modèle d'analyse dont les configurations sont des triplets $\langle \mathbf{S}, \mathbf{B}, A \rangle$ où $\mathbf{S}$ est une pile, $\mathbf{B}$ une file et $A$ un ensemble d'arcs. La donnée à analyser $\mathbf{x} = w_0 \dots w_n$ est une séquence de mots dont le mot $w_0$ est par convention un mot artificiel qui représente la racine de l'arbre.

L'algorithme (Figure 8.1) commence avec un état initial où la pile est vide et la file remplie de la séquence de mots. Il termine lorsque la pile ne

contient plus que la racine de l'arbre et que le buffer est vide. Les actions arc gauche (**left-arc**) et (**right-arc**) sont des contreparties de l'action de réduction d'un analyseur à décalage réduction. L'action de décalage est la contrepartie fidèle d'une action de décalage traditionnelle.

$$
\begin{array}{rrcl}
\textbf{init} & \langle \epsilon, w_0 \dots w_n, \varnothing \rangle & & \\
\textbf{but} & \langle w_0, \epsilon, A \rangle & & \\
\textbf{shift} & \langle \mathbf{S}, w_i | \mathbf{B}, A \rangle & \Rightarrow & \langle \mathbf{S} | w_i, \mathbf{B}, A \rangle \\
\textbf{left} - \textbf{arc} & \langle \mathbf{S} | w_i | w_j, \mathbf{B}, A \rangle & \Rightarrow & \langle \mathbf{S} | w_j, \mathbf{B}, A \cup \{j \rightarrow i\} \rangle \quad (i \neq w_0) \\
\textbf{right} - \textbf{arc} & \langle \mathbf{S} | w_i | w_j, \mathbf{B}, A \rangle & \Rightarrow & \langle \mathbf{S} | w_i, \mathbf{B}, A \cup \{i \rightarrow j\} \rangle
\end{array}
$$

Figure 8.1: Le système de transitions arc standard

Les actions arc-gauche et arc-droit sont augmentées d'une procédure qui collecte les arcs créés en cours d'analyse (représentés par l'ensemble $A$) ce qui permet d'extraire trivialement l'arbre de dépendance en fin d'analyse.

L'algorithme est fondamentalement non déterministe. Il est par conséquent habituel de l'augmenter (1) d'une méthode de pondération des hypothèses et (2) d'une méthode de recherche de solutions approximative (en général non exhaustive en faisceau).

**Dérivations et pondérations** Un pas de dérivation est le passage d'une configuration $c_i$ à une configuration $c_{i+1}$ en utilisant une transition (ou action) $t \in \{shift, left - arc, right - arc, stop\}$. L'action **stop** est exécutée uniquement pour terminer l'analyse lorsque le buffer est vide et que la pile ne peut plus être réduite.

Une séquence de dérivation, ou **dérivation**, est une séquence de la forme $\mathbf{d} = (c_0, t_0) \dots (c_m, t_m)$. On pondère une dérivation en faisant l'hypothèse que son score se décompose par la somme :

$$\Psi(\mathbf{d}) = \sum_{i=0}^{m} \psi(c_i, t_i, \mathbf{x}) \tag{8.1}$$

où $\mathbf{x}$ représente la séquence de mots à analyser. Le problème d'analyse syntaxique consiste en général à résoudre :

$$\hat{\mathbf{d}} = \underset{\mathbf{d} \in \mathbf{D}(\mathbf{x})}{\operatorname{argmax}} \Psi(\mathbf{d}) \tag{8.2}$$

où $\mathbf{D}(\mathbf{x})$ représente l'ensemble des dérivations possibles pour la phrase à analyser.

| Arcs | Pile | File | Action |
|---|---|---|---|
| **root** j' ai réservé un vol pour Sophie | **root** | j' , ai , réservé , un , vol , pour , Sophie | shift |
| **root** j' ai réservé un vol pour Sophie | **root** , j' | ai , réservé , un , vol , pour , Sophie | shift |
| **root** j' ai réservé un vol pour Sophie | **root** , j',ai | réservé , un , vol , pour , Sophie | shift |
| **root** j' ai réservé un vol pour Sophie | **root** , j', ai, réservé | un , vol , pour , Sophie | left-arc |
| **root** j' ai réservé un vol pour Sophie | **root** , j', réservé | un , vol , pour , Sophie | left-arc |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé | un , vol , pour , Sophie | shift |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé , un | vol , pour , Sophie | shift |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé , un, vol | pour , Sophie | left-arc |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé , vol | pour , Sophie | right-arc |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé | pour , Sophie | shift |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé , pour | Sophie | shift |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé , pour , Sophie | $\epsilon$ | right-arc |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé , pour | $\epsilon$ | right-arc |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé | $\epsilon$ | right-arc |
| **root** j' ai réservé un vol pour Sophie | **root** | $\epsilon$ | goal! |

Figure 8.2: Exemple de dérivation arc-standard

Les dérivations complètes du système de transition arc-standard ont comme propriété que leur longueur est de $2n-1$ pas de dérivation pour une phrase de $w_0 \dots w_n$ mots : pour obtenir une analyse complète, il faut réaliser $n$ décalages et $n-1$ opérations d'arcs (gauche ou droite). Lorsqu'on ajoute une action **stop** la longueur de la dérivation vaut $2n$.

Cette propriété facilite l'usage du système arc-standard en combinaison avec une méthode de recherche de solutions en faisceau. En effet, on observe empriquement que le score d'une dérivation tel que donné par un modèle discriminant est à peu près linéairement corrélé à sa longueur : plus une dérivation est longue, plus son score est élevé. Ici, comme toutes les dérivations concurrentes ont la même longueur, ce problème de biais lié à la longueur ne se manifeste pas (contrairement à la plupart des systèmes de transitions présentés par la suite).

**Algorithm 32** Algorithme d'analyse Arc Standard en faisceau

```
function ArcStandardBeamParse(x,K)
    B ← ⟨ϵ, w_0 ... w_n, ∅⟩
    for 0 ⩽ i < 2n − 1 do
        for c ∈ K-argmax_{c∈B} δ(c) do
            for t ∈ T do                    ▷ T est l' ensemble des actions
                c' ← t(c)
                δ(c') ← δ(c) + ψ(c, t, x)
                B' ← B ∪ c'
            end for
        end for
        B ← B'
    end for
    return argmax_{c∈B} δ(c)
end function
```

**Estimation des paramètres** L'estimation des paramètres se réalise à partir d'un corpus annoté $C = (\mathbf{x}_i, \mathbf{y}_i)_{i=1}^{N}$ qui est un exemplaire de $N$ phrases où chaque séquence de mots $\mathbf{x}_i$ est associée à un arbre de référence $\mathbf{y}_i$.

La procédure d'apprentissage consiste à estimer un vecteur de paramètres $\mathbf{w}$ de telle sorte que les dérivations prédites par l'algorithme sont les plus similaires aux dérivations de référence. Comme les données sont naturellement des couples $(\mathbf{x}_i, \mathbf{y}_i)$, il faut transformer les arbres de référence $\mathbf{y}_i$ en dérivations. On réalise cette opération à l'aide d'un **oracle** statique (Algorithme 33) qui permet de simuler l'analyse de référence en utilisant

comme donnée $A$ l'ensemble des arcs de l'arbre de référence. L'oracle permet également de mettre en évidence l'idée bottom-up du système arc-standard : la création d'un arc vers un noeud $i$ exige que tous les dépendants de $i$ soient déjà identifiés.

