Rapport de synthèse du TIPE : Algorithme de *boosting* appliqué à la vision par ordinateur

Antoine Groudiev - n°15039

Introduction

La détection d'objets est la branche de la vision par ordinateur visant à classifier des images selon la présence ou l'absence d'un objet spécifique dans celles-ci. Je me suis intéressé à la détection de panneaux routiers dans un flux vidéo, et plus spécifiquement à un algorithme de *boosting*.

Le terme *boosting* désigne une famille d'algorithmes d'apprentissage automatique fonctionnant sur un principe commun : étant donné un ensemble de classificateurs faibles, qui classent chacun légèrement mieux que le hasard, l'algorithme de *boosting* sélectionne et pondère quelques classificateurs faibles pour former un classificateur fort, de bonne exactitude.

Mon travail a pour objectif la constitution d'un détecteur de panneaux STOP dans un flux vidéo. Je me suis donc intéressé à l'algorithme de boosting *AdaBoost* allié à la méthode de Viola et Jones pour étudier et mesurer l'efficacité de l'implémentation de ces méthodes de détection.

1 Algorithme de Viola et Jones

AdaBoost est un des algorithmes de *boosting* les plus populaires, notamment grâce à son utilisation par la méthode de Viola et Jones, un algorithme de reconnaissance de visages, présenté en 2001 [2].

Le détecteur doit pouvoir prendre en entrée des images de tailles quelconques, et retourner la liste des emplacements dans l'image de l'objet à détecter. La première phase de la création du détecteur se restreint cependant à la détection d'objets dans une image carrée de petite taille : j'ai fait le choix de `19px` de côté. La dernière partie de l'algorithme, détaillée en 1.5, appliquera ce détecteur à une image de taille standard, *i.e.* de plusieurs centaines de pixels de côté.

1.1 Classificateurs faibles

Les algorithmes de *boosting* fonctionnent par sélection de classificateurs faibles. Dans le contexte de la méthode de Viola et Jones, un classificateur faible est constitué de trois éléments.

1.1.1 Les *features*

Une *feature* est constituée de 2 à 4 régions rectangulaires adjacentes, comptées positivement ou négativement. Leurs formes sont imposées comme dans la figure 1. Chaque *feature* va cibler une zone spécifique de l'objet à détecter. Dans le cas d'un visage par exemple, le détecteur peut apprendre que la zone du creux de l'œil est généralement plus sombre que la zone entre les deux yeux. Ainsi, une image comportant cette différence de luminosité caractéristique sera probablement un visage.

Le score d'une *feature* f peut être évalué sur une image x à l'aide de la formule suivante (le score le plus faible en valeur absolue étant le meilleur) :

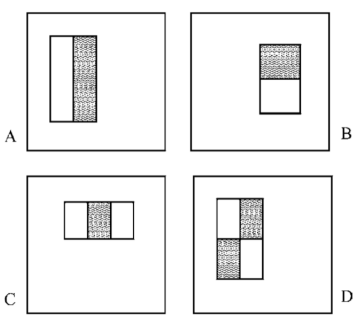

FIGURE 1 – Forme des *features*

$$f(x) = \sum_{r \in R_+} r(x) - \sum_{r \in R_-} r(x) \tag{1}$$

où $r(x)$ désigne la somme des pixels dans la région délimitée par r. Une méthode efficace du calcul de $r(x)$ sera donnée par l'équation 4. Intuitivement, la formule 1 traduit que le score d'une *feature* est d'autant plus faible que les zones négatives compensent les zones positives.

Le nombre de *features* possibles croît exponentiellement avec le côté de l'image (voir la figure 2), d'où la nécessité d'entraîner dans un premier temps un détecteur de côté faible. La construction de toutes les *features* possibles est implémentée en Partie 5.3.1 de l'annexe.

1.1.2 Évaluation par un classificateur faible

Pour former un classificateur faible, l'algorithme associe à chaque *feature* (notée f) un *threshold* (ou seuil) $\theta > 0$, et une polarité $p \in \{-1; 1\}$. Considérons x une image de `19px` de côté. Le classificateur faible $C^{faible}_{(f,\theta,p)}$ convertit le score de la *feature* sur x en un booléen selon la loi suivante :

$$C^{faible}_{(f,\theta,p)}(x) = \begin{cases} 1 \ si \ \ pf(x) < p\theta \\ 0 \ \ sinon \end{cases} \tag{2}$$

Ainsi, si le score de la *feature* sur x est, à la polarité près, sous le seuil, alors le classificateur faible juge que la zone de l'image correspond à l'objet à détecter.

