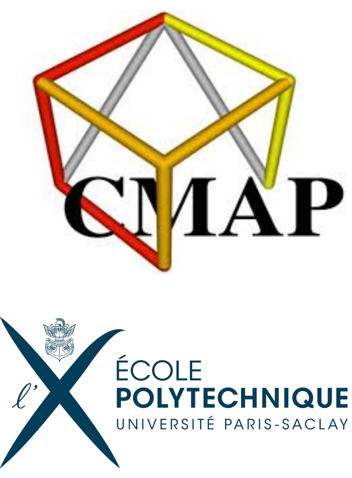


Centre de Mathématiques Appliquées - École Polytechnique

Equipe SIMPAS : Statistique Apprentissage Simulation Image

Deep Reinforcement Learning for Visual Question Answering

Auteur :
Ayoub Abraich

Encadrants :
Pr. Éric Moulines
Ing. Alice Martin

Rapport de Stage

M1 Mathématiques et Interactions - Université Paris Saclay

30 août 2019

Résumé

La conception de bout en bout des systèmes de dialogue est récemment devenue un sujet de recherche populaire grâce à des outils puissants tels que des architectures codeur-décodeur pour l'apprentissage séquence à séquence. Pourtant, la plupart des approches actuelles considèrent la gestion du dialogue homme-machine comme un problème d'apprentissage supervisé, visant à prédire la prochaine déclaration d'un participant, compte tenu de l'historique complet du dialogue. Cette vision est aussi simpliste pour rendre le problème de planification intrinsèque inhérent au dialogue ainsi que sa nature enracinée, rendant le contexte d'un dialogue plus vaste que seulement l'historique. C'est la raison pour laquelle seules les tâches de bavardage et de réponse aux questions ont été traitées jusqu'à présent en utilisant des architectures de bout en bout. Dans ce rapport, nous présentons une méthode d'apprentissage par renforcement profond permettant d'optimiser les dialogues axés sur les tâches, basés sur l'algorithme policy gradient. Cette approche est testée sur un ensemble de données de 120 000 dialogues collectés via Mechanical Turk et fournit des résultats encourageants pour résoudre à la fois le problème de la génération de dialogues naturels et la tâche de découvrir un objet spécifique dans une image complexe.

Remerciements

Je tiens à remercier mes deux encadrants de stage Eric Moulines et Alice Martin qui m'ont guidé avec cordialité et bienveillance durant ces quatre mois . Merci beaucoup Alice de m'avoir accordé pour votre accueil, votre confiance , le temps passé ensemble et le partage de votre expertise au quotidien.

De même, J'adresse mes chaleureux remerciements à tous mes enseignants, qui m'ont aidé tout au long de l'année . En particulier, Mme Agathe Guilloux et Mme Marie Luce Taupin pour leurs efforts et leur soutien avec cordialité et bienveillance durant mon parcours à l'université Evry Val d'Essonne.

Enfin, je tiens à remercier toutes les personnes qui m'ont aidé et conseillé et relu lors de la rédaction de ce rapport de stage : ma famille, mes camarades de classe et mes amis. Merci !

Table des matières

1 Introduction **5**
1.1 État de l'art . . . 5
1.2 Problématique et plan de travail . . . 7

2 GuessWhat ?! **8**
2.1 Règles de jeu . . . 8
2.2 Notations . . . 8
2.3 Image captioning . . . 8
2.4 Visual Question Answering . . . 9
2.5 Dialogue dirigé par objectif . . . 9
2.6 Quelques notions et définitions . . . 9
2.7 Environnement d'apprentisssage . . . 11
2.7.1 Génerations de questions . . . 12
2.7.2 Oracle . . . 12
2.7.3 Devineur . . . 13
2.8 Génération de jeux complets . . . 13
2.9 GuessWhat ?! du point de vue de RL . . . 14
2.9.1 GuessWhat ?! en tant que processus de décision de Markov . . . 14
2.9.2 Entraînement de QGen avec Policy Gradient . . . 15
2.9.3 Fonction de récompense . . . 15
2.9.4 Procédure d'entraînement complète . . . 16
2.10 Expériences . . . 17
2.10.1 Détails de l'entraînement . . . 17
2.10.2 Résultats . . . 17

3 Apprentissage automatique **20**
3.1 Généralités . . . 20
3.2 Apprentissage supervisé . . . 21
3.2.1 Approche algorithmiques . . . 21
3.2.2 Formalisme . . . 21
3.2.3 Optimisation des paramètres . . . 23
3.2.4 Limitations . . . 26
3.3 Apprentissage par renforcement . . . 28
3.4 Apprentissage non supervisé . . . 28
3.4.1 Réseaux de neurones . . . 29

4 Apprentissage par renforcement **31**
4.1 Introduction . . . 31
4.2 Formalisation . . . 32
4.2.1 Le cadre d'apprentissage du renforcement . . . 32
4.2.2 La propriété de Markov . . . 32
4.2.3 Différentes catégories de politiques . . . 33
4.2.4 Le retour attendu et fonction valeur . . . 33

4.2.5 Fonction action-valeur . . . 34
4.2.6 Rétropropagation de la valeur . . . 35
4.3 Différents composants pour apprendre une politique . . . 35
4.4 Différentes configurations pour apprendre une politique à partir de données . . . 35
4.4.1 Offline & online learning . . . 35
4.4.2 Comparaison entre l'off-policy et l'on-policy learning . . . 36
4.5 Méthodes basées sur la valeur pour deep RL . . . 36
4.5.1 Q-learning . . . 37
4.5.2 Q-learning ajusté . . . 37
4.5.3 Deep Q-networks . . . 38
4.5.4 Perspective distributionnelle de RL . . . 38
4.5.5 Multi-step learning . . . 44

5 L'apprentissage profond 45
5.1 Approche . . . 45
5.2 Optimalité globale en apprentissage profond . . . 47
5.2.1 Le défi de la non convexité dans l'apprentissage en réseau de neurones . . . 47
5.3 Stabilité géométrique en apprentissage profond . . . 48
5.4 Théorie basée sur la structure pour l'apprentissage profond . . . 50
5.4.1 Structure des données dans un réseau de neurones . . . 50
5.5 Etat de l'art . . . 50
5.6 RNN standards . . . 51
5.6.1 Introduction . . . 51
5.6.2 Limitations et motivations . . . 52
5.6.3 Les racines de RNN . . . 53
5.7 LSTM . . . 54
5.7.1 Introduction . . . 54
5.7.2 Principe . . . 54
5.7.3 Architecture . . . 55
5.7.4 Variantes . . . 55
5.7.5 Contexte & résultats . . . 56

6 Conclusion 58

Bibliography 60

A Implémentation des algorithmes utilisés dans RL 61
A.1 Exemple d'implémentation de l'algorithme N-Step TD . . . 61
A.2 Une implémentation fonctionnelle du DQN catégorique (distributional RL). . . . 64
A.2.1 Utils : . . . 64
A.2.2 Agents : . . . 73
A.2.3 Estimateurs : Exemple - Jeu Aatari . . . 78
A.2.4 Evaluation catégorique de la politique . . . 79

B Implémentation des variantes LSTM 80
B.1 Implémentation : LSTM simple pour la classification binaire . . . 80
B.1.1 Fonctions auxiliaires . . . 80
B.1.2 Data . . . 81
B.1.3 LSTM Network Architecture . . . 84
B.2 Implémentation : LSTM simple pour la classification multiclass . . . 88
B.3 Implémentation : LSTM Child Sum pour la cassification binaire . . . 103
B.4 Implémentation : LSTM child sum multiclass . . . 115
B.5 Parsing Coco Data set . . . 127

Table des figures

2.1 L'architecture de VGG16 10
2.2 Algorithme : Diverse Beam Search 11
2.3 Modèle de génération de questions 12
2.4 Modèle de l'Oracle 13
2.5 Modèle du Devineur 14
2.6 Entraînement de QGen avec REINFORCE [1] 16
2.7 Précisions des modèles de la performance humaine du QGen formé avec baseline et REINFORCE. Les nouveaux objets font référence à l'échantillonnage uniforme des objets dans l'ensemble d'apprentissage, tandis que les nouvelles images font référence à l'ensemble de test. 17
2.8 Échantillons extraits de l'ensemble de test. Le cadre bleu (resp. Violet) correspond à l'objet choisi par le devineur pour le dialogue de recherche de faisceau - beam search- (resp. REINFORCE). La petite description verbeuse est ajoutée pour faire référence à l'objet sélectionné par le devineur. 18
2.9 Rapport d'achèvement des tâches de QGEN formé par REINFORCE en fonction de la longueur de dialogue 19

3.1 Les trois grandes classes d'apprentissage automatique 20
3.2 Récapitulatif des méthodes d'optimisation et leurs hypothèses. 23
3.3 Exemple de sur/sous-apprentissage [OpenClassrooms] 27
3.4 Un neurone formel 29
3.5 Un perceptron multicouche avec $X = \mathbb{R}^4$ et une couche cachée de 3 neurones. 30

4.1 Interaction agent-environnement dans RL 32
4.2 Schéma général des méthodes RL profond 36
4.3 Esquisse de l'algorithme DQN 39
4.4 Pour deux politiques illustrées sur la figure (a), l'illustration sur la figure (b) donne la distribution de $Z^{(\pi)}(s, a)$ comparée à la valeur attendue $Q^{\pi}(s, a)$. Sur la figure de gauche, on peut voir que π_1 passe avec certitude à un état absorbant avec une récompense à chaque pas $\frac{R_{max}}{5}$, tandis que π_2 se déplace avec une probabilité de 0,2 et 0,8 dans des états absorbants avec des récompenses à chaque pas respectivement R_{max} et 0. À partir de la paire (s, a), les politiques π_1 et π_2 ont le même rendement attendu mais des distributions de valeurs différentes. 44

5.1 Exemple de réseau de neurones avec une couche cachée. 45
5.2 Fonctions d'activations 46
5.3 Exemple de points critiques d'une fonction non convexe (indiqués en rouge). (a, c) Plateaux. (b, d) minima globaux. (e, g) maxima locaux. (f, h) Minimums locaux. 48
5.4 Exemple d'un RNN : medium.com 52
5.5 LSTM Cell 55
5.6 Child-sum tree LSTM au noeud j avec les enfants k1 et k2 56
5.7 N-ary Tree-LSTM 57

Chapitre 1

Introduction

Je me base principalement dans ce rapport sur les articles : [1],[2],[3] et [4] , les thèses : [5] et [6], les livres : [7] et [8], et le cours d'apprentissage statistique de l'université Paris Sud.

1.1 État de l'art

L'apprentissage par renforcement concerne un agent qui interagit avec l'environnement et apprend une politique optimale, par piste et par erreur, pour résoudre des problèmes de prise de décision séquentiels dans un large éventail de domaines. sciences naturelles et sociales et ingénierie (Sutton et Barto, 1998 ; 2017 ; Bertsekas et Tsitsiklis, 1996 ; Bertsekas, 2012 ; Szepesvari, 2010 ; Powell, 2011).

L'intégration de l'apprentissage par renforcement et des réseaux de neurones a une longue histoire (Sutton et Barto, 2017 ; Bertsekas et Tsitsiklis, 1996 ; Schmidhuber, 2015). Avec les récents acquis passionnants d'apprentissage en profondeur (LeCun et al., 2015 ; Goodfellow et al., 2016), les avantages tirés du Big Data, le calcul puissant, les nouvelles techniques algorithmiques, les progiciels et architectures matures et un solide soutien financier, nous avons été témoins la renaissance de l'apprentissage par renforcement (Krakovsky, 2016), en particulier la combinaison des réseaux de neurones profonds et de l'apprentissage par renforcement, c'est-à-dire l'apprentissage par renforcement en profondeur (Deep RL).

L'apprentissage en profondeur, ou réseaux de neurones profonds, a prévalu au cours des dernières années, dans les jeux, la robotique, le traitement du langage naturel, etc. Nous avons assisté à des avancées, comme le réseau Q profond (Mnih et al., 2015) et AlphaGo (Silver et al., 2016a) ; et de nouvelles architectures et applications, telles que l'ordinateur neuronal différenciable (Graves et al., 2016), les méthodes asynchrones (Mnih et al., 2016), les architectures de réseau en duel (Wang et al., 2016b), les réseaux d'itération de valeur (Tamar et al. , 2016), renforcement non supervisé et apprentissage auxiliaire (Jaderberg et al., 2017 ; Mirowski et al., 2017), conception d'architecture neuronale (Zoph et Le, 2017), double apprentissage pour la traduction automatique (He et al., 2016a), systèmes de dialogue parlé (Su et al., 2016b), extraction d'informations (Narasimhan et al., 2016), recherche de politiques guidée (Levine et al., 2016a) et apprentissage par imitation contradictoire génératif (Ho et Ermon, 2016), etc.

Pourquoi l'apprentissage en profondeur a-t-il aidé l'apprentissage par renforcement à réaliser des réalisations aussi nombreuses et aussi énormes ? L'apprentissage par représentation avec apprentissage en profondeur permet une ingénierie automatique des caractéristiques et un apprentissage de bout en bout via une descente de gradient, de sorte que la dépendance à la connaissance du domaine est considérablement réduite, voire supprimée. Auparavant, l'ingénierie des fonctionnalités était réalisée manuellement et prend généralement beaucoup de temps, est sur-spécifiée et incomplète. Les représentations profondes et distribuées exploitent la composition hiérarchique des facteurs dans les données pour lutter contre les défis exponentiels de la malédiction de la dimensionnalité. La généralité, l'expressivité et la souplesse des réseaux de neurones profonds rendent certaines tâches plus faciles ou possibles, par exemple dans les percées et les nouvelles architectures et applications décrites ci-dessus.