**Algorithm 33** Oracle statique pour le système arc standard

**function** ARCSTANDARDSTATICORACLE($\mathbf{S}, \mathbf{B}, A, A_{ref}$)
  **if** $\mathbf{S}|i|j$ **then** ▷ Il y a au moins 2 éléments sur la pile
    **if** $i \rightarrow j \in A_{ref}$ **and** $\forall k(j \rightarrow k \in A_{ref} \Rightarrow j \rightarrow k \in A)$ **then**
      **return right − arc**
    **else if** $j \rightarrow i \in A_{ref}$ **and** $i \neq 0$ **and** $\forall k(i \rightarrow k \in A_{ref} \Rightarrow i \rightarrow k \in A)$ **then**
      **return left − arc**
    **end if**
  **end if**
  **if** $\mathbf{B} \neq \epsilon$ **then** ▷ Il y a au moins un élément dans la file
    **return shift**
  **end if**
**end function**

Le jeu de données transformé où les arbres $\mathbf{y}_i$ sont transformés en dérivation $\mathbf{d}_i$ prend alors la forme suivante. $C = (\mathbf{x}_i, \mathbf{d}_i)_{i=1}^{N}$. L'estimation des paramètres se fait en général avec un modèle à large marge car l'utilisation d'un CRF exige un facteur de normalisation qui demande de calculer l'ensemble des dérivations possibles pour une phrase donnée, ce qui est trop coûteux en temps de calcul. On donne ici une méthode d'estimation à l'aide de l'algorithme du perceptron (Algorithme 34).

**Algorithm 34** Perceptron pour un système d'analyse en dépendances par transitions

**function** ARC-STANDARD-PERCEPTRON($(\mathbf{x}_i, \mathbf{d}_i)_{i=1}^{N}$,$E$)
  $\mathbf{w} \leftarrow \bar{\mathbf{0}}$
  **for** $1 \leqslant e \leqslant E$ **do**
    **for** $1 \leqslant i \leqslant N$ **do**
      $\hat{\mathbf{d}} \leftarrow \text{argmax}_{\mathbf{d} \in \mathbf{D}(\mathbf{x})} \sum_{j=0}^{m} \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, c_j, t_j)$ ▷ Parsing
      **if** $\hat{\mathbf{d}} \neq \mathbf{d}_i$ **then**
        $\mathbf{w} \leftarrow \mathbf{w} + \left[\sum_{j=0}^{m} \mathbf{\Phi}(\mathbf{x}_i, c_j, t_j) - \sum_{j=0}^{m} \mathbf{\Phi}(\mathbf{x}_i, \hat{c}_j, \hat{t}_j)\right]$
      **end if**
    **end for**
  **end for**
  **return w**
**end function**

**Unicité de l'oracle** L'analyse par transitions suppose que chaque arbre de dépendance de référence $\mathbf{y}_i$ peut être transformé en une unique dérivation de référence $\mathbf{d}_i$. Autrement dit, on suppose implicitement une fonction de la forme $A \mapsto D$.

On peut toutefois remarquer que cette fonction n'est pas déterministe pour le système de transitions arc-standard : plusieurs dérivations peuvent représenter le même arbre de dépendances.


| | | | | | | |
|---|---|---|---|---|---|---|
| Dérivation A | S | S | LA | S | RA | RA |
| Dérivation B | S | S | S | RA | LA | RA |

Figure 8.3: Dérivations multiples pour un arbre de dépendances avec le système arc-standard

Cette observation illustre un problème du système arc-standard. On apprend un modèle qui apprend à prédire un sous-ensemble des dérivations qui correspondent aux arbres de référence.

**Problème d'approximation par recherche en faisceau** L' algorithme 34 est en général utilisé avec un algorithme de recherche en faisceau, ce qui a pour conséquence que la meilleure analyse prédite n'est pas nécessairement celle qui a le meilleur poids pour le modèle (celle-ci peut avoir été sortie prématurément du faisceau).

Dans ce cas de figure, il est donc possible que l'algorithme d'estimation réalise une mise à jour des poids alors que sans l'approximation introduite par le faisceau le modèle aurait prédit la bonne analyse. Ce comportement peut perturber la descente de gradient jusqu'à causer une divergence.

Pour garantir que la descente de gradient converge il faut garantir que la mise à jour a lieu lorsqu'il y a bien une violation effective de la marge et que celle-ci ne provient pas de l'approximation en faisceau.

Pour cette raison on s'autorise dans la pratique à réaliser la mise à jour sur des sous-séquences de dérivation pour lesquelles on a la garantie que la marge est effectivement violée indépendamment des approximations introduites par le faisceau. Autrement dit une mise à jour valide respecte la

condition suivante[1] :

$$\Psi(\tilde{\mathbf{y}}^*_{0\ldots k}) > \Psi(\mathbf{y}_{0\ldots k}) \tag{8.3}$$

La méthode la plus utilisée en pratique est la mise à jour rapide (**early update**) où la mise à jour est réalisée sur des sous-dérivations dont le préfixe $0 \ldots k$ correspond à l'étape de dérivation où la dérivation de référence sort du faisceau. Une méthode alternative est la mise à jour à violation maximale de la marge (**max violation update**). Dans ce cas on choisit $k$ tel que

$$k = \underset{0 \leqslant k < 2n-1}{\text{argmax}} \; \Psi(\tilde{\mathbf{y}}^*_{0\ldots k}) - \Psi(\mathbf{y}_{0\ldots k})$$

### 8.2.2 Système Arc-eager

Le système arc-eager est un modèle d'analyse dont les configurations sont des triplets $\langle \mathbf{S}, \mathbf{B}, A \rangle$ où $\mathbf{S}$ est une pile, $\mathbf{B}$ une file et $A$ un ensemble d'arcs. La donnée à analyser $\mathbf{x} = w_0 \ldots w_n$ est une séquence de mots dont le mot $w_0$ est par convention un mot artificiel qui représente la racine de l'arbre.

L'algorithme (Figure 8.4) commence avec un état initial où la pile est vide et la file remplie de la séquence de mots. Il termine lorsque le buffer est vide. Contrairement au système arc-standard, ici le système essaye d'attacher les nouveaux mots le plus rapidement possible dans la structure : les transitions left-arc et right-arc attachent le premier mot du buffer à la structure avant même qu'il ait été placé sur la pile. La transition de réduction qui supprime la tête de pile indique que le mot ne fera plus partie d'une relation de dépendance par la suite. Quant à la transition de décalage, elle est plus classique.

| | | | | |
|---|---|---|---|---|
| **init** | $\langle w_0, w_1 \ldots w_n, \varnothing \rangle$ | | | |
| **but** | $\langle \mathbf{S}, \epsilon, A \rangle$ | | | |
| **shift** | $\langle \mathbf{S}, w_i \vert \mathbf{B}, A \rangle$ | $\Rightarrow$ | $\langle \mathbf{S} \vert w_i, \mathbf{B}, A \rangle$ | |
| **left − arc** | $\langle \mathbf{S} \vert w_i, w_j \vert \mathbf{B}, A \rangle$ | $\Rightarrow$ | $\langle \mathbf{S}, w_j \vert \mathbf{B}, A \cup \{j \to i\} \rangle$ | $(i \neq w_0$ et $\neg \exists k (k \to i \in A))$ |
| **right − arc** | $\langle \mathbf{S} \vert w_i, w_j \vert \mathbf{B}, A \rangle$ | $\Rightarrow$ | $\langle \mathbf{S} \vert w_i \vert w_j, \mathbf{B}, A \cup \{i \to j\} \rangle$ | $\neg \exists k (k \to j \in A)$ |
| **reduce** | $\langle \mathbf{S} \vert w_i, \mathbf{B}, A \rangle$ | $\Rightarrow$ | $\langle \mathbf{S}, \mathbf{B}, A \rangle$ | $\exists k (k \to i \in A)$ |