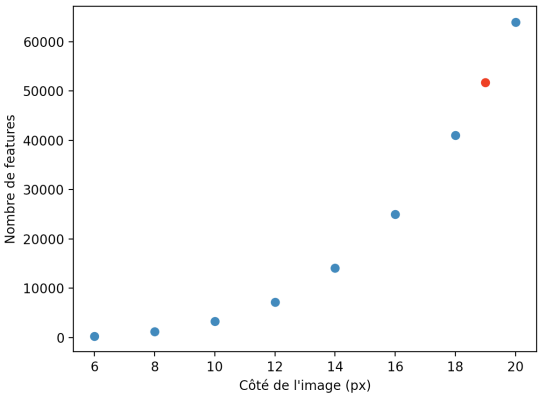

FIGURE 2 – Nombre de *features* en fonction de la taille de l'image

1.2 Image intégrale

L'équation 1 montre que la complexité du calcul du score d'un classificateur faible est déterminée par la complexité du calcul de la somme des valeurs des pixels dans un sous-rectangle de l'image.

Une approche naïve consisterait à recalculer, à chaque évaluation du score d'un classificateur, la somme des valeurs des pixels de l'image dans certaines de ses régions rectangulaires. Un tel calcul pour une région de taille L_r sur l_r a une complexité en $O(L_r \times l_r)$. Si l'on considère une image de dimensions $n \times n$ et que l'on veut calculer la somme dans p régions de dimensions proches de $n \times n$, la complexité totale est en $O(p \times n^2)$.

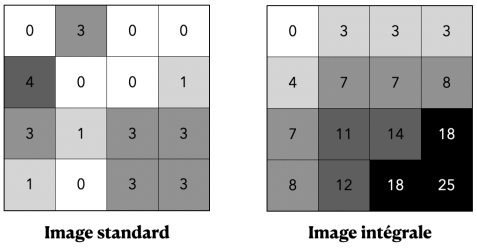

FIGURE 3 – Exemple d'image intégrale

Pour réduire cette complexité, la méthode de Viola et Jones introduit **l'image intégrale**. Si l'image considérée est $(i(x, y))$, alors l'image intégrale, notée $(ii(x, y))$ vérifie :

$$\forall x, y, ii(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y') \tag{3}$$

Intuitivement, chaque coefficient de l'image intégrale contient la somme des coefficients en haut à gauche du pixel considéré. Ceci permet alors d'accéder en temps constant à la somme des pixels dans un sous-rectangle de l'image $(i(x, y))_{x,y}$, en réalisant simplement la somme de quatre termes. En effet, considérons R le sous-rectangle délimité par les sommets (x_1, y_1) et (x_2, y_2). Alors la somme $r(x)$ des pixels de l'image x dans la région R vaut :

$$\boxed{r(x) = ii(x_2, y_2) - ii(x_2, y_1) - ii(x_1, y_2) + ii(x_1, y_1)} \tag{4}$$

1.3 Sélection des caractéristiques par *AdaBoost*

La phase de *boosting* vise à sélectionner un nombre $T \in \mathbb{N}^*$ de classificateurs faibles qui représentent le mieux l'objet à détecter. L'algorithme utilise pour cela en entrée un jeu d'entraînement, c'est-à-dire une liste de tuples $(x, y) \in \mathscr{M}_{19,19}(\llbracket 0, 255 \rrbracket) \times \{0; 1\}$.

Chaque tuple est constitué d'une image contenant ou ne contenant pas l'objet à détecter, centré et cadré le cas échéant, et d'un label booléen, valant 1 si l'objet à détecter est effectivement représenté sur l'image (on identifiera `True` (respectivement `False`) à `1` (respectivement `0`)). La constitution d'un tel jeu sera détaillée dans la Partie 2.

FIGURE 4 – Exemple de deux tuples du jeu d'entraînement

L'algorithme *AdaBoost* est un algorithme glouton qui sélectionne un à un les T meilleurs classificateurs parmi les 50 000 présents dans l'image de `19px` de côté. Son initialisation consiste en l'affectation à chaque image d'un poids, qui équilibre l'importance des images positives et négatives. Ensuite, la sélection d'un classificateur se fait en trois grandes étapes : l'erreur de chaque classificateur est calculée selon le classement qu'il fait de chaque image, et de son poids ; le classificateur d'erreur minimale est sélectionné ; les poids sont mis à jours pour prendre en compte le nouveau classificateur, puis normalisés.