L'apprentissage en profondeur et l'apprentissage par renforcement, choisis parmi les technologies de pointe du MIT Technology Review 10 en 2013 et 2017 respectivement, joueront un rôle crucial dans la réalisation de l'intelligence générale artificielle. David Silver, le principal contributeur d'AlphaGo (Silver

et al., 2016a), a même mis au point une formule : intelligence artificielle = apprentissage par renforcement + apprentissage en profondeur (Silver, 2016).

La conception de bout en bout de systèmes de dialogue est récemment devenue un sujet de recherche populaire grâce à des outils puissants tels que des architectures décodeur-codeur pour l'apprentissage séquentiel. Cependant, la plupart des approches actuelles considèrent la gestion du dialogue homme-machine comme un problème d'apprentissage supervisé, visant à prédire la prochaine déclaration d'un participant compte tenu de l'historique complet du dialogue. Cette vision est trop simpliste pour rendre le problème de planification intrinsèque inhérent au dialogue ainsi que sa nature ancrée, rendant le contexte d'un dialogue plus vaste que la seule histoire. C'est pourquoi seul le bavardage et les tâches de réponse aux questions ont jusqu'à présent été traitées à l'aide d'architectures de bout en bout. Dans cet article, nous introduisons une méthode d'apprentissage en renforcement profond pour optimiser les dialogues orientés tâches orientés visuellement, basés sur l'algorithme de gradient de politique. Cette approche est testée sur un ensemble de données de 120 000 dialogues collectés via Mechanical Turk et fournit des résultats encourageants pour résoudre à la fois le problème de la génération de dialogues naturels et la tâche de découvrir un objet spécifique dans une image complexe.

Les systèmes de dialogue pratiques doivent mettre en oeuvre une stratégie de gestion qui définit le comportement du système, par exemple pour décider quand fournir des informations ou demander des éclaircissements à l'utilisateur. Bien que les approches traditionnelles utilisent des règles à motivation linguistique [Weizenbaum, 1966], les méthodes récentes reposent sur des données et utilisent l'apprentissage par renforcement (RL) [Lemon et Pietquin, 2007]. Des progrès significatifs dans le traitement du langage naturel via des réseaux neuronaux profonds [Bengio et al., 2003] ont ont fait des architectures codeurs-décodeurs neuronaux un moyen prometteur pour la formation d'agents conversationnels [Vinyals and Le, 2015 ; Sordoni et al., 2015 ; Serban et al., 2016].

Le principal avantage de ces systèmes de dialogue de bout en bout est qu'ils ne font aucune hypothèse sur le domaine d'application et qu'ils sont simplement formés de manière supervisée à partir de grands corpus de texte [Lowe et al., 2015].

Cependant, il existe de nombreux inconvénients à cette approche. 1) Premièrement, les modèles codeur-décodeur transforment le problème du dialogue en un apprentissage supervisé, en prédisant la répartition entre les énoncés suivants compte tenu du discours qui a été donné jusqu'à présent. Comme avec la traduction automatique, des dialogues incohérents et des erreurs susceptibles de s'accumuler avec le temps peuvent en résulter. Cela est d'autant plus vrai que l'espace d'action des systèmes de dialogue est vaste et que les jeux de données existants ne couvrent qu'un petit sous-ensemble de toutes les trajectoires, ce qui rend difficile la généralisation à des scénarios invisibles [Mooney, 2006]. 2) Deuxièmement, le cadre d'apprentissage supervisé ne prend pas en compte le problème intrinsèque de planification qui sous-tend le dialogue, c'est-à-dire le processus de prise de décision séquentiel, qui rend le dialogue cohérent dans le temps. Cela est particulièrement vrai lorsque vous vous engagez dans un dialogue axé sur les tâches. En conséquence, l'apprentissage par renforcement a été appliqué aux systèmes de dialogue depuis la fin des années 90 [Levin et al., 1997 ; Singh et al., 1999] et l'optimisation du dialogue a généralement été davantage étudiée que la génération de dialogue. 3) Troisièmement, il n'intègre pas naturellement les contextes externes (plus grands que l'historique du dialogue) qui sont le plus souvent utilisés par les participants au dialogue pour dialoguer. Ce contexte peut être leur environnement physique, une tâche commune qu'ils tentent d'accomplir, une carte sur laquelle ils essaient de trouver leur chemin, une base de données à laquelle ils veulent accéder, etc. Il fait partie de ce qu'on appelle le Common Ground, bien étudié dans la littérature [Clark et Schaefer, 1989]. Au cours des dernières décennies, le domaine de la psychologie cognitive a également apporté des preuves empiriques du fait que les représentations humaines sont fondées sur la perception et les systèmes moteurs [Barsalou, 2008]. Ces théories impliquent qu'un système de dialogue doit être fondé sur un environnement multimodal afin d'obtenir une compréhension du langage au niveau humain [Kiela et al., 2016]. Enfin, l'évaluation des dialogues est difficile car il n'existe pas de mesure d'évaluation automatique qui soit bien corrélée avec les évaluations humaines [Liu et al., 2016a]. D'autre part, les approches RL pourraient traiter les problèmes de planification et de métriques non différentiables, mais requièrent un apprentissage en ligne (bien que l'apprentissage par lots soit possible mais difficile avec de faibles quantités de données [Pietquin et al., 2011]). Pour cette raison, la simulation utilisateur a été proposée pour explorer les stratégies de dialogue dans un contexte RL [Eckert et al., 1997 ; Schatzmann et al., 2006 ; Pietquin et Hastie, 2013]. Cela nécessite également

la définition d'une mesure d'évaluation qui est le plus souvent liée à l'achèvement des tâches et à la satisfaction des utilisateurs [Walker et al., 1997]. En outre, les applications réussies du cadre RL au dialogue reposent souvent sur une structure prédéfinie de la tâche, telle que les tâches de remplissage de créneaux horaires [Williams et Young, 2007] dans lesquelles la tâche peut être remplie comme si elle remplissait un formulaire.

1.2 Problématique et plan de travail

Nous présentons une architecture globale pour l'optimisation de bout en bout du système de dialogue orienté et son application à une tâche multimodale, ancrant le dialogue dans un contexte visuel. Pour ce faire, nous partons d'un corpus de 150 000 dialogues humain-humain rassemblés via le récent jeu GuessWhat ? ! [de Vries et al., 2016]. Le but du jeu est de localiser un objet inconnu dans une image naturelle en posant une série de questions. Cette tâche est ardue car elle nécessite une compréhension de la scène et, plus important encore, une stratégie de dialogue permettant d'identifier rapidement l'objet. À partir de ces données, nous construisons d'abord un agent supervisé et un environnement d'entraînement neuronal. Il sert à former un agent DeepRL en ligne capable de résoudre le problème. Nous comparons ensuite quantitativement et qualitativement les performances de notre système avec une approche supervisée sur la même tâche du point de vue humain.

Le plan de ce rapport est le suivant : nous commençons d'abords par une présentation de l'état de l'art de notre problématique, ensuite nous détaillons les règles du jeu GuessWhat ? ! et les notions liées à notre environnement d'apprentissage dans le chapitre 2. Ensuite,nous présentons une introduction à la théorie de l'apprentissage automatique (ML) dans le chapitre 3, ensuite nous nous concentrons plus sur la théorie de l'apprentissage par renforcement dans le chapitre 4. Puis, nous détaillons les techniques utlisées dans l'apprentissage profond dans le chapitre 5 , ainsi que ses aspects mathématiques , en particulier nous nous concentrons sur LSTM et nous donnons dans les annexes les implémentations de tous les algorithmes utilisés dans ce projet .

Chapitre 2

GuessWhat ? !

Nous expliquons brièvement ici le jeu GuessWhat ? ! cela servira de tâche à notre système de dialogue, mais reportez-vous à [de Vries et al., 2016] pour plus de détails sur la tâche et le contenu exact de l'ensemble de données. Il est composé de plus de 150 000 dialogues humains-humains en langage naturel, rassemblés dans Mechanical Turk.

2.1 Règles de jeu

C'est un jeu coopératif à deux joueurs dans lequel les deux joueurs voient l'image d'une scène visuelle riche avec plusieurs objets. Un joueur - l'oracle - se voit attribuer de manière aléatoire un objet (qui pourrait être une personne) dans la scène. Cet objet n'est pas connu de l'autre joueur - l'interrogateur - dont le but est de localisez l'objet caché. Pour ce faire, le questionneur peut poser une série de questions oui-non auxquelles l'oracle répond, comme le montre la figure. Notez que le questionneur n'a pas connaissance de la liste d'objets et ne peut voir que l'image complète. Une fois que le questionneur a rassemblé suffisamment de preuves pour localiser l'objet, il peut choisir de deviner l'objet. La liste des objets est révélée et si le questionneur sélectionne le bon objet, le jeu est considéré comme réussi.

2.2 Notations

Le jeu est défini par un tuple $(\mathcal{I}, D, O, o^*)$ avec $\mathcal{I} \in \mathbb{R}^{H \times W}$ une image de hauteur H et de largeur W,D un dialogue avec J questions-réponses couples : $D = \left(\mathbf{q}_j, a_j\right)_{j=1}^{J}$. O une liste de K objets : $O = (o_k)_{k=1}^{K}$ et o^* l'objet cible.De plus, chaque question $\mathbf{q}_j = \left(w_i^j\right)_{i=1}^{I_j}$ est une séquence de longueur I_j avec chaque token w_i^j tiré d'un vocabulaire prédéfini V. Le vocabulaire V est composé d'une liste de mots prédéfinie, d'une étiquette de question $< ? >$ qui termine une question et un token d'arrêt $< stop >$ qui met fin à un dialogue. Une réponse est restreinte pour être oui, non ou non applicable (N.A) i.e $a_j \in \{< yes >, < no >, < na >\}$. Pour chaque objet k, une catégorie d'objet $c_k \in \{1, \ldots, C\}$ et un masque de segmentation en pixels $S_k \in \{0, 1\}^{H \times W}$ sont disponibles.Enfin, pour accéder aux sous-ensembles d'une liste, nous utilisons les notations suivantes. Si $l = \left(l_i^j\right)_{i=1}^{I_j, j}$ est une liste à double indice,alors $l_{1:i}^j = \left(l_p^j\right)_{p=1}^{i,j}$ sont les i premiers éléments de la j^{eme} liste si $1 \leqslant i \leqslant I_j$, sinon $l_{1:p}^j = \emptyset$. Ainsi, par exemple, $w_{1:i}^j$ fait référence aux premiers i tokens de la j^{eme} question et $(\mathbf{q}, a)_{1:j}$ réfère aux j premières paires question / réponse d'un dialogue.

2.3 Image captioning

Le sous-titrage des images s'appuie sur la base de données MS COCO constituée de 120 000 images avec plus de 800k de segmentations d'objets. En outre, le jeu de données fournit 5 sous-titres par image, ce qui a déclenché une explosion d'intérêt de la part des chercheurs pour la génération d'images en langage naturel. Plusieurs méthodes ont été proposées [ref], toutes inspirées de l'approche codeur-décodeur [ref] qui s'est avérée efficace pour la traduction automatique. La recherche sur le sous-titrage des images a

permis de découvrir des méthodes efficaces permettant de générer automatiquement des déclarations factuelles cohérentes sur les images. Modéliser les interactions dans GuessWhat ?! nécessite plutôt de modéliser le processus de poser des questions utiles sur les images.