Figure 8.4: Le système de transitions arc eager

Le système arc-eager offre des garanties plus faibles que arc-standard : le but est satisfait dès que le buffer est vide. En particulier il ne garantit

[1] La mise à jour invalide qui consiste à comparer la meilleure analyse complète dans le faisceau en fin d'analyse avec la référence ne vérifie pas cette condition dans tous les cas.

pas que l'analyse renvoie un arbre de dépendances unique. Il renvoie en général une forêt : plusieurs racines sont possibles. Il est par conséquent classique d'ajouter une étape de finalisation dans les implémentations qui relie l'ensemble des racines résultantes à $w_0$.

| Arcs | Pile | File | Action |
|---|---|---|---|
| **root** j' ai réservé un vol pour Sophie | **root** | j' , ai , réservé , un , vol , pour , Sophie | shift |
| **root** j' ai réservé un vol pour Sophie | **root** , j' | ai , réservé , un , vol , pour , Sophie | shift |
| **root** j' ai réservé un vol pour Sophie | **root** , j' , ai | réservé , un , vol , pour , Sophie | left-arc |
| **root** j' ai réservé un vol pour Sophie | **root** , j' | réservé , un , vol , pour , Sophie | left-arc |
| **root** j' ai réservé un vol pour Sophie | **root** | réservé , un , vol , pour , Sophie | shift |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé | un , vol , pour , Sophie | shift |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé , un , | vol , pour , Sophie | left-arc |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé , | vol , pour , Sophie | right-arc |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé , vol | pour , Sophie | reduce |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé | pour , Sophie | right-arc |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé , pour | Sophie | right-arc |
| **root** j' ai réservé un vol pour Sophie | **root** , réservé , pour , Sophie | $\epsilon$ | terminate (or redu |

**Dérivations et pondérations** Un pas de dérivation est le passage d'une configuration $c_i$ à une configuration $c_{i+1}$ en utilisant une transition (ou action) $t \in \{shift, left-arc, right-arc, reduce, stop\}$. L'action **stop** est exécutée uniquement pour terminer l'analyse lorsque le buffer est vide.

Une séquence de dérivation, ou **dérivation**, est une séquence de la forme $\mathbf{d} = (c_0, t_0) \dots (c_m, t_m)$. On pondère une dérivation en faisant l'hypothèse que son score se décompose par la somme :

$$\Psi(\mathbf{d}) = \sum_{i=0}^{m} \psi(c_i, t_i, \mathbf{x}) \tag{8.4}$$

où $\mathbf{x}$ représente la séquence de mots à analyser. Le problème d'analyse

syntaxique consiste en général à résoudre :

$$\hat{\mathbf{d}} = \underset{\mathbf{d} \in \mathbf{D}(\mathbf{x})}{\operatorname{argmax}} \Psi(\mathbf{d}) \tag{8.5}$$

où $\mathbf{D}(\mathbf{x})$ représente l'ensemble des dérivations possibles pour la phrase à analyser.

Les dérivations complètes du système de transition arc-eager ont comme propriété que leur longueur vaut **au plus** $2n-1$ pas de dérivation pour une phrase de $w_0 \dots w_n$ mots : il est en effet possible de terminer une analyse en $n$ étapes (après $n$ décalages tous les mots sont racines d'un arbre différent et l'analyse est potentiellement terminée).

Cette propriété complique l'usage du système arc-eager en combinaison avec une méthode de recherche de solutions en faisceau. En effet il faut pouvoir comparer des dérivations de longueurs différentes, ce qui est généralement problématique tant pour la gestion du faisceau que de la gestion de la corrélation entre longueur de la séquence et score de la dérivation. Pour ces raisons, le système arc-eager est habituellement utilisé avec un modèle de recherche de solutions qui est glouton et une approximation par des modèles statistiques locaux. Il faut finalement remarquer que dans le contexte où le modèle est glouton, la procédure d'analyse syntaxique est particulièrement simple (Algorithme 35).

**Algorithm 35** Algorithme d'analyse Arc Eager glouton

```
function ArcEagerGreedyParse(x)
    c ← ⟨w_0, w_1 ... w_n, ∅⟩
    while c_buffer ≠ ∅ do
        t ← argmax_{t∈T} ψ(c, t, x)        ▷ T est l'ensemble des actions
        c ← t(c)
    end while
    return c
end function
```

**Estimation des paramètres par modèle local** L'estimation des paramètres se réalise à partir d'un corpus annoté $C = (\mathbf{x}_i, \mathbf{y}_i)_{i=1}^{N}$ qui est un exemplaire de $N$ phrases où chaque séquence de mots $\mathbf{x}_i$ est associée à un arbre de référence $\mathbf{y}_i$.

Dans le contexte d'un modèle local, la procédure d'apprentissage consiste à estimer un vecteur de paramètres $\mathbf{w}$ de telle sorte que pour chaque configuration d'analyse identifiable dans les données de référence, la prédiction

de l'algorithme soit la plus similaire à l'action de référence trouvée dans les données. Cette procédure suppose de transformer un treebank $(\mathbf{x}_i, \mathbf{y}_i)_{i=1}^{N}$ en une séquence de couples $(x_i, y_i)_{i=1}^{N'}$ de configurations $x_i$ et d'actions $y_i$ à prédire.

On réalise cette opération à l'aide d'un **oracle** statique (Algorithme 36) qui permet de simuler l'analyse de référence en utilisant comme donnée $A$ l'ensemble des arcs de l'arbre de référence.

**Algorithm 36** Oracle statique pour le système arc eager

```
function ArcEagerStaticOracle(S, B, A, A_ref)
    if S|i and j|B then                          ▷ Il y a au moins 1 élément sur la pile et 1 sur la file
        if j → i ∈ A_ref and i ≠ 0 then
            return left − arc
        else if i → j ∈ A_ref then
            return right − arc
        end if
    end if
    if S|i then                                  ▷ Il y a au moins 1 élément sur la pile
        if ∃k(k → i ∈ A) and ∀k'(i → k') ∈ A_ref ⇒ (i → k') ∈ A then
            return reduce
        end if
    end if
    if j|B then                                  ▷ Il y a au moins 1 élément dans le buffer
        return shift
    end if
end function
```

Le jeu de données transformé où les arbres $\mathbf{y}_i$ sont transformés en une séquence de couples $(x_i, t_i)_{i=1}^{N}$ est utilisé pour entraîner tout modèle d'apprentissage approprié : modèle à large marge local, modèle de régression logistique multinomiale ou réseau de neurones. Contrairement aux modèles structurés présentés précédemment, calculer le facteur de normalisation pour les modèles logistiques ne pose pas de problème particulier dans le cas des modèles locaux. On donne ici un exemple d'algorithme d'estimation de paramètres à l'aide de l'algorithme du perceptron (Algorithme 37).

**Unicité de la dérivation et oracles dynamiques** La procédure d'apprentissage local avec oracle statique consiste à transformer un treebank $(\mathbf{x}_i, \mathbf{y}_i)_{i=1}^{N}$ en une séquence de couples $(x_i, y_i)_{i=1}^{N'}$ de configurations $x_i$ et d'actions $y_i$ à prédire.