Une implémentation en est également donnée en Partie 5.3.2 de l'annexe.

Algorithme 1 Entraînement par AdaBoost

Input
$(x_1, y_1), ..., (x_n, y_n)$ ▷ Jeu d'entraînement
(C_k^{faible}) ▷ Tous les classificateurs constructibles dans l'image
$m \leftarrow$ nombre d'images négatives
$l \leftarrow$ nombre d'images positives
for $i \in \llbracket 1, n \rrbracket$ **do** ▷ Initialisation des poids

$$w_{1,i} \leftarrow \begin{cases} \frac{1}{m} \ si \ y_i = 0 \\ \frac{1}{l} \ sinon \end{cases}$$

▷ Le premier indice désigne l'indice de l'itération
▷ Le second désigne l'indice de l'image
end for
for $t \in \llbracket 1, T \rrbracket$ **do**
  for $k \in \llbracket 1,$ nombre de classificateurs$\rrbracket$ **do**
    $\varepsilon_k = \sum_i w_{t,i} \times \delta(C_k^{faible}(x_i), \overline{y_i})$ ▷ Calcul de l'erreur de chaque classificateur
  end for
  $C_t^{faible} \leftarrow \text{argmin}_{C_k^{faible}} \varepsilon_k$
  for $i \in \llbracket 1, n \rrbracket$ **do** ▷ Mise à jour des poids
    if image x_i bien classée par C_t^{faible} **then**
      $w_{t+1,i} \leftarrow w_{t,i} \times \frac{\varepsilon_t}{1-\varepsilon_t}$ ▷ le poids baisse à $t+1$
    else
      $w_{t+1,i} \leftarrow w_{t,i}$ ▷ le poids augmente à $t+1$
    end if
  end for
  Normaliser les poids : $w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$
end for

En toute généralité, la complexité de *AdaBoost* est en $O(n \cdot F \cdot \tau + T \cdot n)$ où n désigne le nombre d'images d'entraînement, F le nombre total de classificateurs faibles, et $\tau(n)$ le temps moyen de classification d'un classificateur faible sur un x_i. On a $T = O(F)$ et dans le cas de Viola-Jones, l'image intégrale garantit $\tau(n) = O(1)$, ce qui donne une complexité en $O(n \cdot F)$.

Après sélection des T classificateurs faibles, l'algorithme les combine en un unique classificateur fort C^{fort}, défini par :

$$C_T^{fort}(x) = \begin{cases} 1 \ si \ \sum_{t=1}^{T} \alpha_t C_t^{faible}(x) \leq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 \ sinon \end{cases} \tag{5}$$

où les $\alpha_t = \log(\frac{1-\varepsilon_t}{\varepsilon_t})$ pondèrent les classificateurs selon leur erreur. Ainsi, aux pondérations près, si au moins la moitié des classificateurs faibles retournent 1, le classificateur fort retourne 1. L'expression des α_t garantit la diminution à chaque étape de l'erreur globale du classificateur. Une preuve mathématique en est donnée en Partie 5.2 de l'annexe.

1.4 Mise en cascade

Selon sa valeur de T, un classificateur fort est soit très efficace (pour T faible), soit très exact (pour T élevé). L'introduction du concept de cascade permet d'allier exactitude et efficacité.