2.4 Visual Question Answering

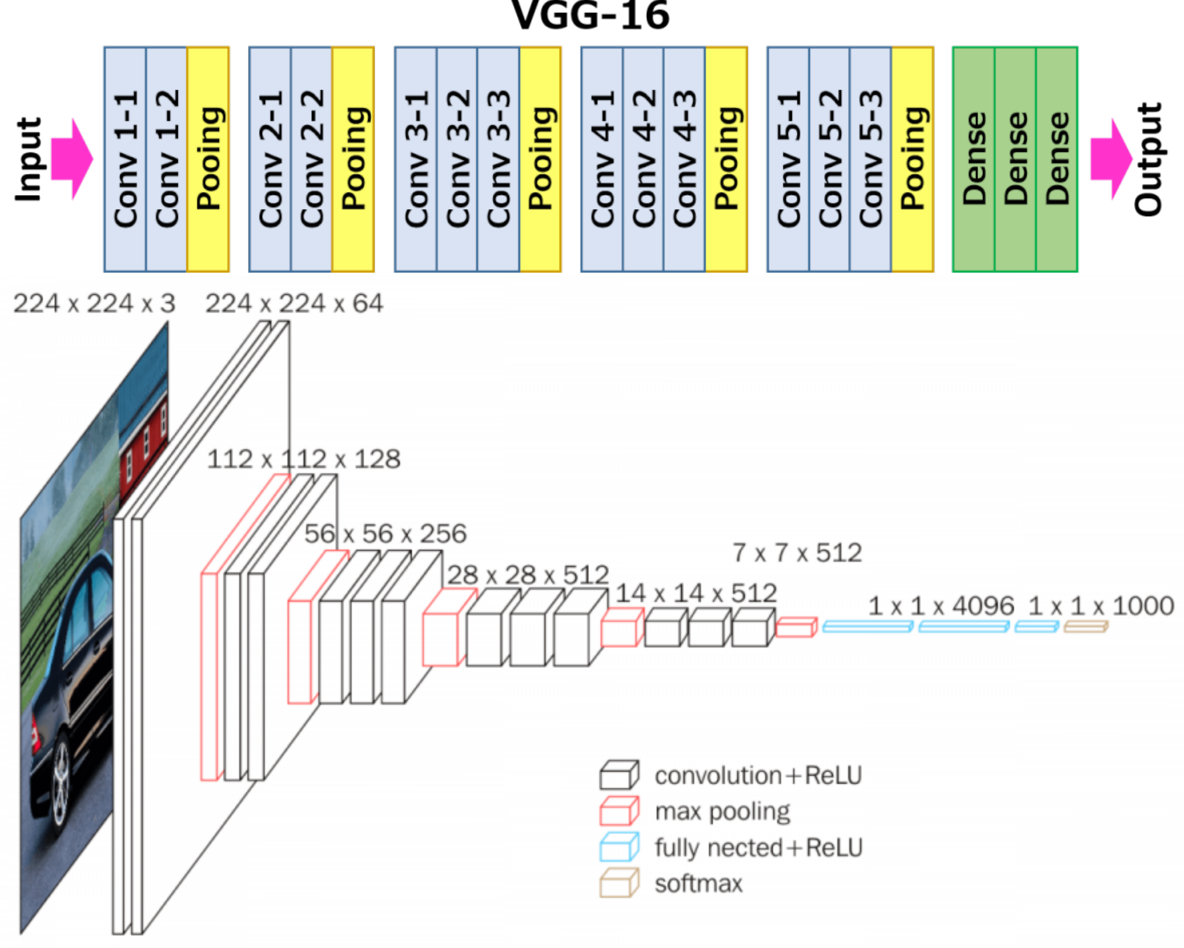
Les tâches de VQA : Visual Question Answering constituent une autre extension bien connue de la tâche de sous-titrage. Ils doivent plutôt répondre à une question à partir d'une image (par exemple, «Combien de zèbres y a-t-il sur l'image ?», «Est-ce qu'il pleut dehors ?»). Récemment, le défi VQA a fourni un nouvel ensemble de données bien plus volumineux que les tentatives précédentes où, tout comme dans GuessWhat ?!, Les questions sont de forme libre. De nombreux travaux ont découlé de cette publication, s'appuyant en grande partie sur la littérature relative au sous-titrage d'images . Malheureusement, beaucoup de ces méthodes avancées ont montré une amélioration marginale sur des lignes de base simples . Des travaux récents indiquent également que les modèles formés signalent souvent la même réponse à une question, quelle que soit l'image, ce qui suggère qu'ils exploitent largement les corrélations prédictives entre les questions et les réponses présentes dans l'ensemble de données. Le jeu GuessWhat ?! tente de contourner ces problèmes. En raison de l'objectif de l'interrogateur de localiser l'objet caché, les questions générées sont de nature différente : elles favorisent naturellement la compréhension spatiale de la scène et des attributs des objets qu'elle contient, ce qui rend plus utile la consultation de l'image. En outre, il ne contient que des questions binaires dont nous trouvons les réponses équilibrées et a deux fois plus de questions en moyenne par image.

2.5 Dialogue dirigé par objectif

Devine quoi ?! est également pertinent pour la communauté de recherche sur les dialogues dirigés. De tels systèmes visent à atteindre un objectif en collaboration avec un utilisateur, tel que la récupération d'informations ou la résolution d'un problème. Bien que les systèmes de dialogue axés sur les objectifs soient attrayants, ils restent difficiles à concevoir. Ainsi, ils sont généralement limités à des domaines spécifiques tels que la vente de billets de train, les informations touristiques ou l'acheminement des appels [32, 40, 47]. En outre, les jeux de données de dialogue existants sont limités à moins de 100 000 exemples de dialogues [12], à moins qu'ils ne soient générés avec des formats de gabarit [12, 43, 44] ou de simulation [33, 36], auquel cas ils ne reflètent pas le caractère libre. forme de conversations naturelles. Enfin, les travaux récents sur les systèmes de dialogue de bout en bout ne parviennent pas à gérer les contextes dynamiques. Par exemple, [43] croise un dialogue avec une base de données externe pour recommander des restaurants. Les systèmes de dialogue bien connus basés sur les jeux [1, 2] reposent également sur des bases de données statiques. En revanche, devinez quoi ?! les dialogues sont fortement ancrés par les images. Le dialogue qui en résulte est hautement contextuel et doit être basé sur le contenu de l'image actuelle plutôt que sur une base de données externe. Ainsi, à notre connaissance, le GuessWhat ?! Le jeu de données marque une étape importante pour la recherche sur le dialogue, car il s'agit du premier jeu de données à grande échelle qui est à la fois orienté objectif et multimodal.

2.6 Quelques notions et définitions

Word Embeddings Word Embeddings (L'incorporation de mots) est une représentation d'un mot dans un espace vectoriel où des mots sémantiquement similaires sont mappés sur des points proches. Les mots incorporés peuvent être formés et utilisés pour dériver des similitudes entre les mots. Ils sont un arrangement de nombres représentant les informations sémantiques et syntaxiques des mots dans un format compréhensible par les ordinateurs. Pendant de nombreuses années, les systèmes et les techniques de la PNL représenteraient le sens des mots en utilisant WordNet (George A. Miller, Université de Princeton, 1985), qui est fondamentalement un très grand graphique qui définit différentes relations entre les mots. En termes d'espace vectoriel, chaque mot est un vecteur avec un 1 et beaucoup de zéros (taille de vocabulaire -1). C'est ce qu'on appelle un one-hot qui décrit les mots de la manière la plus simple. Cependant, cette représentation discrète posait de nombreux problèmes, tels que des nuances



L'architecture de VGG16

manquantes, des mots nouveaux manquants, la nécessité de créer et d'adapter le travail humain, il était difficile de calculer la similarité des mots avec précision et, surtout, lorsque le vocabulaire est vaste, la représentation vectorielle est gigantesque.

VGG16 : C'est un modèle de réseau neuronal convolutionnel proposé par K. Simonyan et A. Zisserman de l'Université d'Oxford dans l'article intitulé «Very Deep Convolutional Networks for Large-Scale Image Recognition». Le modèle atteint une précision de 92,7% dans le top 5 des tests dans ImageNet, qui est un jeu de données de plus de 14 millions d'images appartenant à 1 000 classes. C'était l'un des fameux modèle soumis à ILSVRC-2014. Il apporte des améliorations par rapport à AlexNet en remplaçant les grands filtres de la taille du noyau (respectivement 11 et 5 dans la première et la deuxième couche de convolution) par plusieurs filtres de la taille du noyau 3×3, l'un après l'autre. VGG16 a été entrainé pendant des semaines et utilisait NVIDIA GPU Titan noir.[https ://neurohive.io/en/popular-networks/vgg16/]

Problème de décodage : Les RNN sont formés pour estimer la probabilité de séquences de tokens à partir du dictionnaire $\mathcal{V}$ afin de prendre une entrée x. Le RNN met à jour son état interne et estime la distribution de la probabilité conditionnelle pour la sortie suivante en fonction de l'entrée et de tous les tokens de sortie précédents. On note le logarithme de cette distribution de probabilité conditionnelle sur tous les tokens (jetons) en instant t par

$$\theta(y_t) = \log \Pr(y_t | y_{t-1}, \ldots, y_1, \mathbf{x}) \tag{2.1}$$

Algorithm : Diverse Beam Search

```
1 Perform a diverse beam search with G groups using a beam width of B
2 for t = 1, ... T do
      // perform one step of beam search for first group without diversity
3     Y^1_[t] ← argmax_(y^1_1,[t],...,y^1_B',[t]) Σ_b∈[B'] Θ(y^1_b,[t])
4     for g = 2, ... G do
          // augment log-probabilities with diversity penalty
5         Θ(y^g_b,[t]) ← Θ(y^g_b,[t]) + Σ_h λ_g Δ(y^g_b,[t], Y^h_[t])    b ∈ [B'], y^g_b,[t] ∈ 𝒴^g and λ_g > 0
          // perform one step of beam search for the group
6         Y^g_[t] ← argmax_(y^g_1,[t],...,y^g_B',[t]) Σ_b∈[B'] Θ(y^g_b,[t])
7 Return set of B solutions, Y_[T] = ∪_{g=1}^G Y^g_[T]
```

Algorithme : Diverse Beam Search

Pour simplifier la notation, on indexe $\theta(\cdot)$ avec une seule variable y_t,mais il devrait être clair que cela dépend des sorties précédentes $\mathbf{y}_{[t-1]}$.Le log-probabilité d'une solution partielle (c'est-à-dire la somme des log-probabilités de tous les jetons précédents décodés) peut maintenant être écrit comme $\Theta\left(\mathbf{y}_{[t]}\right) = \sum_{\tau\in[t]} \theta\left(y_\tau\right)$. Le problème du décodage est alors la tâche de trouver une séquence qui maximise $\Theta(\mathbf{y})$. Comme chaque sortie est conditionnée par toutes les sorties précédentes, le décodage de la séquence optimale de longueur T dans cette configuration peut être considéré comme une inférence MAP sur la chaîne de Markov d'ordre T, les T nœuds correspondant aux jetons de sortie. Non seulement la taille du facteur le plus important dans un tel graphique augmente-t-elle en $|\mathcal{V}|^T$,mais nécessite également une transmission inutile du RNN à plusieurs reprises pour calculer les entrées dans les facteurs. Ainsi, des algorithmes approximatifs sont utilisés.

Beam-search : Les architectures de génération de séquences basées sur RNN modélisent la probabilité conditionnelle $\Pr(\mathbf{y}|\mathbf{x})$ d'une séquence de sortie $\mathbf{y} = (y_1, \ldots, y_T)$ étant donné une entrée $\mathbf{x}$(éventuellement aussi une séquence) ; où les jetons (tokens) de sortie y_t sont d'un vocabulaire fini $\mathcal{V}$. L'inférence maximale a posteriori (MAP) pour les RNN est la tâche qui consiste à trouver la séquence de sortie la plus probable compte tenu de l'entrée. Comme le nombre de séquences possibles augmente avec $|\mathcal{V}|^T$, l'inférence exacte étant NP-difficile, des algorithmes d'inférence approximatifs tels que Beam Search (BS) sont couramment utilisés. Cette heuristique vise à trouver la séquence de mots la plus probable en explorant un sous-ensemble de toutes les questions et en conservant les séquences de B-candidats les plus prometteuses à chaque pas de temps où B est connu comme la largeur du faisceau (beam width) . Notons l'ensemble des solutions B détenues par BS au début du temps t comme $Y_{[t-1]} = \left\{\mathbf{y}_{1,[t-1]}, \ldots, \mathbf{y}_{B,[t-1]}\right\}$. A chaque pas de temps, BS considère toutes les extensions possibles de jetons uniques de ces faisceaux données par l'ensemble $\mathcal{Y}_t = Y_{[t-1]} \times \mathcal{V}$ et sélectionne les B-extensions les plus probables. Plus formellement, à chaque étape,

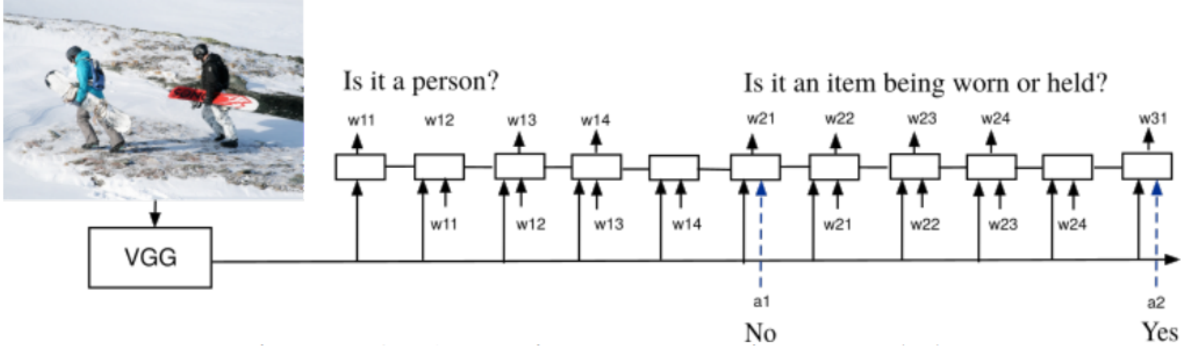
$$Y_{[t]} = \underset{\mathbf{y}_{1,[t]},\ldots,\mathbf{y}_{B,[t]}\in\mathcal{Y}_t}{\operatorname{argmax}} \sum_{b\in[B]} \Theta\left(\mathbf{y}_{b,[t]}\right) \quad \text{si } \mathbf{y}_{i,[t]} \neq \mathbf{y}_{j,[t]} \tag{2.2}$$

Il existe une autre version plus optimale de cet algorithme qui se nomme : Diverse Beam Search (voir [9] pour plus de détails), dont le pseudo-code [2.2].

Vous trouverez son implémentation par [9] sur `https://github.com/ashwinkalyan/dbs`.