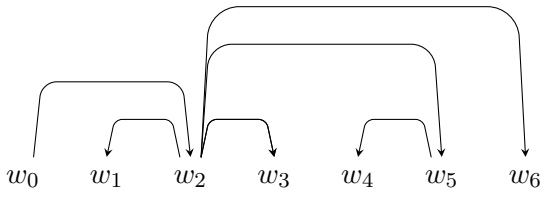
Cette transformation pose deux types de problèmes. D'une part, on suppose (a) qu'il existe une et une seule dérivation qui correspond à un arbre de dépendances, ce qui n'est pas toujours vrai (Figure 8.5). D'autre part (b) le modèle statistique est entraîné uniquement sur des couples configurations et actions $(x_i, y_i)$ de référence : lorsque l'analyseur a commis une

**Algorithm 37** Perceptron pour un système à apprentissage local

```
1: function ARC-EAGER-LOCAL-PERCEPTRON((x_i, t_i)_{i=1}^N, E)
2:     w ← 0̄
3:     for 1 ⩽ e ⩽ E do
4:         for 1 ⩽ i ⩽ N do
5:             t̂ ← argmax_{t∈T} w^T Φ(c_i, t)
6:             if t̂ ≠ t_i then
7:                 w ← w + [Φ(c_j, t_j) − Φ(c_j, t̂)]
8:             end if
9:         end for
10:    end for
11:    return w
12: end function
```

erreur en cours d'analyse, il se trouve potentiellement face à de nouvelles configurations qu'il n'aura jamais vues lors de l'apprentissage du modèle statistique.

L'idée des oracles dynamiques consiste à corriger (a) et (b) en fournissant à la procédure d'apprentissage des couples $(x_i, y_i)$ qui proviennent des multiples dérivations potentielles de l'arbre de référence ou de couples issus de dérivations légèrement erronnées qui simulent l'état de l'analyseur lorsqu'il a commis une ou plusieurs erreurs d'analyse. Le but est de fournir à l'analyseur des données qui lui permettent d'éviter la propagation d'erreurs.




| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Dérivation A | S | LA | RA | RA | S | LA | R | RA | R | RA |
| Dérivation B | S | LA | RA | RA | R | S | LA | RA | R | RA |

Figure 8.5: Dérivations multiples pour un arbre de dépendances

Un **oracle dynamique** est une fonction qui étant donnée une configuration d'analyse et un arbre de référence donne l'ensemble des meilleures actions à réaliser pour minimiser l'erreur d'analyse.

En notant $A_{ref}$ l'ensemble des arcs d'un arbre de dépendances de référence et $A$ l'ensemble des arcs d'une configuration donnée, on définit :

$$\mathcal{L}(A_{ref}, A) = |A_{ref} - A| \tag{8.6}$$

Ce coût représente le nombre d'arcs de référence $A_{ref}$ qui sont absents de l'ensemble $A$ des arcs de la configuration. De plus, on dit qu'un ensemble d'arcs $A$ est atteignable depuis une configuration $c = \langle \mathbf{S}, \mathbf{B}, A' \rangle$, ce que l'on note $c \rightsquigarrow A$, si et seulement si il existe une séquence de transitions qui mène de $c$ à $c' = \langle \mathbf{S}', \mathbf{B}', A \rangle$ Étant donnée une configuration $c$, une action $t$ et un arbre de référence $A_{ref}$, on définit le coût d'exécution d'une action comme suit :

$$\mathcal{C}(c, t, A_{ref}) = \left( \min_{A : t(c) \rightsquigarrow A} \mathcal{L}(A_{ref}, A) \right) - \left( \min_{A : c \rightsquigarrow A} \mathcal{L}(A_{ref}, A) \right) \tag{8.7}$$

Autrement dit, le coût d'une transition représente la différence de coût entre le meilleur arbre atteignable après avoir exécuté l'action $t$ et le meilleur arbre atteignable avant d'avoir exécuté l'action $t$. Dans le cas d'une analyse en dépendances cela correspond

Remarquons qu'il existe au moins une transition $t$ pour laquelle $\mathcal{C}(c, t, A_{ref}) = 0$. La raison est que si $A$ est atteignable depuis $c$ alors il existe nécessairement une transition qui permet d'y arriver. On peut donc définir la fonction indicatrice suivante :

$$o(c, t, A_{ref}) = \begin{cases} 1 & \text{si } \mathcal{C}(c, t, A_{ref}) = 0 \\ 0 & \text{sinon} \end{cases} \tag{8.8}$$

Étant donnée une configuration d'analyse $c$, l'ensemble des actions qui minimisent les erreurs d'analyse est l'ensemble des actions pour lequel (8.8) est vrai, c'est-à-dire l'ensemble $Z = \{t | o(c, t, A_{ref}) = 1\}$. On peut se convaincre que $Z$ est un ensemble de plus d'un élément en considérant la figure 8.5.

L'oracle dynamique est une fonction capable de nous dire quelles actions sont optimales dans une configuration donnée. On peut donc l'utiliser dans un scénario d'apprentissage comme illustré en Algorithme 38 pour le cas du perceptron. La fonction de choix (notée CHOOSE) choisira $\hat{t}$ si $t \in Z$, sinon elle choisira un élément aléatoirement dans $Z$.

On peut également modifier la fonction de choix pour faire explorer lors de l'apprentissage des configurations d'erreur. Dans ce contexte on décidera d'une stratégie d'exploration approriée des configurations d'erreur. Par exemple on pourra choisir une action $\hat{t} \notin Z$ après un certain nombre d'époques et en fixant une certaine probabilité de choisir entre l'action erronée ou sa correction prise dans l'ensemble $Z$.

**Algorithm 38** Apprentissage par oracle dynamique pour un système arc eager pondéré par un perceptron

```
function DYNAMIC-PERCEPTRON-ARC-EAGER((x_i, y_i)_{i=1}^N, E)
    w ← 0
    for 1 ⩽ e ⩽ E do
        for 1 ⩽ i ⩽ N do
            A_ref ← ARCS(y_i)
            w_0 ... w_n ← x_i
            c ← ⟨w_0, w_1 ... w_n, ϵ⟩
            while c_buffer ≠ ∅ do
                t̂ = argmax_{t∈T} w^T Φ(c, t)
                Z ← {t|o(c, t, A_ref) = 1}
                t_o ← argmax_{t∈Z} w^T Φ(c, t)
                if t̂ ∉ Z then
                    w ← w + Φ(c, t_o) − Φ(c, t̂)
                end if
                t ←CHOOSE(e,t̂,Z)
                c ← t(c)
            end while
        end for
    end for
end function
```

**Oracle dynamique pour le système arc-eager** La difficulté pour créer un oracle dynamique réside dans la résolution des problèmes de minimisation en équation (8.7) : déterminer le meilleur arbre atteignable depuis une configuration $c = \langle \mathbf{S}, \mathbf{B}, A \rangle$ donnée n'est pas un problème algorithmiquement trivial pour la plupart des systèmes de transitions.