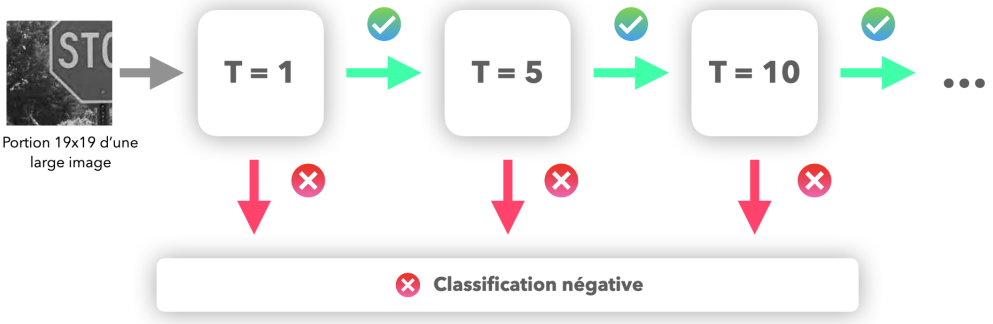

FIGURE 5 – Schéma d'une cascade

Une image analysée par la cascade va être classée successivement par une suite $(C_T^{fort})_T$ pour des valeurs de T croissantes. Une image est alors rejetée par la cascade dès qu'elle est classée négativement par un des classificateurs forts, permettant une grande efficacité. Au contraire, une image ultimement classée positivement par la cascade sera passée à travers des classificateurs forts avec T très grands, garantissant un faible nombre de faux positifs.

Une simple implémentation de la cascade est donnée en Partie 5.3.3 de l'annexe.

1.5 Application à des images de taille standard

L'application du détecteur à une image standard se fait en analysant des sous-fenêtres carrées de la grande image.

Le détecteur ne pouvant analyser l'intégralité des millions de sous-fenêtres de l'image, l'analyse est déterminée par deux paramètres : le paramètre Δ, décalage entre deux sous-fenêtres de même taille, et le paramètre s (pour *scaling*), rapport des tailles de deux sous-fenêtres de côtés différents.

Le choix de ces deux paramètres permettra de contrôler la relation entre exactitude et efficacité du détecteur, commme observé dans les résultats de la Partie 3.3.

2 Pré-traitement des images d'entaînement et de test

Viola et Jones ont utilisé pour l'entraînement de leur algorithme une base de données constituée de 4916 images positives et 9544 images négatives. Le nombre d'images positives a même été doublé en

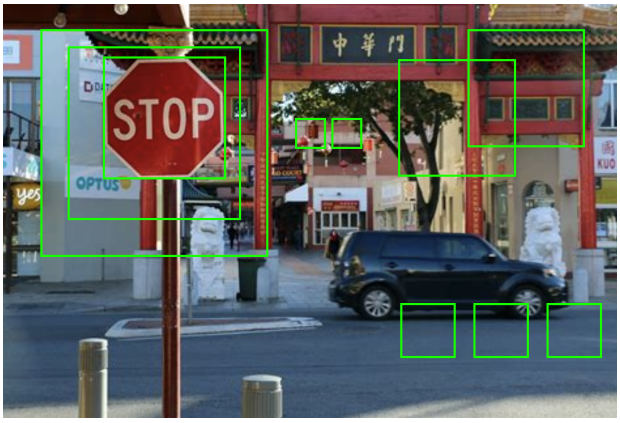

FIGURE 6 – Exemples de sous-fenêtres d'une image de taille standard

introduisant les symétriques verticaux, ce qui est impossible dans mon cas en raison de l'asymétrie de la plupart des panneaux.

Une telle quantité d'images n'existe pas pour l'objet sur lequel je me suis concentré, le panneau STOP, ce qui a naturellement une influence sur les résultats de mon implémentation. J'ai pû réunir 400 images de panneaux STOP, que j'ai pré-traitées avant d'entraîner le détecteur.

FIGURE 7 – Étapes du pré-traitement d'une image

La première étape consiste en un recadrage manuel de l'image. La seconde, réalisée par un script, redimensionne l'image vers `19px` de côté par interpolation bilinéaire, et la convertit en niveaux de gris, chaque pixel appartenant finalement à $[\![0, 255]\!]$ (`8 bits`).

Les images traitées sont finalement réparties en deux ensembles : un jeu d'entraînement et un jeu de test, pour éviter de tester le détecteur sur des images qu'il a déjà rencontré.

3 Mesure de l'exactitude et de l'efficacité de l'implémentation

Le langage courant confond l'exactitude, c'est à dire la proximité du résultat expérimental à la valeur théorique, et la précision, qui quantifie la dispersion des résultats. L'objectif de cette dernière partie vise à déterminer précisément l'exactitude du détecteur précédemment implémenté.

3.1 Quantification de l'exactitude

Toute quantification de l'exactitude est une expression des coefficients de la **matrice de confusion** d'un détecteur, matrice qui compare le classement du détecteur au label réel (représentée sur la figure 8).