2.7 Environnement d'apprentisssage

Nous construisons un environnement d'apprentisssage qui permet au RL d'optimiser la tâche de l'interrogateur en créant des modèles pour les tâches Oracle et de devineur.



Modèle de génération de questions

2.7.1 Génerations de questions

Nous divisons le travail du questionneur en deux tâches différentes : l'une pour poser les questions et l'autre pour deviner l'objet. La tâche de génération de question nécessite de générer une nouvelle question q_{j+1} sachant l'image $\mathcal{I}$ et l'historique de j questions/réponses $(q, a)_{1:j}$. Nous modélisons le générateur de questions (QGen) avec un réseau de neurones récurrents (RNN), qui produit une séquence de vecteurs d'état RNN $\mathbf{s}^j_{1:i}$ pour une séquence d'entrée donnée $\boldsymbol{w}^j_{1:i}$ en appliquant la fonction de transition f : $s^j_{i+1} = f\left(s^j_i, w^j_i\right)$.Nous utilisons la populaire cellule de mémoire à long terme (LSTM) [Hochreiter et Schmidhuber, 1997] comme fonction de transition. [Child sum ? ?]. Afin de construire un modèle de séquence probabiliste, on peut ajouter une fonction softmax $\sigma : \mathbb{R}^K \to \mathbb{R}^K$ qui calcule une distribution sur les tokens w^j_i à partir du vocabulaire V. σ est définie par :

$$\sigma(\mathbf{z})_i := \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.3}$$

pour $i = 1, \ldots, K$ et $\mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$

Dans notre cas, cette distribution de sortie dépend de tous les tokens de questions et réponses précédents, ainsi que de l'image $\mathcal{I}$:

$$p\left(w^j_i | w^j_{1:i-1}, (\mathbf{q}, a)_{1:j-1}, \mathcal{I}\right) \tag{2.4}$$

Nous conditionnons le modèle à l'image en obtenant ses caractéristiques VGG16 FC8 et en le concaténant avec l'imbrication d'entrée (input embedding) à chaque étape, comme illustré sur la Figure 2.3. On précise quelques termes utilisés :

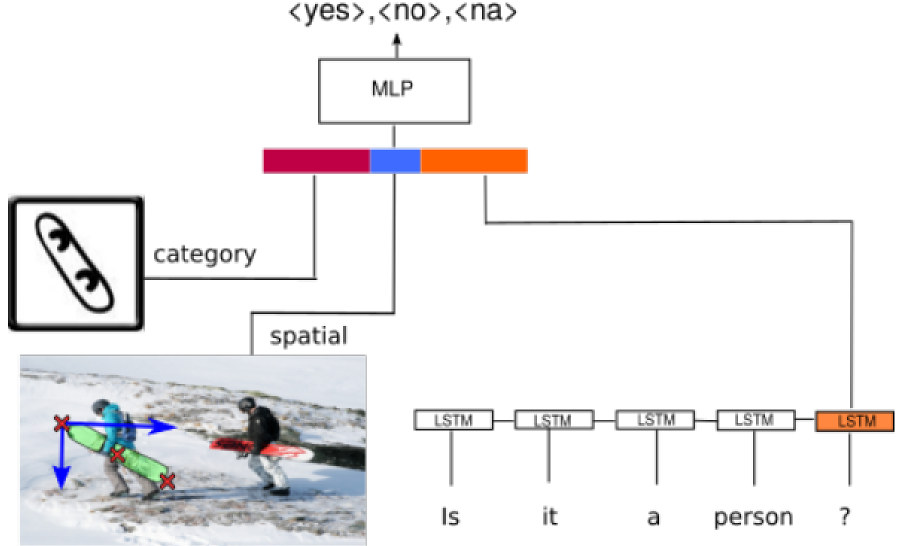
Nous formons le modèle en minimisant la log-vraisemblance négative conditionnelle :

$$\begin{aligned} -\log p\left(\mathbf{q}_{1:J} | a_{1:J}, \mathcal{I}\right) &= -\log \prod_{j=1}^{J} p\left(\mathbf{q}_j | (\mathbf{q}, a)_{1:j-1}, \mathcal{I}\right) \\ &= -\sum_{j=1}^{J} \sum_{i=1}^{I_j} \log p\left(w^j_i | w^j_{1:i-1}, (\mathbf{q}, a)_{1:j-1}, \mathcal{I}\right) \end{aligned} \tag{2.5}$$

Au moment du test, nous pouvons générer un échantillon $p\left(\mathbf{q}_j | (\mathbf{q}, a)_{1:j-1}, \mathcal{I}\right)$ à partir du modèle comme suit. À partir de l'état s^j_1, nous échantillonnons un nouveau token w^j_i à partir de la distribution σ de sortie et alimentez le token intégré $e\left(w^j_i\right)$ retourné comme entrée au RNN. Nous répétons cette boucle jusqu'à rencontrer un token de fin de séquence. Pour trouver approximativement la question la plus probable,$\max_{\mathbf{q}_j} p\left(\mathbf{q}_j | (\mathbf{q}, a)_{1:j-1}, \mathcal{I}\right)$ nous utilisons la procédure Beam-Search (2.6) couramment utilisée.

2.7.2 Oracle

La tâche oracle nécessite de produire une réponse oui-non pour tout objet dans une image à partir d'une question en langage naturel. Nous décrivons ici l'architecture de réseau de neurones qui a réalisé la meilleure performance et nous nous référons à [de Vries et al., 2016] pour une étude approfondie de l'impact d'autres informations d'objet et d'image. Tout d'abord, nous intégrons les informations spatiales de la culture en extrayant un vecteur à 8 dimensions de l'emplacement du cadre de sélection



Modèle de l'Oracle

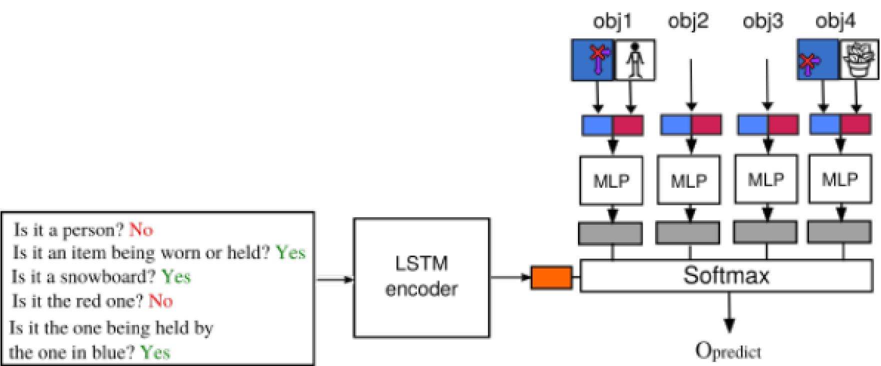
$[x_{min}, y_{min}, x_{max}, y_{max}, x_{center}, y_{center}, w_{box}, h_{box}]$ où w_{box} et h_{box} désignent respectivement la largeur et la hauteur du cadre de sélection. Nous normalisons la hauteur et la largeur de l'image de telle sorte que les coordonnées vont de -1 à 1, et plaçons l'origine au centre de l'image. Deuxièmement, nous convertissons la catégorie d'objets c^* en une catégorie dense incorporant une table de consultation apprise. Enfin, nous utilisons un LSTM pour coder la question actuelle q. Nous concaténons ensuite les trois imbrications en un seul vecteur et le transmettons en entrée à une seule couche MLP masquée qui produit la distribution de la réponse finale $p\left(a|q, c^*, x^*_{spatial}\right)$ en utilisant une couche softmax, illustrée à la Fig2.4.

2.7.3 Devineur

Le modèle de devineur prend une image $\mathcal{I}$ et une séquence de questions et réponses $(q, a)_{1:N}$, et prédit le bon objet o^* de l'ensemble de tous les objets. Ce modèle considère un dialogue comme une séquence plate de tokens question-réponse et utilise le dernier état caché du codeur LSTM en tant que représentation de dialogue. Nous effectuons un produit scalaire entre cette représentation et l'intégration de tous les objets de l'image, suivi d'un softmax pour obtenir une distribution de prédiction sur les objets. Les embeddings d'objets sont obtenues à partir des caractéristiques catégorielles et spatiales. Plus précisément, nous concaténons la représentation spatiale en 8 dimensions et la recherche de catégorie d'objet et la transmettons à travers une couche MLP pour obtenir un emmbeding de l'objet. Notez que les paramètres MLP sont partagés pour gérer le nombre variable d'objets dans l'image. Voir la figure 2.5 pour un aperçu du devineur.

2.8 Génération de jeux complets

Avec les modèles de génération de questions, d'oracle et de devineur, nous avons tous les composants pour simuler un jeu complet. Étant donné une image initiale I, nous générons une question q_1 en échantillonnant des jetons (tokens) à partir du modèle de génération de questions jusqu'à atteindre le jeton de point d'interrogation. Alternativement, nous pouvons remplacer la procédure d'échantillonnage par une recherche de faisceau (beam search) pour trouver approximativement la question la plus probable en fonction du générateur. L'oracle prend alors la question q_1, la catégorie d'objet c^* et $x^*_{spatial}$ en tant qu'entrées, et génère la réponse a_1. Nous annexons (q_1, a_1) au dialogue et répétons la génération de paires question-réponse jusqu'à ce que le générateur émette un jeton d'arrêt du dialogue ou que le



Modèle du Devineur

nombre maximal de questions-réponses soit atteint. Enfin, le modèle de devineur prend le dialogue généré D et la liste des objets O et prédit le bon objet.

Pour comprendre les notions de la partie suivante plus en détail, voir le chapitre 4 .

2.9 GuessWhat ?! du point de vue de RL

L'un des inconvénients de la formation du QGen dans une configuration d'apprentissage supervisé est que sa séquence de questions n'est pas explicitement optimisée pour trouver le bon objet. De tels objectifs de formation passent à côté de la planification sous-jacente aux dialogues (axés sur les objectifs). Dans cet article, nous proposons de transformer la tâche de génération de question en tâche RL. Plus spécifiquement, nous utilisons l'environnement de formation décrit précédemment et considérons l'oracle et le devineur dans l'environnement de l'agent RL. Dans ce qui suit, nous formalisons d'abord les tâches GuessWhat ?! en tant que processus de décision de Markov (MDP) afin d'appliquer un algorithme de gradient de politique (policy gradient) au problème QGen.

2.9.1 GuessWhat ?! en tant que processus de décision de Markov

Nous définissons l'état $\mathbf{x_t}$ en tant que statut du jeu à l'étape t. Plus précisément, nous définissons $\mathbf{x_t} = \left(\left(w^j_1, \dots, w^j_i\right), (\mathbf{q}, a)_{1:j-1}, \mathcal{I}\right)$ où $t = \sum_{j=1}^{j-1} I_j + i$ correspond au nombre de jetons générés depuis le début du dialogue. Une action u_t correspond à la sélection d'un nouveau mot w^j_{i+1} dans le vocabulaire V. Le passage à l'état suivant dépend de l'action sélectionnée :

- Si $w^j_{i+1} =< \text{stop} >$, le dialogue complet est terminé.
- Si $w^j_{i+1} =< ? >$, la question en cours est terminée et une réponse a_j est échantillonnée à partir de l'oracle. Le prochain état est $\mathbf{x}_{t+1} = ((\mathbf{q}, a)_{1:j}, \mathcal{I})$ où $\mathbf{q}_j = \left(w^j_1, \dots, w^j_i, < ? >\right)$.
- Sinon le nouveau mot est ajouté à la question en cours et $\mathbf{x}_{t+1} = \left(\left(w^f_1, \dots, w^j_i, w^J_{i+1}\right), (\mathbf{q}, a)_{1:j-1}, \mathcal{I}\right)$.

Les questions se terminent automatiquement après I_{max} mots. De même, les dialogues se terminent après J_{max} questions. De plus, une récompense $r(\mathbf{x}, u)$ est définie pour chaque couple de réactions. Une trajectoire $\tau = (\mathbf{x}_t, u_t, \mathbf{x}_{t+1}, r(\mathbf{x}_t, u_t))_{1:T}$ est une séquence finie de tuples de longueur T qui contient un état, une action, l'état suivant et la récompense où $T \leqslant J_{max} * I_{max}$. Ainsi, le jeu tombe dans un RL scénario épisodique de comme le dialogue se termine après une séquence finie de paires question-réponse. Enfin, la sortie de QGen peut être vue comme une politique stochastique $\pi_{\theta}(u|\mathbf{x})$ paramétrée par θ qui associe une distribution de probabilité sur les actions (c'est-à-dire des mots) pour chaque état (c'est-à-dire un dialogue et une image intermédiaires).