Le système de transitions arc-eager a la propriété d'être *arc-decomposable*, ce qui permet d'évaluer cette équation très efficacement. Pour comprendre cette propriété on définit au préalable deux notions, l'*arc-reachability* et la *tree-reachability*. On dit d'un ensemble d'arcs $A$ qu'il est atteignable (*tree-reachable*) depuis $c$ si et seulement si une séquence de transitions mène de $c$ à une configuration $c' = \langle \mathbf{S}', \mathbf{B}', A' \rangle$ telle que $A = A'$. On dit d'un arc $i \rightarrow j$ (resp. $j \rightarrow i$) qu'il est atteignable en $c$ si $i \rightarrow j \in A$ ou que $i \in \mathbf{S} \cup \mathbf{B}$ et $j \in \mathbf{B}$. On dit d'un système de transitions qu'il est arc décomposable si le problème de *tree reachability* se réduit au problème d'*arc reachability*.

Un système arc décomposable a donc des propriétés qui permettent très facilement de détecter si un arc de référence est perdu en suivant une transition, ce qui permet d'évaluer (8.7) avec un faible coût en calculs. Contrairement au système arc-eager, le système arc standard ne possède pas cette propriété de décomposabilité, ce qui explique pourquoi c'est le premier qui est utilisé principalement avec des oracles dynamiques.

À partir de là, on peut ainsi dériver un oracle dynamique pour le système

arc-eager (Figure 8.6). Considérons une configuration $c = \langle \mathbf{S}|i, j|\mathbf{B}, A\rangle$ et un ensemble d'arcs de référence $A_{ref}$.

| $c = \langle \mathbf{S}\|i, j\|\mathbf{B}, A\rangle$ | | |
|---|---|---|
| $o(c, LA, A_{ref})$ | = | $\begin{cases} 0 & \text{si } \exists k \in \mathbf{B}\|i \rightarrow k \in A_{ref} \vee k \rightarrow i \in A_{ref} \\ 1 & \text{sinon} \end{cases}$ |
| $o(c, RA, A_{ref})$ | = | $\begin{cases} 0 & \text{si } \exists k \in \mathbf{B} \cup \mathbf{S}\|k \rightarrow j \in A_{ref} \vee \exists k \in \mathbf{S}\|j \rightarrow k \in A_{ref} \\ 1 & \text{sinon} \end{cases}$ |
| $o(c, R, A_{ref})$ | = | $\begin{cases} 0 & \text{si } \exists k \in \mathbf{B}\|i \rightarrow k \in A_{ref} \\ 1 & \text{sinon} \end{cases}$ |
| $o(c, S, A_{ref})$ | = | $\begin{cases} 0 & \text{si } \exists k \in \mathbf{S}\|k \rightarrow j \in A_{ref} \vee \exists k \in \mathbf{S}\|j \rightarrow k \in A_{ref} \\ 1 & \text{sinon} \end{cases}$ |

Figure 8.6: Oracle dynamique pour le système arc-eager

L'action left-arc crée un arc $j \rightarrow i$ et enlève $i$ de la pile. Cela signifie que tout arc $i \rightarrow k (k \in \mathbf{B})$ ou $k \rightarrow i (k \in \mathbf{B})$ devient inaccessible. Le coût est le nombre d'arcs de ce type qui existent dans $A_{ref}$.

L'action right-arc crée un arc $i \rightarrow j$ et empile $j$. Cela signifie que $j$ ne peut plus dominer un dépendant dans $\mathbf{S}$ et que $j$ ne peut plus devenir dépendant d'aucun autre mot dans $\mathbf{S}$ et $\mathbf{B}$ (vu qu'un mot ne peut avoir qu'une tête). Le coût est le nombre d'arcs de ce type qui existent dans $A_{ref}$.

L'action de réduction qui dépile $i$ signifie que plus aucun arc de la forme $i \rightarrow k (k \in \mathbf{B})$ ne peut plus être créé. Le coût est le nombre d'arcs de ce type qui existent dans $A_{ref}$[2].

L'action de décalage signifie que $j$ ne pourra plus être mis en relation de dépendance avec des éléments de la pile. Le coût est le nombre d'arcs de ce type qui existent dans $A_{ref}$.

### 8.2.3 Modèle de Covington

Idée : $\mathbf{S}_1$ = mots déjà traités $\mathbf{S}_2$ = liste qui contient successivement les candidats à attacher à $w_j$

[2]Les arcs de la forme $k \rightarrow i$ ne peuvent plus être créés non plus, mais ils sont déjà pris en compte par les autres règles.

$$
\begin{array}{rlcl}
\textbf{init} & \langle \epsilon, \epsilon, w_0 \dots w_n, \varnothing \rangle & & \\
\textbf{but} & \langle \epsilon, w_0|\mathbf{S}_2, w_n, A \rangle & & (tocheck) \\
\textbf{shift} & \langle \mathbf{S}_1, \mathbf{S}_2, w_i|\mathbf{B}, A \rangle & \Rightarrow & \langle \mathbf{S}_1\mathbf{S}_2|w_i, \epsilon, \mathbf{B}, A \rangle \\
\textbf{no}-\textbf{arc} & \langle \mathbf{S}_1|w_i, \mathbf{S}_2, \mathbf{B}, A \rangle & \Rightarrow & \langle \mathbf{S}_1, w_i|\mathbf{S}_2, \mathbf{B}, A \rangle \\
\textbf{left}-\textbf{arc} & \langle \mathbf{S}_1|w_i, \mathbf{S}_2, w_j|\mathbf{B}, A \rangle & \Rightarrow & \langle \mathbf{S}_1, w_i|\mathbf{S}_2, w_j|\mathbf{B}, A \cup \{j \rightarrow i\} \rangle (\neg\exists k (k \rightarrow i \in A) \wedge i \xrightarrow{*} j \notin A*) \\
\textbf{right}-\textbf{arc} & \langle \mathbf{S}_1|w_i, \mathbf{S}_2, w_j|\mathbf{B}, A \rangle & \Rightarrow & \langle \mathbf{S}_1, w_i|\mathbf{S}_2, w_j|\mathbf{B}, A \cup \{i \rightarrow j\} \rangle (\neg\exists k (k \rightarrow j \in A) \wedge j \xrightarrow{*} i \notin A*)
\end{array}
$$

Figure 8.7: Le système de transitions à pile partagée

## 8.3 Analyse CKY

### 8.3.1 La version naïve

On peut envisager adapter l'algorithme CKY au cas de l'analyse en dépendances de manière directe et naïve. C'est ce que nous présentons dans ce qui suit. Cette première solution pose toutefois un problème de complexité car les analyses sont produites en $\mathcal{O}(n^5)$. On donne en section 8.3.2 une version moins naïve qui produit des analyses projectives en $\mathcal{O}(n^3)$ et qui est la version utilisée dans les implémentations habituelles.

Sous forme déductive, les items de l'algorithme sont des triplets de la forme $\langle i, j, h \rangle$ où $i$ est l'indice de l'extrémité gauche de l'empan, $j$ l'indice de l'extrémité droite de l'empan et $h\,(i \leqslant h \leqslant j)$ est la position de la tête dans la séquence de mots $\mathbf{x} = w_1 \dots w_n$ à analyser. En suivant le principe classique de l'algorithme CKY, l'algorithme donné ici commence en supposant que chaque mot définit son propre empan puis essaye de combiner des empans contigus en empans de plus en plus larges en utilisant deux actions de réductions qui se distinguent par leur manière d'assigner la tête au nouvel empan. On résume cet algorithme sous forme de système déductif en figure 8.8. On peut remarquer que chacune des règles de réduction (gauche et droite) fait intervenir 5 indices libres ce qui suggère une complexité en $\mathcal{O}(n^5)$.