3.1.1 Approche standard

Il semble intuitif de poser l'exactitude comme étant :

$$A = \frac{bons\ classements}{total}$$

Ou encore avec les notations de la figure 8 :

$$\boxed{A = \frac{V_p + V_n}{V_P + V_n + F_p + F_n}} \tag{6}$$

	Classé : P	**Classé : N**
Réel : P	V_p	F_n
Réel : N	F_p	V_n

FIGURE 8 – Matrice de confusion

Cependant, cette méthode de calcul introduit des biais en cas de déséquilibre important entre le nombre d'images positives et le nombre d'images négatives. Mes échantillons de tests étant fortement déséquilibrés, il faut introduire une nouvelle méthode de calcul de l'exactitude.

3.1.2 F-Score

On introduit alors souvent le F-Score (ou F_1-Score), défini comme la moyenne harmonique de la précision et du rappel. [5]

La précision et le rappel sont définis comme :

$$P = \frac{bons\ classements}{classements\ positifs} = \boxed{\frac{V_p}{V_P + F_p}} \qquad R = \frac{bons\ classements}{images\ positives} = \boxed{\frac{V_p}{V_P + F_n}} \tag{7}$$

Finalement, le F-Score est donnée comme la moyenne harmonique des deux :

$$\boxed{F_1 = \frac{2}{\frac{1}{P} + \frac{1}{R}} = \frac{2PR}{P + R}} \tag{8}$$

C'est cette grandeur qui m'a servi à juger de l'exactitude de mon détecteur dans les partie suivantes.

3.2 Résultats du détecteur de `19px`

Après entraînement sur un jeu d'images de panneaux STOP, j'ai obtenu les résultats suivants :

T par couche	Jeu d'entraînement	Jeu de test	Exactitude (standard)	Exactitude (F-Score)	Temps moyen de classification
1, 5, 10	324 / 4548 / 1 :14	69 / 3450 / 1 :50	96,7 %	81,3 %	0,633 ms
1, 5, 10, 20, 50			98,6 %	91,5 %	0,696 ms

On remarque que l'augmentation du nombre de couches du détecteur dans la seconde ligne améliore l'exactitude du détecteur, au coût d'un temps moyen de classification plus élevé. Le temps de classification est néanmoins loin d'être linéaire en le nombre total de classificateurs faibles, ce qui est logique puisque seules les images positives prennent sensiblement plus de temps à être traitées.

3.3 Résultats du détecteur de taille standard

J'ai par la suite testé mon détecteur sur des images de taille standard, et ai obtenu les résultats suivants :

T par couche	Δ	s	Jeu de test	Exactitude (standard)	Exactitude (F-Score)	Temps moyen de classification	FPS
1, 5, 10, 20, 50	3	1,5	302 / 433 / 1 :1,43	96,3 %	81,5%	0,17 s	5,9
	2	1,25		98,2 %	89,4 %	0,35 s	2,8

Des valeurs plus faibles du couple (Δ, s) ont tendance à augmenter le F-Score, mais en augmentant le temps moyen de classification.

4 Conclusion

L'apprentissage automatique par *AdaBoost* et son utilisation dans le cadre de la méthode de Viola et Jones s'avère être un algorithme intuitivement simple, de par son charactère glouton, mais néanmoins efficace. Le principe de *boosting* me semble être généralisable à d'autres domaines de l'apprentissage supervisé.

5 Annexes

5.1 Complexité de l'image intégrale

En notant $(i(x,y))$ l'image source et $(ii(x,y))$ l'image intégrale, on a la formule de récurrence suivante :

$$\begin{cases} ii(1,1) = i(1,1) \\ ii(x,y) = i(x,y) + ii(x-1,y) + ii(x,y) - ii(x-1,y-1) \end{cases} \tag{9}$$

Ceci permet de calculer (ii) avec une complexité linéaire en le nombre de pixels de l'image, soit la même complexité que pour le seul calcul de la somme de tous les pixels dans l'image, correspondant au sous-rectangle maximal.