2.9.2 Entraînement de QGen avec Policy Gradient

Bien que plusieurs approches existent dans la littérature RL, nous optons pour les méthodes de gradient de politique car elles sont connues pour s'adapter à de grands espaces d'action. Ceci est particulièrement important dans notre cas car la taille du vocabulaire est d'environ 5 000 mots. L'optimisation de la politique a pour objectif de trouver une politique $\pi_\theta(u|x)$ qui maximise le rendement attendu, également appelée valeur moyenne :

$$J(\theta) = E_{\pi_\theta}\left[\sum_{t=1}^{T} \gamma^{t-1} r(x_t, u_t)\right] \tag{2.6}$$

où $\gamma \in [0, 1]$ est le facteur d'actualisation (discount factor), T la longueur de la trajectoire et l'état de départ x_1 est tiré d'une distribution p1. Notez que $\gamma = 1$ est autorisé car nous sommes dans le scénario épisodique [Sutton et al., 1999]. Pour améliorer la politique, ses paramètres peuvent être mis à jour dans la direction du gradient de la valeur moyenne :

$$\theta_{h+1} = \theta_h + \alpha_h \nabla_\theta J|_{\theta=\theta_h} \tag{2.7}$$

où h désigne le pas de temps de formation et α_h est un taux d'apprentissage tel que $\sum_{h=1}^{\infty} \alpha_h = \infty$ et $\sum_{h=1}^{\infty} \alpha_h^2 < \infty$.

Grâce au théorème de la politique de gradient [Sutton et al., 1999], le gradient de la valeur moyenne peut être estimé à partir d'un ensemble de trajectoires $\mathcal{T}_h$ échantillonnées à partir de la politique actuelle π_{θ_h} par :

$$\nabla J(\theta_h) = \left\langle \sum_{t=1}^{T} \sum_{u_t \in V} \nabla_{\theta_h} \log \pi_{\theta_h}(u_t|x_t) \left(Q^{\pi_{\theta_h}}(x_t, u_t) - b\right) \right\rangle_{\mathcal{T}_h} \tag{2.8}$$

où $Q^{\pi_{\theta_h}}(x, u)$ est la fonction de valeur action d'état qui estime la récompense attendue cumulative pour un couple état-action donné et b une fonction de base arbitraire qui peut aider à réduire la variance de l'estimation du gradient. Plus précisément

$$Q^{\pi_{\theta_h}}(x_t, u_t) = E_{\pi_\theta}\left[\sum_{t'=t}^{T} \gamma^{t'-t} r(x_{t'}, u_{t'})\right] \tag{2.9}$$

Notez que l'estimation dans Eq (1) n'est valable que si la distribution de probabilité de l'état initial x_1 est uniformément distribuée. La fonction de valeur d'état-action $Q^{\pi_{\theta_h}}(x, u)$ peut ensuite être estimée soit en apprenant un approximateur de fonction (méthodes acteur-critique), soit par des déploiements de Monte-Carlo (REINFORCE [Williams, 1992]). Dans REINFORCE, la somme interne des actions est estimée en utilisant les actions de la trajectoire. Par conséquent, l'équation (1) peut être simplifiée pour :

$$\nabla J(\theta_h) = \left\langle \sum_{t=1}^{T} \nabla_{\theta_h} \log \pi_{\theta_h}(u_t|x_t) \left(Q^{\pi_{\theta_h}}(x_t, u_t) - b\right) \right\rangle_{\mathcal{T}_h} \tag{2.10}$$

Enfin, en utilisant le GuessWhat ?! notation de jeu pour Eq (2), le Policy Gradient du QGen peut s'écrire comme suit :

$$\nabla J(\theta_h) = \left\langle \sum_{j=1}^{J} \sum_{i=1}^{I_j} \nabla_{\theta_h} \log \pi_{\theta_h}\left(w_i^j | w_{1:i-1}^j, (q, a)_{1:j-1}, \mathcal{I}\right) \left(Q^{\pi_{\theta_h}}\left(\left(w_{1:i-1}^j, (q, a)_{1:j-1}, \mathcal{I}\right), w_i^j\right) - b\right) \right\rangle_{\mathcal{T}_h} \tag{2.11}$$

2.9.3 Fonction de récompense

Un aspect fastidieux de RL est de définir une fonction de récompense correcte et valable. La politique optimale étant le résultat de la fonction de récompense, il convient de concevoir avec soin une récompense qui ne modifierait pas la politique optimale finale attendue [Ng et al., 1999]. Par conséquent, nous mettons

Algorithm Training of QGen with REINFORCE

Require: Pretrained QGen,Oracle and Guesser
Require: Batch size K
1: **for** Each update **do**
2: # Generate trajectories $\mathcal{T}_h$
3: **for** $k = 1$ to K **do**
4: Pick Image $\mathcal{I}_k$ and the target object $o_k^* \in O_k$
5: # Generate question-answer pairs $(\boldsymbol{q}, a)_{1:j}^k$
6: **for** $j = 1$ to J_{max} **do**
7: $q_j^k = QGen(\boldsymbol{q}, a)_{1:j-1}^k, \mathcal{I}_k)$
8: $a_j^k = Oracle(\boldsymbol{q}_j^k, o_k^*, \mathcal{I}_k)$
9: **if** <*stop*> $\in \boldsymbol{q}_j^k$ **then**
10: delete $(q, a)_j^k$ and break;
11: $p(o_k|\cdot) = Guesser((q, a)_{1:j}^k, \mathcal{I}_k, O_k)$
12: $r(\boldsymbol{x}_t, u_t) = \begin{cases} 1 & \text{If argmax}_{o_k} p(o_k|\cdot) = o_k^* \\ 0 & \text{Otherwise} \end{cases}$
13: Define $\mathcal{T}_h = ((q, a)_{1:j_k}^k, \mathcal{I}_k, r_k)_{1:K}$
14: Evaluate $\nabla J(\boldsymbol{\theta}_h)$ with Eq. (3) with $\mathcal{T}_h$
15: SGD update of QGen parameters $\boldsymbol{\theta}$ using $\nabla J(\boldsymbol{\theta}_h)$
16: Evaluate $\nabla L(\boldsymbol{\phi}_h)$ with Eq. (4) with $\mathcal{T}_h$
17: SGD update of baseline parameters using $\nabla L(\boldsymbol{\phi}_h)$

Entraînement de QGen avec REINFORCE [1]

un minimum de connaissances préalables dans la fonction de récompense et construisons une récompense 0-1 en fonction de la prédiction du devineur :

$$r(x_t, u_t) = \begin{cases} 1 & \text{Si argmax }_o [\text{Devineur}(x_t)] = o^* \text{ et } t = T \\ 0 & \text{Sinon} \end{cases} \tag{2.12}$$

Donc, nous donnons une récompense de un si l'objet correct est trouvé dans les questions générées, et de zéro sinon.

Notez que la fonction de récompense requiert l'objet cible o^* alors qu'il n'est pas inclus dans l'état $x = ((q, a)_{1:J}, \mathcal{I})$. Cela brise l'hypothèse du PDM selon laquelle la récompense devrait être fonction de l'état et de l'action actuels. Cependant, les méthodes à gradient de politique, telles que REINFORCE, restent applicables si le PDM est partiellement observable [Williams, 1992].

2.9.4 Procédure d'entraînement complète

Pour QGen, l'oracle et le devineur, nous utilisons les architectures de modèle décrites dans 2.7 . Nous formons d'abord indépendamment les trois modèles avec une perte d'entropie croisée (cross-entropy loss). Nous maintenons ensuite les modèles oracle et devineur fixes, tandis que nous formons le QGen dans le cadre RL décrit. Il est important de pré-former le QGen pour qu'il entame la formation à partir d'une politique raisonnable. La taille de l'espace d'action est tout simplement trop grande pour partir d'une politique aléatoire.

Afin de réduire la variance du gradient de politique, nous implémentons la ligne de base $b_\phi(x_t)$ en fonction de l'état actuel, paramétré par ϕ. Plus précisément, nous utilisons un MLP à une couche qui prend l'état caché LSTM du QGen et prédit la récompense attendue. Nous formons la fonction de base en minimisant l'erreur quadratique moyenne (MSE) entre la récompense prévue et la récompense actualisée de la trajectoire au pas de temps actuel :

$$L(\phi_h) = \left\langle \left[b_{\phi_h}(x_t) - \sum_{t'=t}^{T} \gamma^{t'} r_{t'} \right]^2 \right\rangle_{\mathcal{T}_h} \tag{2.13}$$

		New Objects	New Pictures
Baseline	Sampling	46.4% ± 0.2	45.0% ± 0.1
	Greedy	48.2% ± 0.1	46.9%
	BSearch	53.4% ± 0.0	53.0%
REINFORCE	Sampling	**63.2% ± 0.3**	**62.0% ± 0.2**
	Greedy	58.6% ± 0.0	57.5%
	BSearch	54.3% ± 0.1	53.2%

Précisions des modèles de la performance humaine du QGen formé avec baseline et REINFORCE. Les nouveaux objets font référence à l'échantillonnage uniforme des objets dans l'ensemble d'apprentissage, tandis que les nouvelles images font référence à l'ensemble de test.

Nous résumons notre procédure d'entraînement dans l'algorithme 2.6.

2.10 Expériences

Comme déjà dit, nous avons utilisé l'ensemble de données GuessWhat ? ! comprenant 155 281 dialogues contenant 821 955 paires question / réponse composées de 4900 mots de 66 537 images uniques et de 134 074 objets uniques. Le code source des expériences est disponible sur `https://guesswhat.ai`.

2.10.1 Détails de l'entraînement

Nous pré-entraînons les réseaux décrits dans la section 2.7. Après l'entraînement, le réseau oracle obtient une erreur de 21,5% et le réseau de devineur rapporte une erreur de 36,2% sur l'ensemble de tests. Dans le reste de cette section, nous nous référons au QGen pré-entraîné comme notre modèle de base. Nous initialisons ensuite notre environnement avec les modèles pré-formés et formons le QGen avec REINFORCE pendant 80 périodes (époques) avec descente de gradient stochastique simple (SGD) avec un taux d'apprentissage de 0,001 et une taille de lot (batch size) de 64. Pour chaque époque, nous échantillonnons chaque image de formation une fois, et nous choisissons au hasard un de ses objets comme cible. Nous optimisons simultanément les paramètres de base avec SGD avec un taux d'apprentissage de 0,001. Enfin, nous fixons le nombre maximal de questions à 8 et le nombre maximal de mots à 12.

2.10.2 Résultats

Précision : Comme nous nous intéressons aux performances au niveau humain, nous rapportons la précision des modèles sous forme de pourcentage des performances humaines (84,4%), estimées à partir de l'ensemble de données. Nous reportons les scores dans le tableau 2.7, dans lesquels nous comparons des objets d'échantillonnage de l'ensemble d'apprentissage (Nouveaux objets) et de l'ensemble d'essai (Nouvelles images), c'est-à-dire des images invisibles. Nous rapportons l'écart type sur 5 analyses afin de prendre en compte la stochasticité de l'échantillonnage. Sur la série de tests, la référence obtient une précision de 45,0%, tandis que l'entraînement avec REINFORCE passe à 62,0%. Il s'agit également d'une amélioration significative par rapport à la baseline de recherche de faisceaux (beam-search baseline), qui atteint 53,0% sur l'ensemble de tests. La procédure de recherche de faisceau (beam-search) améliore l'échantillonnage par rapport à la baseline, mais abaisse de manière intéressante le score de REINFORCE.

Echantillons : Nous comparons qualitativement les deux méthodes en analysant quelques échantillons générés, comme indiqué dans le tableau 2.8. Nous observons que la base de recherche de faisceaux formée de manière supervisée ne cesse de répéter les mêmes questions, comme on peut le voir dans les deux premiers exemples du tableau. 1. Nous avons constaté ce comportement en particulier sur l'ensemble de tests, c'est-à-dire lorsque nous sommes confrontés à des images invisibles, ce qui peut mettre en

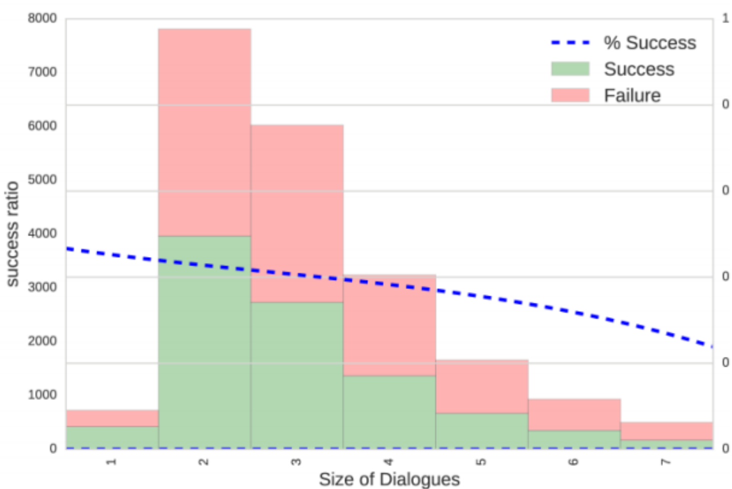
Image	Beam Search	REINFORCE	Image	Beam Search	REINFORCE
	Is it a person ? no Is it a ball ? no Is it a ball ? no Is it a ball ? no Is it a ball ? no Failure (blue bat)	Is it a person ? no Is a glove ? no Is an umbrella ? no Is in the middle ? no On a person? no is it on on far right? yes Success (red chair)		Is it a cat ? no Is it a book ? no Is it a book ? no Is it a book ? no Is it a book ? no Failure (person)	Is it a cat ? no Is it on the table ? yes Is it the book ? no Is it fully visible? yes Success (bowl)
	Is it a person ? yes Is it the one in front ? yes Is it the one on the left ? no Is it the one in the middle with the red umbrella ? yes Is it the one to the right of the girl in ? no Failure (umbrella)	Is it a person ? yes Is it girl in white ? yes Success (girl)		Is it a bag ? yes Is it red ? no Is it the one in the middle ? no Is it the one on the far right ? no Is it the one with the blue bag ? yes Success (most left bag)	Is it a suitcase? yes Is it in the left side ? yes Failure (left bag)

Échantillons extraits de l'ensemble de test. Le cadre bleu (resp. Violet) correspond à l'objet choisi par le devineur pour le dialogue de recherche de faisceau - beam search- (resp. REINFORCE). La petite description verbeuse est ajoutée pour faire référence à l'objet sélectionné par le devineur.