**Estimation des poids** Cet algorithme peut être pondéré par tout modèle discriminant. L'estimation des poids pour un modèle CRF demande de déployer des récurrences dedans dehors (*inside-outside*) et est difficile à rendre efficace à l'usage. Les modèles à large marge sont en principe plus faciles à utiliser mais cet algorithme naïf n'est en général pas utilisé en pratique. C'est l'algorithme par factorisation des arcs présenté ci-dessous qui est utilisé

$$\begin{array}{rll} \textbf{init} & \dfrac{}{\langle i,i+1,w_i\rangle} & (0 \leqslant i < n) \\ \textbf{but} & \langle 0,n,h\rangle & \\ \textbf{reduce}(\leftharpoonup) & \dfrac{\langle i,k,h\rangle \quad \langle k,j,h'\rangle}{\langle i,j,h\rangle} & \\ \textbf{reduce}(\rightharpoonup) & \dfrac{\langle i,k,h\rangle \quad \langle k,j,h'\rangle}{\langle i,j,h'\rangle} & \end{array}$$

Figure 8.8: Algorithme CKY pour l'analyse en dépendances

en pratique.

### 8.3.2 Factorisation en arcs

L'algorithme d'analyse en dépendances par factorisation des arcs (*arc factored parser*, Algorithme 39) peut-être vu comme une reformulation de l'algorithme CKY au cas des dépendances projectives. L'algorithme cherche à construire itérativement des sous-arbres qui couvrent des empans de plus en plus grands et s'arrête lorsqu'il a construit le meilleur sous-arbre pour un empan qui couvre toute la phrase à analyser.

Pour un empan $(i, j)$ l'algorithme construit les **sous-arbres incomplets** ($\perp$) qui couvrent cet empan, qu'ils commencent en $i$ ou en $j$ en créant les arcs correspondants. Cette première étape cherche à construire des arbres dont un arc $i \rightarrow j$ ou $j \rightarrow i$ couvre l'intégralité de l'empan. Dans un second temps, l'algorithme cherche à produire des **sous-arbres complets** ($\top$) en combinant un arbre incomplet avec un arbre complet qui couvre un empan plus petit, ce qui permet de couvrir l'empan avec des relations de dépendances indirectes de la forme $i \xrightarrow{*} j$ ou $j \xrightarrow{*} i$.

**Arbres incomplets** Pour un empan donné $(i, j)$ l'algorithme commence par créer les arcs $i \rightarrow j$ et $j \rightarrow i$. On dit de ces arcs qu'ils représentent des arbres incomplets car ils ne codent pas toute la projection du gouverneur. Par exemple, si $i$ est choisi comme gouverneur et que l'arc $i \rightarrow j$ est ajouté, l'arbre incomplet va capturer toutes les relations $i \xrightarrow{*} k$ pour $i \leqslant k \leqslant j$. Par contre il est possible que des relations $i \xrightarrow{*} l$ existent avec $l > j$.

On peut remarquer en Figure 8.9 que l'ajout d'un arc $i \rightarrow j$ (ou $j \rightarrow i$) n'est autorisé que dans peu de configurations préexistantes. Les différents cas sont illustrés sur la figure et motivent les lignes correspondantes de l'algorithme 39. Seul le cas où à la fois $i$ et $j$ ont des dépendants qui sont situés dans l'intervalle $i \ldots j$ autorise la création de l'arc. Les cas alternatifs

**Algorithm 39** Algorithme d'analyse à arcs factorisés (Eisner)

**function** ArcFactored(**x**,n)
  C[i][i][d][c] $\leftarrow 0.0 \quad 0 \leqslant i \leqslant n, d \in \{\leftarrow, \rightarrow\}, c \in \{\top, \bot\}$
  **for** $1 \leqslant span \leqslant n$ **do**
    **for** $1 \leqslant i \leqslant n - span$ **do**
      $j \leftarrow i + span$
      ▷ Arbres incomplets
      C[i][j][$\leftarrow$][$\bot$] $\leftarrow \max_{i \leqslant k < j}(C[i][k][\rightarrow][\top] + C[k+1][j][\leftarrow][\top] + score(\mathbf{x}, j, i))$
      C[i][j][$\rightarrow$][$\bot$] $\leftarrow \max_{i \leqslant k < j}(C[i][k][\rightarrow][\top] + C[k+1][j][\leftarrow][\top] + score(\mathbf{x}, i, j))$
      ▷ Arbres complets
      C[i][j][$\leftarrow$][$\top$] $\leftarrow \max_{i \leqslant k < j}(C[i][k][\leftarrow][\top] + C[k][j][\leftarrow][\bot])$
      C[i][j][$\rightarrow$][$\top$] $\leftarrow \max_{i < k \leqslant j}(C[i][k][\rightarrow][\bot] + C[k][j][\rightarrow][\top])$
    **end for**
  **end for**
**end function**

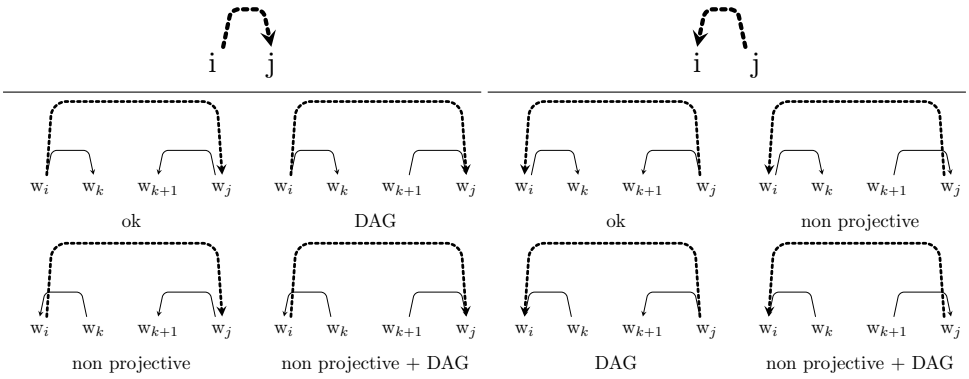



Figure 8.9: Cas admissibles pour les arbres incomplets

potentiels créent des structures en graphe ou des structures non projectives et sont donc interdits par l'algorithme.

**Arbres complets** La création des arbres incomplets par ajout de l'arc $i \rightarrow j$ permet de créer des sous-arbres de dépendances tels que le gouverneur $i$ domine l'ensemble des dépendants $k$ tels que $i \leqslant k \leqslant j$. Autrement dit, l'algorithme garantit la projectivité de l'analyse sur l'empan $i \dots j$. Par contre $i$ peut potentiellement dominer des mots $l$ tels que $l > j$ et tels que $i \xrightarrow{*} l$. C'est pour capturer ce manque qu'il y a une étape supplémentaire de création d'arbres complets. Celle-ci consiste à créer des liens de type $i \xrightarrow{*} l$ en concaténant un arbre incomplet $i \rightarrow j$ avec un arbre complet de la forme $j \xrightarrow{*} l$.

En résumé, les arbres incomplets relient directement $i \rightarrow j$ (ou $j \rightarrow i$) sur un empan $i \dots j$ alors que les arbres incomplets représentent des liens potentiellement indirects $i \xrightarrow{*} j$ (ou $j \xrightarrow{*} i$) qui combinent plusieurs empans.