5.2 Justification formelle du fonctionnement d'*AdaBoost*

On considère un jeu d'entraînement $(x_i, y_i) \in \mathscr{M}_{n,n}(\mathbb{R}) \times \{-1, 1\}$ (un label négatif est ici représenté par $y_i = -1$ et non pas par $y_i = 0$). Montrons que l'Algorithme 1 est d'exactitude croissante. On définit pour cela la fonction de perte exponentielle L [4] :

$$L(C^{fort}(x), y) := \exp(-y \cdot C^{fort}(x)) \tag{10}$$

où l'on a défini le classificateur fort suivant, constitué de T classificateurs faibles :

$$C^{fort}(x) = \sum_{t=1}^{T} \alpha_t \cdot C_t^{faible}(x)$$

L'objectif de l'algorithme est de choisir les $(C_t^{faible})_{1 \leq t \leq T}$ et les $(\alpha_t)_{1 \leq t \leq T}$ qui minimisent la fonction de perte, c'est à dire :

$$\min_{(\alpha_t)_{1 \leq t \leq T}, (C_t^{faible})_{1 \leq t \leq T}} w_{1,i} \cdot L\left(\sum_{t=1}^{T} \alpha_t \cdot C_t^{faible}(x_i), y_i \right)$$

Supposons alors $t-1$ classificateurs choisis et posons Z_{t+1} le minimum de la fonction de perte après le choix du t-ième classificateur :

$$Z_{t+1} := \min_{\alpha_t, C_t^{faible}} \sum_{i=1}^{n} D_i(t+1) = \min_{\alpha_t, C_t^{faible}} \sum_{i=1}^{n} w_{1,i} \exp\left(-y_i \cdot \sum_{s=1}^{t} \alpha_s C_s^{faible}(x_i) \right) \tag{11}$$

En factorisant il vient :

$$\begin{aligned} Z_{t+1} &= \min_{\alpha_t, C_t^{faible}} \sum_{i=1}^{n} D_i(t) \exp\left(-y_i \alpha_t C_t^{faible}(x_i) \right) \\ &= \min_{\alpha_t, C_t^{faible}} e^{-\alpha_t} \sum_{i=1}^{n} D_i(t) \cdot \delta(y_i, C_t^{faible}(x_i)) + e^{\alpha_t} \sum_{i=1}^{n} D_i(t) \cdot \delta(-y_i, C_t^{faible}(x_i)) \\ &= \min_{\alpha_t, C_t^{faible}} e^{-\alpha_t} \sum_{i=1}^{n} D_i(t) + (e^{\alpha_t} - e^{-\alpha_t}) \sum_{i=1}^{n} D_i(t) \cdot \delta(-y_i, C_t^{faible}(x_i)) \\ &= Z_t \min_{\alpha_t, C_t^{faible}} e^{-\alpha_t} + (e^{\alpha_t} - e^{-\alpha_t}) \sum_{i=1}^{n} \frac{D_i(t)}{Z_t} \cdot \delta(-y_i, C_t^{faible}(x_i)) \end{aligned}$$

La forme de cette expression sépare les deux variables α_t et C_t^{faible} sur lesquelles portent le minimum. Cela justifie que l'on peut sélectionner dans un premier temps le classificateur C_t^{faible}, puis ensuite choisir α_t.

Pour cela, on définit naturellement l'erreur minimale et le classificateur associé :

$$\varepsilon_t := \min_{C^{faible}} \sum_{i=1}^{n} \frac{D_i(t)}{Z_t} \cdot \delta(-y_i, C_t^{faible}(x_i))$$

$$C_t^{faible} := \operatorname*{argmin}_{C^{faible}} \sum_{i=1}^{n} \frac{D_i(t)}{Z_t} \cdot \delta(-y_i, C_t^{faible}(x_i))$$

Et l'on choisit dans un second temps le α_t qui minimise le tout :

$$\alpha_t := \operatorname*{argmin}_{\alpha>0} e^{-\alpha} + (e^{\alpha} - e^{-\alpha})\varepsilon_t = \frac{1}{2} \ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$$

Le calcul de l'argument minimisant cette quantité pouvant être fait par une simple étude de fonction. Imposer $\alpha > 0$ revient à demander $\varepsilon_t > \frac{1}{2}$, soit que au moins un classificateur faible classe mieux que le hasard, ce qui est un prérequis de tout algorithme de *boosting*. Sous ces conditions :

$$Z_{t+1} = Z_t \times 2\sqrt{\varepsilon_t \cdot (1-\varepsilon_t)} \leq Z_t \tag{12}$$

Ce qui montre bien la décroissance de la fonction de perte du classificateur fort.