évidence certains problèmes de généralisation. Nous constatons également que la base de recherche de faisceaux génère des questions plus longues (7,1 tokens en moyenne) par rapport à REINFORCE (4,0 tokens en moyenne). Cette différence qualitative est clairement visible dans l'exemple en bas à gauche, qui souligne également que la ligne de base supervisée génère parfois des séquences de questions pertinentes du point de vue visuel mais incohérentes. Par exemple, demander «Is it the one to the right of the girl in ?» n'est pas une suite très logique de «Is it the one in the middle with the red umbrella ?». En revanche, REINFORCE semble mettre en œuvre une stratégie plus fondée et plus pertinente : "Is it girl in white ?" est une suite raisonnable de "Is it a person ? ?". En général, nous observons que REINFORCE est favorable pour énumérer les catégories d'objets («is it a person ?») ou d'informations spatiales absolues («Is it left ?»). Notez que ce sont également les types de questions auxquelles l'oracle est censé répondre correctement. C'est pourquoi REINFORCE est en mesure d'adapter sa stratégie aux forces de l'oracle.

Longueur du dialogue : Pour le QGen formé à REINFORCE, nous étudions l'impact de la durée du dialogue sur le taux de succès de la figure 2.9. Fait intéressant, REINFORCE apprend à s'arrêter en moyenne après 4,1 questions, bien que nous n'ayons pas codé de pénalité de question dans la fonction récompense. Le devineur peut appliquer cette règle car poser des questions supplémentaires mais bruyantes réduit considérablement la précision de prédiction du devineur comme indiqué dans Tab 2.8. Par conséquent, le QGen apprend à ne plus poser de questions lorsqu'un dialogue contient suffisamment d'informations pour récupérer l'objet cible. Cependant, nous observons que le QGen s'arrête parfois trop tôt, surtout lorsque l'image contient trop d'objets de la même catégorie. Fait intéressant, nous avons également constaté que la recherche de faisceau (beam-search) ne parvient pas à arrêter le dialogue. La recherche par faisceau utilise une vraisemblance logarithmique normalisée en longueur pour marquer les séquences candidates afin d'éviter un biais vers des questions plus courtes. Cependant, des questions dans GuessWhat ?! presque toujours commencent par «is it», ce qui augmente la log-vraisemblance moyenne d'une question de manière significative. Le score d'une nouvelle question pourrait donc (presque) toujours être supérieur à celui émis par un seul jeton <stop>. Notre conclusion a également été confirmée par le fait qu'une procédure d'échantillonnage a effectivement mis fin au dialogue.

Vocabulaire : L'échantillonnage à partir de la ligne de base supervisée sur l'ensemble de tests donne 2 893 mots uniques, tandis que celui du modèle formé par REINFORCE réduit sa taille à 1 194. Cependant, la recherche de faisceau utilise uniquement 512 mots uniques, ce qui est cohérent avec la faible variété observée de questions.



Rapport d'achèvement des tâches de QGEN formé par REINFORCE en fonction de la longueur de dialogue

Chapitre 3

Apprentissage automatique

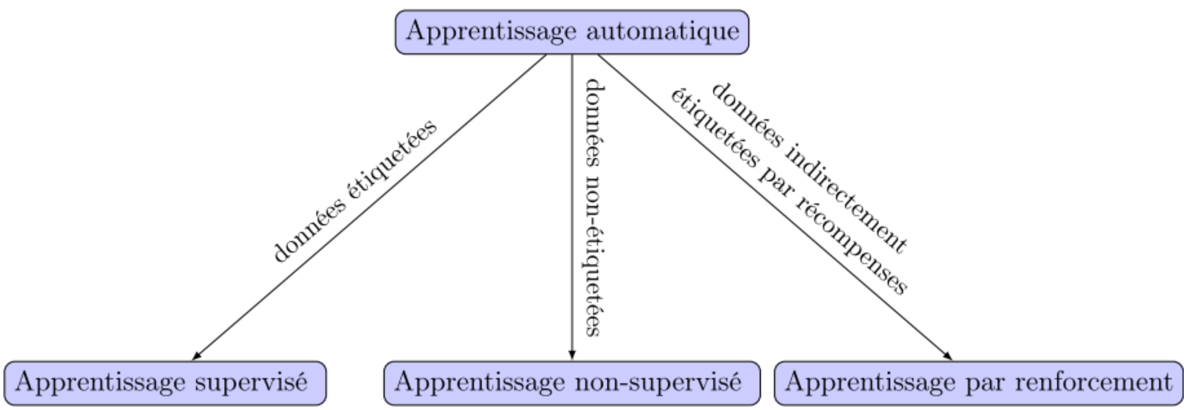
Je me base principalement dans ce chapitre sur le cours d'apprentissage de M2 de Paris Sud de Sylvain Arlot et Francis Bach , et aussi la thèse "Apprentissage par renforcement développemental" de Matthieu Zimmer.

3.1 Généralités

L'apprentissage automatique est basé sur la théorie des probabilités et les statistiques (Hastie et al., 2009) et l'optimisation (Boyd et Vandenberghe, 2004), est la base du big data, la science des données (Blei et Smyth, 2017 ; Provost et Fawcett, 2013), la modélisation prédictive (Kuhn et Johnson, 2013), l'exploration de données, la récupération d'informations (Manning et al., 2008), etc., et devient un ingrédient essentiel de la vision par ordinateur, du traitement du langage naturel, de la robotique, etc. L'apprentissage par renforcement est proche du contrôle optimal (Bertsekas, 2012), et la recherche opérationnelle et la gestion (Powell, 2011), et est également liée à la psychologie et aux neurosciences (Sutton et Barto, 2017). L'apprentissage automatique est un sous-ensemble de l'intelligence artificielle (IA) et est en train de devenir critique pour tous les domaines de l'IA.

Ce chapitre sert principalement à introduire des techniques d'apprentissage automatique que nous utiliserons en apprentissage par renforcement dans la suite . En particulier, la régression par descente de gradient avec des réseaux de neurones.

L'apprentissage automatique consiste à modéliser une application mesurable $\mathcal{C} : \mathcal{X} \mapsto \mathcal{Y}$ à partir de données qu'on appelle un prédicteur/classifieur . Selon les données disponibles et l'objectif , nous classons généralement l'apprentissage automatique en apprentissage supervisé, non supervisé et par renforcement. Dans l'apprentissage supervisé, il existe des données étiquetées ; dans l'apprentissage non supervisé, il n'y a pas de données étiquetées ; et dans l'apprentissage par renforcement, il y a des retours d'évaluation, mais



Les trois grandes classes d'apprentissage automatique

pas de signaux supervisés. La classification et la régression sont deux types de problèmes d'apprentissage supervisé, avec des sorties catégorielles et numériques respectivement.

3.2 Apprentissage supervisé

3.2.1 Approche algorithmiques

- Méthodes par moyennage local : k-ppv, Nadaraya-Watson, fenêtres de Parzen, arbres de décisions
- Méthodes par minimisation du risque empirique : modèle linéaire, méthodes à noyaux.
- Réseaux de neurones
- Méthodes de modélisation probabilistes (modèle graphiques, méthodes bayésiennes)
- Approche PAC-bayésienne

3.2.2 Formalisme

(Cours Orsay) On dispose de n observations $D_n := (X_i, Y_i)_{1\leqslant i\leqslant n}$ qui sont i.i.d de loi inconnue P. Si on se donne une nouvelle réalisation (X_{n+1}, Y_{n+1}), on veux prédire Y_{n+1} sachant Y_{n+1} en minimisant l'erreur de prédiction. Le but du classifieur $\mathcal{C}$ est de fournir une étiquette $\mathcal{C}(X_{n+1})$ à X_{n+1} en espérant de coïncider $\mathcal{C}(X_{n+1})$ avec Y_{n+1} .

On définit la fonction contraste pour mesurer la qualité du classifieur :

$$\begin{aligned}\gamma\colon \mathcal{S} \times (\mathcal{X} \times \mathcal{Y}) &\to \mathbb{R}\\ (c, (x, y)) &\mapsto \gamma(c, (x, y))\end{aligned}$$

L'objectif est désormais de trouver $c \in \mathcal{S} :=$ [ensemble des classifieurs] qui minimise la fonction perte P_γ définie par :

$$P_\gamma(c) := \mathbb{E}_{(X,Y)\sim\mathbb{P}}[\gamma(c, (X, Y))]$$

On appelle prédicteur de Bayes tout prédicteur c^* qui minimise la fonction de perte :

$$c^* := \mathrm{argmin}_{c\in\mathcal{S}} P_\gamma(c)$$

Sachant qu'on ne peut pas trouver un meilleur prédicteur que le prédicteur de Bayes, le but est alors de s'approcher au plus du prédicteur de Bayes au sens de minimiser la perte relative définie par :

$$l(c^*, c) := P_\gamma(c) - P_\gamma(c^*) \geqslant 0$$

Comme la loi P est inconnue , on doit estimer les objets introduits précédemment . On appelle estimateur toute application mesurable qui à un n-échantillon associe un classifieur :

$$\hat{c} : (\mathcal{X} \times \mathcal{Y})^n \to \mathcal{S}$$

Notons que la perte $P_\gamma(\hat{c}) := \mathbb{E}[\gamma(\hat{c}, (X, Y)) \mid D_n]$ est aléatoire. On appelle l'exces de risque l'espérance de la perte relative : $\mathbb{E}(l(c^*, \hat{c}))$.

On dit qu'il y a une consistance faible pour la loi P si :

$$\mathbb{E}(l(c^*, \hat{c}(D_n))) \overset{n\to\infty}{\to} 0$$

Régression Dans le cas de la régression on considère que : $\mathcal{Y} = \mathbb{R}$. Ici,$\mathcal{Y}$ est donc continue. On peut toujours écrire que Y et X sont reliés par la relation :

$$Y = \eta(X) + \varepsilon \tag{3.1}$$

avec $\eta(X) = \mathbb{E}[Y|X]$. Ceci implique que $\mathbb{E}[\varepsilon|X] = 0$. On dénit le contraste des moindres carrés par :

$$\gamma(t,(x,y)) = (t(x) - y)^2 \tag{3.2}$$

Si $\mathcal{Y} = \mathbb{R}$ et γ est le contraste des moindres carrés, alors, pour tout $t \in \mathbb{S}$,

$$P\gamma(t) = \mathbb{E}\left[(t(X) - \eta(X))^2\right] + P\gamma(\eta) \geqslant P\gamma(\eta) \tag{3.3}$$

En effet ,

$$\begin{aligned} P\gamma(t) &= \mathbb{E}\left[(t(X) - Y)^2\right] \\ &= \mathbb{E}\left[(t(X) - \eta(X) - \varepsilon)^2\right] \\ &= \mathbb{E}\left[(t(X) - \eta(X))^2\right] + \mathbb{E}\left[\varepsilon^2\right] - 2\mathbb{E}[\mathbb{E}[\varepsilon(t(X) - \eta(X))|X]] \end{aligned} \tag{3.4}$$

Or,

$$\mathbb{E}[\mathbb{E}[\varepsilon(t(X) - \eta(X))|X]] = \mathbb{E}[(t(X) - \eta(X))\underbrace{\mathbb{E}[\varepsilon|X]}_{=0}] = 0 \tag{3.5}$$

si bien que $\eta = s^*$ est un prédicteur de Bayes qui réalise l'égalité.

No free lunch theorem on n'a rien sans rien. Le théorème suivant montre que dans le cas de la classication il n'est pas possible d'avoir une consistance universelle faible uniforme pour le contraste 0-1 ($\gamma_{0-1}(t,(x,y)) = \gamma(t,(x,y)) = \mathbb{1}_{t(x) \neq y}$) sur l'ensemble des lois sur $\mathcal{X} \times \mathcal{Y}$ lorsque $\mathcal{X}$ est inni.