**Complexité** L'intérêt de cet algorithme est qu'il permet d'obtenir un analyseur en dépendances projectif en $\mathcal{O}(n^3)$ alors que l'adaptation naïve de l'algorithme CKY a une complexité en $\mathcal{O}(n^5)$

Comparison with naive CYK

**Arc factored**

$$\textbf{init} \quad \langle i, i, d, c \rangle \qquad (0 \leqslant i < n, d \in \{\curvearrowright, \curvearrowleft\}, c \in \{\top, \bot\})$$

$$\textbf{reduce}(\curvearrowleft) \quad \frac{\langle i,k,\curvearrowright,\top\rangle \quad \langle k+1,j,\curvearrowleft,\top\rangle}{\langle i,j,\curvearrowleft,\bot\rangle}$$

$$\textbf{reduce}(\curvearrowright) \quad \frac{\langle i,k,\curvearrowright,\top\rangle \quad \langle k+1,j,\curvearrowleft,\top\rangle}{\langle i,j,\curvearrowright,\bot\rangle}$$

$$\frac{\langle i, k, \curvearrowright, \top\rangle \quad \langle k, j, \curvearrowright, \bot\rangle}{\langle i, j, \curvearrowright, \top\rangle} \quad complete(\curvearrowright)\ s.c.(k < j)$$

$$\frac{\langle i, k, \curvearrowleft, \bot\rangle \quad \langle k, j, \curvearrowleft, \top\rangle}{\langle i, j, \curvearrowleft, \top\rangle} \quad complete(\curvearrowleft)\ s.c.(k < j)$$

**Estimation des poids** L'analyse syntaxique par factorisation des arcs (*arc factored parsing*) suppose que le score d'un arbre se décompose comme la somme des scores de ses arcs :

$$\Psi(\mathbf{y}) = \sum_{(i,j)\in\mathbf{y}} \psi(\mathbf{x}, i, j) \tag{8.9}$$

On peut en principe utiliser n'importe quelle méthode d'apprentissage structurée (perceptron, large marge ou CRF) pour réaliser l'estimation de poids à partir de données.

Comme pour la plupart des algorithmes d'analyse syntaxique, l'estimation des poids par un CRF est un processus relativement lourd en pratique (mais pas impossible en théorie). Par conséquent, on présente ici une variante de l'algorithme du perceptron qui permet d'estimer les poids à partir d'un treebank de référence.

On suppose qu'un corpus annoté $C = (\mathbf{x}_i, \mathbf{y}_i)_{i=1}^N$ est un exemplaire de $N$ phrases où chaque séquence de mots $\mathbf{x}_i$ est associée à un arbre de référence $\mathbf{y}_i$. L'algorithme 40 est un variante de l'algorithme du perceptron vu précédemment qui tire parti de la décomposition en arcs de l'arbre de dépendances telle que posée en équation (8.9).

**Algorithm 40** Perceptron pour l'analyse factorisée en arcs

**function** ARC-FACTORED-PERCEPTRON($(\mathbf{x}_i, \mathbf{y}_i)_{i=1}^N$,$E$)
  $\mathbf{w} \leftarrow \bar{\mathbf{0}}$
  **for** $1 \leqslant e \leqslant E$ **do**
    **for** $1 \leqslant i \leqslant N$ **do**
      $\hat{\mathbf{y}} \leftarrow \text{argmax}_{\mathbf{y} \in \mathbf{Y}} \sum_{(i,j) \in \mathbf{y}} \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i, i, j)$ ▷ Parsing
      **if** $\hat{\mathbf{y}} \neq \mathbf{y}_i$ **then**
        $\mathbf{w} \leftarrow \mathbf{w} + \left[ \sum_{(i,j) \in \mathbf{y}_i} \mathbf{\Phi}(\mathbf{x}_i, i, j) - \sum_{(i,j) \in \hat{\mathbf{y}}} \mathbf{\Phi}(\mathbf{x}_i, i, j) \right]$
      **end if**
    **end for**
  **end for**
  **return w**
**end function**

## 8.4 Exercices

- Add I/O and eval functions
- Add dependency Labels
- Add bi-LSTM tagger input (and switch to feed forward scorer ?)
- Error analysis

# Chapter 9

# Convolutions

## 9.1 Origine

Les modèles de convolutions proviennent du traitement du signal. Étant donné un signal source représenté par une fonction du temps $x(t)$, et un filtre $h(t)$, la convolution des fonctions $x$ et $h$ est la fonction :

$$(x \circledast h)(t) = \sum_{i=-\infty}^{\infty} x(i)\, h(t-i)$$

pour le cas discret.

La fonction de filtre est une fonction qui en général vaut 0 pour la très grande partie du domaine temporel. Elle prend des valeurs non nulles plutôt autour de 0. À titre d'exemple, si:

$$h(j) = \begin{cases} 1 & \text{si } j = 0 \\ 0 & \text{sinon} \end{cases}$$

alors le filtre $h$ reproduit le signal $x$ à l'identique. Si maintenant on définit :

$$h(j) = \begin{cases} \frac{1}{3} & \text{si } -1 \leqslant j \leqslant 1 \\ 0 & \text{sinon} \end{cases}$$

alors le filtre $h$ produit une sortie qui moyenne la valeur $x(t)$ avec les valeurs $x(t \pm 1)$ trouvées dans une fenêtre autour de $t$. Intuitivement l'opération de convolution déplace le filtre comme une fenêtre glissante le long du domaine.

Lorsqu'on calcule $(x \circledast h)(t)$ pour tout le domaine temporel, Chaque valeur résultante est fonction des valeurs de $x$ autour de $t$. Par exemple, en supposant un signal discrétisé $x(t)$:

| $x(t)$ | 1 | 3 | 1 | 1 | 1 | 1 | 3 | 1 |
|---|---|---|---|---|---|---|---|---|
| $t$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

La convolution en $(x \circledast h)(1) = \frac{1}{3}(1 + 3 + 1) = \frac{5}{3}$. On peut calculer la convolution sur tout le domaine, ce qui donne:

| $x(t)$ | 1 | 3 | 1 | 1 | 1 | 1 | 3 | 1 |
|---|---|---|---|---|---|---|---|---|
| $t$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| $(x \circledast h)(t)$ | $\frac{4}{3}$ | $\frac{5}{3}$ | $\frac{5}{3}$ | 1 | 1 | $\frac{5}{3}$ | $\frac{5}{3}$ | $\frac{4}{3}$ |

On peut observer que dans ce cas-ci le filtre produit un effet de lissage autour des valeurs maximales de $x(t)$. Ici , il calcule une moyenne glissante.

**Filtre comme tableau de nombres** Observons toutefois que le filtre ne doit pas contenir que des paramètres constants, par exemple :

$$h(j) = \begin{cases} \frac{1}{2} & \text{si } j = \pm 1 \\ 2 & \text{si } j = 0 \\ 0 & \text{sinon} \end{cases}$$

Ce dernier filtre peut être noté alternativement comme un tableau de nombres de la forme :

| $h(t)$ | $\frac{1}{2}$ | 2 | $\frac{1}{2}$ |
|---|---|---|---|
| $t$ | -1 | 0 | 1 |

où seules les valeurs de $t$ pour lesquelles $h(t)$ est non nulle sont renseignées.

## 9.2 Implémentations

Les librairies numériques (Matlab, Numpy) proposent des fonctions de convolution qui tirent parti de l'intuition qu'un filtre peut être vu comme une fenêtre glissante de nombres que l'on fait progresser dans le temps. L'idée des implémentations est de ne pas calculer le grand nombre d'opérations de multiplication par 0 mais uniquement les opérations où le filtre a des valeurs non nulles.