5.3 Éléments significatifs du code

5.3.1 Génération de toutes les *features*

```python
def construction_features(format_image: tuple) -> list:
    hauteur, largeur = format_image
    features = []

    for l in range(1, largeur+1):
        for h in range(1, hauteur+1):
            # Pour toutes les dimensions possibles de rectangles
            x = 0
            while x + l < largeur:
                y = 0
                while y + h < hauteur:
                    # Pour tous les rectangles possibles

                    # Creation des sous-zones possibles
                    normal = RegionRectangulaire(x, y, w, h)
                    droite = RegionRectangulaire(x + w, y, w, h)
                    bas = RegionRectangulaire(x, y + h, w, h)

                    droite_2 = RegionRectangulaire(x + 2*w, y, w, h)
                    bas_2 = RegionRectangulaire(x, y + 2*h, w, h)

                    bas_droite = RegionRectangulaire(x + w, y + h, w, h)

                    # Features avec 2 rectangles
                    if x + 2*w < width: # Adjacent horizontalement
                        features.append(([droite], [normal]))
                    if y + 2*h < height: # Adjacent verticalement
                        features.append(([direct], [bas]))

                    # Features avec 3 rectangles
                    if x + 3 * w < width: # Adjacent horizontalement
                        features.append(([droite], [droite_2, normal]))
                    if y + 3 * h < height: # Adjacent horizontalement
                        features.append(([bas], [bas_2, normal]))

                    # Features avec 4 rectangles
```

```
                    if x + 2 * l < largeur and y + 2 * h < hauteur:
                        features.append(([droite, bas], [normal, bas_droite]))

                    y += 1
                x += 1
    return np.array(features, dtype=object)
```

5.3.2 Implémentation de *AdaBoost*

```
def AdaBoost(jeu_entrainement: list) -> ClassificateurFaible list * int list:
    classificateurs, alpha = [], []

    # Comptage des exemples positifs et negatifs
    nb_pos = 0
    nb_neg = 0
    for (image, y) in jeu_entrainement:
        if y:
            nb_pos += 1
        else:
            nb_neg += 1

    # Initialisation des poids et conversion en image integrale
    donnees = []
    poids = np.zeros(len(jeu_entrainement))
    for i in range(len(jeu_entrainement)):
        donnees.append((integrale(jeu_entrainement[i][0]), jeu_entrainement[i][1]))
        if jeu_entrainement[i][1]:
            poids[i] = 1.0 / (2 * nb_pos)
        else:
            poids[i] = 1.0 / (2 * nb_neg)

    # Construction des features et calcul de l'erreur sur chaque image
    features = construction_features(donnees[0][0].shape)
    X, y = application_features(features, donnees)

    for t in range(len(features)):
        # Selection du classificateur avec l'erreur la plus faible
        c_faibles = train_weak_classifiers(X, y, features, weights)

        clf, erreur, classements = meilleur_classificateur(c_faibles, weights,
                                                            donnees)
        classificateurs.append(clf)

        # Calcul de beta et mise a jour des poids
        beta = erreur / (1.0 - erreur)
        for i in range(len(classements_jeu)):
            poids[i] = poids[i] * (beta ** (1 - classements[i]))
        alpha = -np.log(beta)
        alphas.append(alpha)

        # Normalisation des poids
        poids = poids / np.linalg.norm(poids)

    return classificateurs, alpha
```

5.3.3 Classement par la Cascade

```
def classement_cascade(cascade: list, image:list) -> bool:
    ii = integrale(image)
    for clf in cascade: # pour chaque classificateur fort de la cascade
        if not clf.classifier(ii): # des que l'image est classee negativement
```

```
            return False # la cascade classe negativement
        return True # si l'image passe tous les classificateurs, elle est positive
```

Références

[1] Richard Szeliski, *Computer Vision : Algorithms and Applications, 2nd ed. (2022)*, https://szeliski.org/Book/

[2] Paul Viola, Michael Jones, *Rapid Object Detection using a Boosted Cascade of Simple Features*, Conference on Computer Vision and Pattern Recognition, https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf

[3] Michael Pound, Sean Riley, Computerphile, *Detecting Faces (Viola Jones Algorithm)*, https://www.youtube.com/watch?v=uEJ71VlUmMQ&t=15s

[4] Yi-Qing Wang, *An Analysis of the Viola-Jones Face Detection Algorithm*, IPOL, https://www.ipol.im/pub/art/2014/104/?utm_source=doi

[5] David M W Powers, *Evaluation : From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation*, Journal of Machine Learning Technologies, https://web.archive.org/web/20191114213255/https://www.flinders.edu.au/science_engineering/fms/School-CSEM/publications/tech_reps-research_artfcts/TRRA_2007.pdf