Théorème 3.2.1. Si $\mathcal{X}$ est inni, $\mathcal{Y} = \{0,1\}, \gamma = \gamma_{0-1}$, alors pour tout entier$n \in \mathbb{N}$ et pour tout estimateur $\widehat{s} : (\mathcal{X}, \mathcal{Y})^n \to \mathbb{S}$:

$$\sup_{P \text{ loi sur } \mathcal{X}\times\mathcal{Y}} \{\mathbb{E}_{D_n \sim P^{\otimes n}}\left[\ell\left(s^\star, \widehat{s}\left(D_n\right)\right)\right]\} \geqslant \frac{1}{2} \tag{3.6}$$

Démonstration. Soit un entier$K \geqslant 1$ et $A_1, \ldots, A_K \in \mathcal{X}$. Pour simplier,on suppose $A_i = i$ pour tout i. Soit $r \in \{0,1\}^K$ fixé.On dénit une loi P_r sur $\mathcal{X} \times \mathcal{Y}$ comme suit :

$$(X,Y) \sim P_r \Leftrightarrow X \text{suit une loi uniforme sur l'ensemble} \{1,\ldots,K\} \tag{3.7}$$

et $Y = r_X$ est une fonction de X uniquement. Ainsi, sous la loi Pr,$s^\star(X) = s_r^\star(X) = r_X$ et $P_r\gamma\left(s_r^\star\right) = 0$. On écrit alors que :

$$\begin{aligned} &\sup\nolimits_{P \text{ loi sur } \mathcal{X}\times\mathcal{Y}} \{\mathbb{E}_{D_n \sim P^{\otimes n}}\left[\ell\left(s^\star, \widehat{s}\left(D_n\right)\right)\right]\} \\ &\geqslant \sup\nolimits_{r\in\{0,1\}^K} \left\{\mathbb{E}_{D_n \sim P_r^{\otimes n}}\left[\ell_r\left(s_r^\star, \widehat{s}\left(D_n\right)\right)\right]\right\} \\ &= \sup\nolimits_{r\in\{0,1\}^K} \left\{\mathbb{P}_{D_n \sim P_r^{\otimes n}, (X,Y)\sim P_r}\left(\widehat{s}\left(D_n;X\right) \neq Y\right)\right\} \\ &\geqslant \mathbb{P}_{r\sim R, D_n \sim P_r^{\otimes n}, (X,Y)\sim P_r}\left(\widehat{s}\left(D_n;X\right) \neq Y\right) \end{aligned} \tag{3.8}$$

où R est une loi quelconque sur $\{0,1\}^K$. Réécrivons la dernière probabilité écrite an de pouvoir échanger l'ordre d'intégration (c'est-à-dire, prendre d'abord une moyenne vis-à-vis de r, et ensuite moyenner par rapport aux X_i et à X :

$$\begin{aligned} &\mathbb{P}_{r\sim R, D_n \sim P_r^{\otimes n}, (X,Y)\sim P_r}\left(\widehat{s}\left(D_n;X\right) \neq Y\right) \\ =&\mathbb{P}_{r\sim R, X_1,\ldots,X_n, X\sim\mathcal{U}(\{1,\ldots,K\})}\left(\widehat{s}\left(\left(X_i, r_{X_i}\right)_{1\leqslant i\leqslant n}; X\right) \neq r_X\right) \\ =&\mathbb{E}_{X_1,\ldots,X_n, X\sim\mathcal{U}(\{1,\ldots,K\})}\left[\mathbb{P}_{r\sim R}\left(\widehat{s}\left(\left(X_i, r_{X_i}\right)_{1\leqslant i\leqslant n}; X\right) \neq r_X | X_1,\ldots,X_n, X\right)\right] \end{aligned} \tag{3.9}$$

On s'intéresse désormais à la probabilité sachant $X_1, \ldots, X_n, X$ écrite ci-dessus. Il s'agit de la probabilité que l'on ait une certaine fonction de $(X, X_1, \ldots, X_n, r_{X_1}, \ldots, r_{X_n})$ égale à r_X . On souhaite choisir R telle que cette probabilité est plutôt grande.Une idée naturelle est de prendre $r_1, \ldots, r_K$ indépendantes et de même loi $\mathcal{B}\left(\frac{1}{2}\right)$. On suppose désormais que R est dénie de la sorte. Ainsi, lorsque $X \notin \{X_1, \ldots, X_n\}$, $\widehat{s}\left(\left(X_i, r_{X_i}\right)_{1\leqslant i\leqslant n}; X\right)$ est indépendante de r_X. On en déduit que :

$$\mathbb{P}_{r\sim R}\left(\widehat{s}\left(\left(X_i, r_{X_i}\right)_{1\leqslant i\leqslant n}; X\right) \neq r_X | X_1,\ldots,X_n, X\right) \geqslant \frac{1}{2}\mathbb{1}_{X\notin\{X_1,\ldots,X_n\}} \tag{3.10}$$

Méthodes	Modèles	
	linéaire ou fonctions de base fixées	quelconque (non linéaire)
solution analytique	existe	n'existe pas
descente de gradient	optimum	optimum local si gradient calculable
méthode de Newton	optimum	optimum local si Hessien calculable
optimisation *black-box*	optimum local sans calcul de gradient	

Récapitulatif des méthodes d'optimisation et leurs hypothèses.

Donc ,

$$\begin{aligned} & \mathbb{P}_{r\sim R, D_n\sim P_r^{\otimes n}}(X,Y)\sim P_r\left(\widehat{s}\left(D_n;X\right)\neq Y\right) \\ \geqslant & \frac{1}{2}\mathbb{P}\left(X\notin\{X_1,\dots,X_n\}\right) \\ = & \frac{1}{2}\mathbb{E}\left[\mathbb{P}\left(X_1\neq X,\dots,X_n\neq X\right)|X\right] \\ = & \frac{1}{2}\mathbb{E}\left[\mathbb{P}\left(X_1\neq X|X\right)\times\cdots\times\mathbb{P}\left(X_n\neq X\right)|X\right] \\ = & \frac{1}{2}\left(1-\frac{1}{K}\right)^n \end{aligned} \tag{3.11}$$

En faisant tendre K vers $+\infty$, on obtient la minoration cherchée. □

3.2.3 Optimisation des paramètres

Nous allons maintenant examiner les moyens d'apprendre les paramètres des modèles précédents. Pour cela, la qualité d'un modèle doit d'abord être définie. Pour mesurer la qualité, un critère est utilisé ; on parle aussi d'objectif. L'un des critères les plus utilisés dans la régression se nomme MSE ou erreur quadratique moyenne.

Definition 3.2.1. Soit un modèle $\Psi : X \to Y$ et un ensemble de données étiquetées $(x_i, y_i)_{\{1,\dots,n\}} \in X^n \times Y^n$, le critère empirique MSE dénit la qualité du modèle par :

$$\mathrm{MSE}(\Psi) = \frac{1}{n}\sum_{i=1}^{n}\|\Psi(x_i) - y_i\|_2^2$$

où $\|\cdot\|_2$ représente la norme $L2$.

Plus la MSE est petite, meilleur est le modèle. Pour un modèle paramétrique Ψ_θ , les paramètres optimaux θ^*_{MSE} sont donc :

$$\theta^*_{\mathrm{MSE}} = \arg\min_{\theta\in\Theta}\frac{1}{n}\sum_{i=1}^{n}\|\Psi_\theta(x_i) - y_i\|_2^2$$

Pour résoudre ce problème d'optimisation, il existe de nombreuses méthodes ayant chacune des hypothèses et garanties diérentes. On cite les méthodes les plus couramment utilisées dans la table [3.2]. Un gradient calculable pour J signie que $\nabla_\theta J(\Psi_\theta)$ existe.

Descente de gradient à pas fixe

Je me base principalement dans cette partie sur le cours d'optimisation de Pr. Pierre Gilles Lemarié-Rieusset.

Nous allons détailler la descente de gradient, car c'est cette méthode la plus utilisée. La raison de son utilisation étant sa capacité de mise à l'échelle. En eet, le calcul de la prochaine solution est linéaire : il se fait en $O(n)$ opérations où n est le nombre de paramètres. Par rapport aux méthodes d'optimisation black-box, la descente de gradient est plus informée, la rendant plus ecace lorsque n est grand et que le

gradient est pertinent. Néanmoins, comme toutes méthodes dépendantes du gradient, elles sont sujettes à rester bloquées dans un optimal local et sont lentes lorsque des plateaux surviennent dans le gradient.

On suppose que les hypothèses suivantes soient verifiées :

- H1 : Ω est un ouvert convexe de $\mathbb{R}^d$.
- H2 : J est une fonction de classe $\mathcal{C}^2$ de Ω dans $\mathbb{R}$
- H3 : $\forall x \in \Omega, z \to d^2J(x)(z,z)$ est une forme bilinéaire définie positive.
- H4 : $x_0 \in \Omega$ et $K = \{x \in \Omega / J(x) \leqslant J(x_0)\}$ est un compact de Ω et x^* le point optimal de J.

Une conséquence de H3 et H4 est l'ellipticité de J :

$$\exists a_K > 0, \forall x \in K, \forall z \in \mathbb{R}^d \quad d^2J(x)(z,z) \geqslant a_K \|z\|^2$$

En effet, la meilleure constante a_K est :

$$a_K = \min_{(x,z)\in K\times S^{d-1}} d^2J(x)(z,z)$$

qui est bien un nombre strictement positif (a_K est atteint, en au moins un point (x_m, z_m) par compacité de $K \times S^{d-1}$, de plus, $a_K > 0$ puisque $d^2J(x_m)$ est une forme bilinéaire définie positive).

Ensuite,la compacité et la convexité de K entrainent aussi la lipschitzianité des dérivées premières :

$$\exists A_K \geqslant 0, \forall x \in K, \forall y \in K \quad \|J'(x) - J'(y)\| \leqslant A_K \|x - y\|$$

où

$$A_K = \min_{(x,z)\in K\times S^{d-1}} d^2J(x)(z,z) = \sup_{x\in K} \|H_J(x)\|_{op}$$

Un lemme important,

Lemme 3.2.1. *Sous les hypothèses (H1), (H2), (H3), (H4), on a pour tout* $x \in K$ *:*

$$\frac{a_K}{2}\|x - x^*\|^2 \leqslant J(x) - J(x^*) \leqslant \frac{A_K}{2}\|x - x^*\|^2$$

Démonstration. La formule de Taylor à l'ordre 2 donne pout tout $x \in K$:

$$J(x) = J(x^*) + \int_0^1 {}^t(X - X^*)\, H_J(x^* + \theta(x - x^*))\,(X - X^*)\,(1-\theta)d\theta$$

avec :

$$a_K\|x - x^*\|^2 \leqslant {}^t(X - X^*)\, H_J(x^* + \theta(x - x^*))\,(X - X^*) \leqslant A_K\|x - x^*\|^2$$

Le lemme est immédiat. □

Une remarque importane : Si $x \in K$ et si $x \neq x^*$, on définit l'intervalle $I_x = \{t \in \mathbb{R} / x - t\vec{\nabla}J(x) \in \Omega\}$. C'est un intervalle ouvert qui contient $t = 0$ et au voisinage de 0 on a d'après Taylor :

$$J(x - t\vec{\nabla}J(x)) = J(x) - t\|\vec{\nabla}J(x)\|^2 + O\left(t^2\right)$$

L'ensemble

$$A_x = \left\{t \in I_x / J(x - t\vec{\nabla}J(x)) < J(x)\right\}$$

est donc un intervalle ouvert non vide de la forme $A_x =]0, T_x[$.Pour $t \in A_x$:

$$\frac{a_K}{2}t^2\|\vec{\nabla}J(x)\|^2 \leqslant J(x - t\vec{\nabla}J(x)) - J(x) - t\|\vec{\nabla}J(x)\|^2 \leqslant \frac{A_K}{2}t^2\|\vec{\nabla}J(x)\|^2$$

En particulier,

$$\frac{2}{A_K} \leqslant T_x \leqslant \frac{2}{a_K}$$

Dans l'algorithme de la descente à pas dixe , le pas ρ ne dépends pas de l'itération , et pour garantir la stabilité dans K , on suppose de plus que $0 < \rho < \frac{2}{A_K}$ d'après la remarque précedente .

Théorème 3.2.2. On suppose les hypothèses H1,H2,H3 et H4 vérifiées. On fixe ρ tel que $0 < \rho < \frac{2}{A_K}$. On définit x_k par récurrence à partir de x_0 par :

$$x_{k+1} = x_k - \rho\vec{\nabla}J(x_k)$$

Alors la suite (x_k) converge vers le point optimal x^* de J.

Démonstration. On a vu que si $0 < \rho < \frac{2}{A_K}$, alors, lorsque $x \in K$, $x - \rho\vec{\nabla}J(x)$ reste bien dans K, de sorte que la suite (x_k) est bien dénie par récurrence. Supposons que $J'(x) \neq 0$.

De plus, on a :

$$J(x - \rho\vec{\nabla}J(x)) \leqslant J(x) - \rho\,\|J'(x)\|^2 + \frac{A_K}{2}\rho^2\,\|J'(x)\|^2$$

Donc , $J(x_{k+1}) \leqslant J(x_k)$. Comme $(J(x_k))_{k\in\mathbb{N}}$ est décoissante et minorée par $J(x^*)$, elle est convergente. De plus, on a :

$$\rho\left(1 - \frac{A_K\rho}{2}\right)\|J'(x_k)\|^2 \leqslant J(x_k) - J(x_{k+1})$$

On en déduit que :

$$\lim_{k\to+\infty} J'(x_k) = 0$$

de sorte que

$$\limsup_{k\to+\infty}\|x_k - x^*\| \leqslant \frac{1}{a_K}\lim_{k\to+\infty}\|J'(x_k)\| = 0$$

La convergence est donc démontrée. □

Algorithme 3.2.1 (Descente de gradient)**.** [these] Étant donné un modèle paramétrique $\Psi_\theta : X \to Y$ avec un critère $J(\Psi_\theta)$ continu et dérivable à minimiser, la descente de gradient met à jour θ par :

$$\theta_{t+1} \leftarrow \theta_t - \alpha\frac{\partial J(\Psi_{\theta_t})}{\partial\theta_t}$$

où α est un taux d'apprentissage.