Ainsi dans les librairies numériques où le vecteur $h$ est de longueur $h_N$ et celui-ci sera indexé à partir de 0, comme par exemple:

| $h(t)$ | $\frac{1}{2}$ | 2 | $\frac{1}{2}$ |
|---|---|---|---|
| $t$ | 0 | 1 | 2 |

La convolution est implémentée en itérant uniquement sur les valeurs de $h$

$$(x \circledast h)(t) = \sum_{j=0}^{h_N-1} x(t - j - 1 + \lceil h_N/2 \rceil) \quad h(j)$$

les valeurs de $j$ sont choisies pour que les indices soient valides pour les tableaux de nombres $x$ et $h$. Par exemple, supposons que l'on veuille calculer $(x \circledast h)(2)$ dans l'exemple ci-dessous.

| | | $\frac{1}{2}$ | 2 | $\frac{1}{2}$ | | | | |
|---|---|---|---|---|---|---|---|---|
| $x(t)$ | 0 | 0 | 3 | 0 | 0 | 0 | 3 | 0 |
| $t$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| $(x \circledast h)(t)$ | 0 | $\frac{3}{2}$ | 6 | $\frac{3}{2}$ | 0 | $\frac{3}{2}$ | 6 | $\frac{3}{2}$ |

La formule nous indique qu'il faut centrer le filtre autour de $t = 2$ (illustré en gris). Ce qui donne $\frac{1}{2} \times 0 + 2 \times 3 + \frac{1}{2} \times 0 = 6$

## 9.3 Convolutions et apprentissage profond

Les convolutions sont utilisées en deep learning principalement pour traiter les problèmes de vision artificielle. Mais une application au modèles de langage existe également. Celle-ci consiste à représenter un mot ou un caractère par un vecteur (embedding) à chaque étape $t$ temporelle. La convolution, au lieu de réaliser des opérations sur des scalaires, opère directement sur des vecteurs pour renvoyer un scalaire qui caractérise $t$.

# Appendix A

# Représentations pour les modèles creux

```
from collections import defaultdict

class SparseWeightVector:

    def  __init__(self):

        self.weights = defaultdict(int)

    def __call__(self,x_key,y_key):
        """
        This returns the weight of a feature couple (x,y)
        Enables an  x = w('a','b') syntax.

        @param x_key: a tuple of observed values
        @param y_key: a string being a class name
        @return : the weight of this feature
        """
        return self.weights[(x_key,y_key)]

    def dot(self,xvec_keys,y_key):
        """
        This computes the dot product : w . Phi(x,y).
        Phi(x,y) is implicitly  generated by the function given (x,y)
        @param xvec_keys: a list (vector) of x values
```

```
        @param y_key    : a y class name
        @return  w . Phi(x,y)
        """
        return sum([self.weights[(x_key,y_key)] for x_key in xvec_keys])

    @staticmethod
    def code_phi(xvec_keys,ykey):
        """
        Explictly generates a sparse boolean Phi(x,y) vector from (x,y) values
        @param xvec_keys:  a list of symbols
        @param ykey: a y class name
        """
        w = SparseWeightVector()
        for xkey in xvec_keys:
            w[(xkey,ykey)] = 1.0
        return w

    def __getitem__(self,key):
        """
        This returns the weight of feature couple (x,y) given as value.
        Enables the 'x = w[]' syntax.

        @param key: a couple (x,y) of observed and class value
        @return : the weight of this feature
        """
        return self.weights[tuple(key)]

    def __setitem__(self,key,value):
        """
        This sets the weight of a feature couple (x,y) given as key.
        Enables the 'w[] = ' syntax.
        @param key:   a couple (x,y) of observed value and class value
        @param value: a real
        """
        self.weights[key] = value

    def __add__(self,other):
            """
        Vector addition
        """
```

```
        weights =  self.weights.copy()
        for key,value in other.weights.items() :
            weights[key] += value
        w = SparseWeightVector()
        w.weights = weights
        return w

    def __sub__(self,other):
            """
        Vector substraction
        """
        weights =  self.weights.copy()
        for key,value in other.weights.items() :
            weights[key] -= value
        w = SparseWeightVector()
        w.weights = weights
        return w

    def __mul__(self,scalar):
            """
        Scalar multiplication
        """
        weights =  self.weights.copy()
        for key,value in self.weights.items() :
            weights[key] *= scalar
        w = SparseWeightVector()
        w.weights = weights
        return w

    def __rmul__(self,scalar):
            """
        commutativity of scalar multiplication
        """
        return self.__mul__(scalar)


    def __truediv__(self,scalar):
            """
        Python 3 division '/'
        """
```

```
        weights =  self.weights.copy()
        for key,value in self.weights.items() :
            weights[key] /= scalar
        w = SparseWeightVector()
        w.weights = weights
        return w


    def __iadd__(self,other):
        """
        Sparse Vector inplace addition. Enables the '+=' operator.
        @param  other: a  SparseVectorModel object
        """
        for key,value in other.weights.items():
            self.weights[key] += value
        return self
    def __isub__(self,other):
        """
        Sparse Vector inplace substraction. Enables the '-=' operator.
        @param  other: a  SparseVectorModel object
        """
        for key,value in other.weights.items():
            self.weights[key] -= value
        return self

    def load(self,istream):
        """
        Loads a model parameters from a text stream
        @param istream: an opened text stream
        """
        self.weights =  defaultdictionary(int)
        for line in istream:
            fields = line.split()
            key,value = tuple(fields[:-1]),float(fields[-1])
            self.weights[key] =  value

    def save(self,ostream):
        """
        Saves model parameters to a text stream
        @param ostream: an opened text output stream
```

```
            """
        for key,value in self.weights.items():
            print(' '.join(list(key)+[str(value)]),file=ostream)

    def __str__(self):
        """
        Pretty prints the weights vector on std output.
        May crash if vector is too wide/full
        """
        s = ''
        for key,value in self.weights.items():
            X,Y = key
            if isinstance(X,tuple):
                s += 'phi(%s,%s) = 1 : w = %f\n'%('&'.join(X),Y,value)
            else:
                s += 'phi(%s,%s) = 1 : w = %f\n'%(key,Y,value)
        return s
```

# Appendix A

# Exercice noté (descente de gradient)

Implémenter sur le même jeu de données avec Y catégorique :

- Descente de gradient (régression logistique hyperparamètres)
- Une descente de gradient en vecteur sparse de type (modèle à large marge ; utiliser le fichier `multiclass.py` et ajouter la méthode d'entrainement à large marge)
- Une descente de gradient multiclasse avec une librairie comme Keras
- Donnez l'accurracy du classifieur résultant sur les données.

# Appendix B

# Exercice Noté (tagger)

Le fichier `nn_tagger.py` propose un exemple de tagger unigramme avec un système de pondération basé sur un réseau de neurones à propagation avant.

- Méthode baseline : produire un tagger qui assigne à chaque mot le tag le plus fréquent observé avec les données
- Exercice principal (base) : transformer le tagger unigramme en tagger bigramme, ce qui implique de changer la forme du jeu de données pour l'entrainement et d'implémenter l'algorithme de Viterbi pour la prédiction. (ou de Dijkstra)
- Exercice principal (alternative) : implémenter un tagger bigramme de la famille MEMM
- Formalisez une méthode de gestion des mots inconnus.
- Evaluez votre tagger en divisant les données en train et en test.

Vous pouvez vous insipirer des fichiers `crf.py` et `structured_perceptron.py` pour vous aider à formuler la solution.