Au début, θ_0 est initialisé aléatoirement. Le taux d'apprentissage α définit la vitesse de déplacement des paramètres qui peut changer à chaque itération. Le choix de ce taux pose un problème : un déplacement trop lent nécessitera plus d'itération,or un déplacement trop rapide peut entraîner une divergence des paramètres, ce qui justifie nos hypothèses précedentes de stabilité et de convergence. La difficulté réelle de ces méthodes de descente de gradient réside dans la bonne estimation du gradient. Le nombre de données, utilisées pour calculer J, détermine la version de descente de gradient utilisée. La descente de gradient stochastique se sert d'un sous-ensemble tiré aléatoirement parmi les échantillons disponibles. On utilise également le terme de mini-batch, ou encore online lorsque qu'un seul échantillon est utilisé. Ainsi, le calcul du critère J, et donc de son gradient, est moins coûteux en temps, car il est seulement calculé sur un sous-ensemble. La version stochastique permet potentiellement de sortir des optima locaux. Tandis que la version batch, qui utilise toutes les données disponibles, calcule une meilleure approximation du gradient.

Pour pallier ce problème de définition de taux d'apprentissage , il existe des algorithmes permettant de ne pas définir le taux d'apprentissage a prior (Igel et Hüsken, 2000; Riedmiller et Braun, 1992). Dans ce cas , il est défini par l'algorithme lui meme . L'algorithme Rprop 4 permet cela (Algorithme 1.2), de plus, au lieu d'avoir un seul taux d'apprentissage scalaire, α devient un vecteur définissant un taux d'apprentissage par paramètre, accélérant ainsi l'apprentissage (LeCun et al., 2012). Le principe de Rprop est de regarder si le signe du gradient a changé par rapport au précédent, s'il n'a pas changé, le taux d'apprentissage du paramètre augmente; dans le cas contraire, il diminue. Pour avoir une certaine

stabilité, il est ainsi préférable d'utiliser une partie importante de l'échantillon disponible.

Data: Un modèle paramétrique Ψ_θ, un critère J à minimiser,$(\Delta_{max}, \Delta_{min}) \in \mathbb{R}_+^2$: la variation maximale et minimale sur une itération, $(\Delta^+, \Delta^-) \in \mathbb{R}_+^2$: l'augmentation et la diminution de la variation sur une itération,et $t \in \mathbb{N}$: l'itération

initialization;

for $\theta_i \in \Theta$ **do**

$z \leftarrow \nabla_{\theta_i} J^{(t-1)} \cdot \nabla_{\theta_i} J^{(t)}$

$$\Delta\theta_i^{(t)} \leftarrow \begin{cases} \min\left(\Delta\theta_i^{(t-1)}\Delta^+, \Delta_{max}\right) & si\ z < 0 \\ \max\left(\Delta\theta_i^{(t-1)}\Delta^-, \Delta_{min}\right) & si\ z > 0 \end{cases}$$

if *z<0* **then**

$\nabla_{\theta_i} J^{(t)} \leftarrow 0$**end**

end

$\Delta\theta_i^{(t)} \leftarrow -\Delta\theta_i^{(t)}\ \text{signe}\left(\nabla_{\theta_i} J^{(t)}\right)$

$\theta_i^{(t+1)} \leftarrow \theta_i^{(t)} + \Delta\theta_i^{(t)}$

Algorithm 1: Resilient backpropagation iRPROP

Il existe de nombreuses améliorations possibles à la descente de gradient stochastique classique. Il s'agit d'ADAM (Adaptive Moment Estimation Optimizer), l'algorithme [] le plus utilisé qui estime la moyenne et la variance du gradient pour chaque paramètre de façon géométriquement décroissante par rapport au temps (Kingma et Ba, 2015). Grâce à ces estimations, la mise à jour des paramètres est plus lisse et la variance réduite. Néanmoins, dans cet algorithme, il faut définir un taux d'apprentissage global α. Cet algorithme est utilisé en deep learning (Goodfellow et al., 2016) et en renforcement (Lillicrap et al., 2015).

Data: Un modèle paramétrique Ψ_θ, un critère J à minimiser,$(\Delta_{max}, \Delta_{min}) \in \mathbb{R}_+^2$: la variation maximale et minimale sur une itération, $(\Delta^+, \Delta^-) \in \mathbb{R}_+^2$: l'augmentation et la diminution de la variation sur une itération,et $t \in \mathbb{N}$: l'itération

for $\theta_i \in \Theta$ **do**

$z \leftarrow \nabla_{\theta_i} J^{(t-1)} \cdot \nabla_{\theta_i} J^{(t)}$

$$\Delta\theta_i^{(t)} \leftarrow \begin{cases} \min\left(\Delta\theta_i^{(t-1)}\Delta^+, \Delta_{max}\right) & si\ z < 0 \\ \max\left(\Delta\theta_i^{(t-1)}\Delta^-, \Delta_{min}\right) & si\ z > 0 \end{cases}$$

if *z<0* **then**

$\nabla_{\theta_i} J^{(t)} \leftarrow 0$**end**

end

$\Delta\theta_i^{(t)} \leftarrow -\Delta\theta_i^{(t)}\ \text{signe}\left(\nabla_{\theta_i} J^{(t)}\right)$

$\theta_i^{(t+1)} \leftarrow \theta_i^{(t)} + \Delta\theta_i^{(t)}$

Algorithm 2: Resilient backpropagation iRPROP

Data: Un modèle paramétrique Ψ_θ, un critère J à minimiser,$\alpha \in \mathbb{R}$: le taux d'apprentissage , $(\beta_1, \beta_2) \in \mathbb{R}_+^2$: taux d'apprentissage pour l'estimation exponentiellement décroissante de la moyenne et de la variance, $\zeta \in \mathbb{R}$:décalage pour éviter une division par 0 et $t \in \mathbb{N}$ l'itération

for $\theta_i \in \Theta$ **do**

$\mu_i^{(t)} \leftarrow \beta_1 \mu_i^{(t-1)} + (1-\beta_1)\nabla_{\theta_i} J^{(t)}$

$\sigma_i^{(t)} \leftarrow \beta_2 \sigma_i^{(t-1)} + (1-\beta_2)\nabla_{\theta_i} J^{(t)^2}$

$\theta_i^{(t+1)} \leftarrow \theta_i^{(t)} - \alpha \frac{\sqrt{1-\beta_2}}{1-\beta_1} \frac{\mu_i^{(t)}}{\sqrt{\sigma_i^{(t)}}+\zeta}$

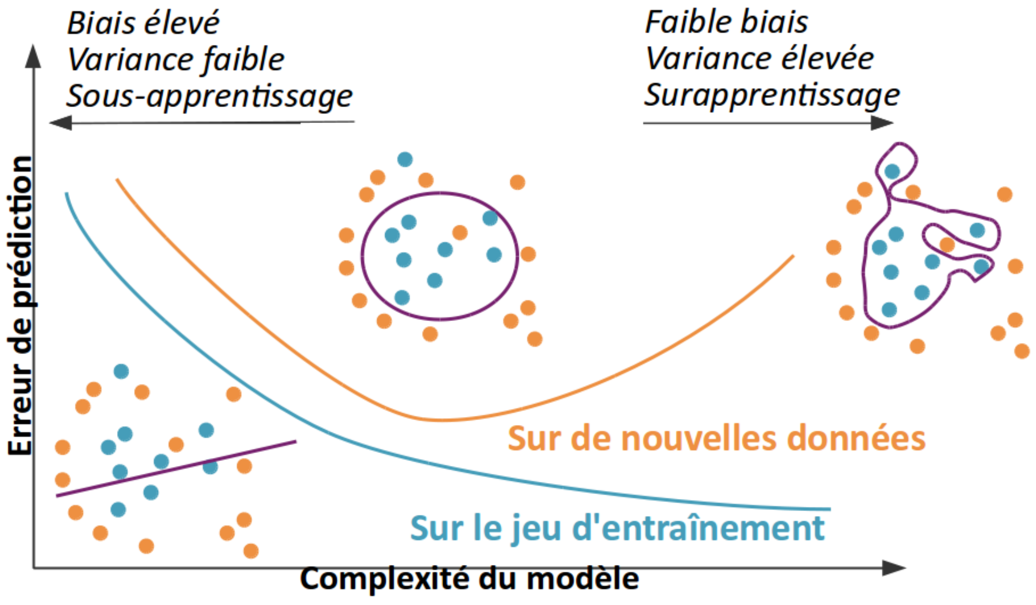
end

Algorithm 3: ADAM : Adaptive Moment Estimation Optimizer

Il existe d'autres méthode d'optimisation telles que les méthodes de second ordre (Newton..) ou

3.2.4 Limitations

Il existe plusieurs limitations à l'utilisation de modèles pour approcher une fonction. Nous allons maintenant aborder les plus importantes.



Exemple de sur/sous-apprentissage [OpenClassrooms]

Surapprentissage ou généralisation

Lors de l'optimisation des paramètres, il est possible qu'un phénomène de surapprentissage [Fig.3.3] survienne : il s'agit d'une mauvaise généralisation des données qui n'a pas été présentée dans le modèle (figure 1.6 à droite). La source du surapprentissage peut provenir essentiellement de deux origines : trop de liberté laissée au modèle (trop de neurones, de couches, de fonctions de base, etc.) ou un temps d'apprentissage long pour certaines données. On dit d'un modèle qui a trop appris qu'il avait réussi à apprendre le bruit présent dans l'échantillon et non la relation sous-jacente entre l'entrée et la sortie.

Pour détecter ce phénomène d'apprentissage, il est possible d'utiliser la validation croisée (Kohavi, 1995). Dans sa version la plus simple, il s'agit de couper l'échantillon, sur lequel J est calculé, en deux sous-ensembles : l'ensemble d'apprentissage et l'ensemble de test.Les échantillons appartenant à l'ensemble d'apprentissage seront utilisés pour mettre à jour les paramètres du modèle. Tandis que ceux appartenant à l'ensemble de test ne serviront qu'à évaluer le modèle sans modification.

La régularisation permet de modifier le critère à optimiser de façon à privilégier les modèles avec une complexité plus simple. La justification de cette pratique provient du principe du rasoir d'Ockham. En pratique, elle permet de limiter de manière efficace le surapprentissage (Girosi et al., 1995). Mathématiquement, on ajoute un terme au critère J à optimiser :

$$J(\Psi_\theta) + \beta\mathcal{R}(\Psi_\theta) \tag{3.12}$$

où $\mathcal{R}$ est le terme de régularisation, et β un méta paramètre permettant d'équilibrer les deux termes. Le terme de régularisation peut prendre de nombreuses formes (Bishop, 2006) :

1. une distance L1 sur les paramètres : $\mathcal{R}(\Psi_\theta) = \|\theta\|_1$. Elle privilégie les ensembles de poids épars.
2. une distance L2 : $\mathcal{R}(\Psi_\theta) = \|\theta\|_2$.Cette distance empêche les paramètres de devenir trop grands. Dans les réseaux de neurones.

3.3 Apprentissage par renforcement

L'apprentissage par renforcement (Sutton et Barto,1998) repose sur l'utilisation de données indirectement étiquetées par des récompenses. Cet étiquetage est moins informatif qu'en apprentissage supervisé. Comme on a vu que en apprentissage supervisé,il existe un oracle de type oracle(x_i) = y_i pour étiqueter les données à priori,pourtant en l'apprentissage par renforcement cet oracle n'est capable que de quantifier une relation de type : oracle(x_i, y_i) = récompense. De plus, les données (et leur ordre) présentées à l'oracle ne sont pas maîtrisées entièrement par l'utilisateur. L'hypothèse d'avoir des données i.i.d n'est pas valable et il n'existe pas de base de données a priori. Or, cette hypothèse est largement utilisée dans l'analyse formelle de nombreux algorithmes d'apprentissage automatique.[5]

Le chapitre 4 est consacré à l'apprentissage par renforcement avec des explications plus profondes.

3.4 Apprentissage non supervisé

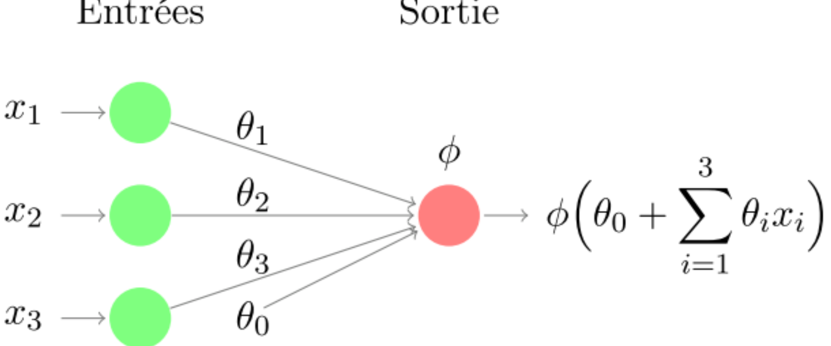
L'apprentissage non supervisé tente d'extraire des informations à partir de données sans étiquette, par exemple, une classification et une estimation de la densité. L'apprentissage par représentation est un type classique d'apprentissage non supervisé. Cependant, les réseaux de formation continue ou les réseaux de neurones convolutionnels avec apprentissage supervisé constituent une forme d'apprentissage par la représentation. L'apprentissage de la représentation (l'ACP par exemple) trouve une représentation permettant de conserver autant d'informations que possible sur les données d'origine, afin de la rendre plus simple ou plus accessible que les données d'origine, avec des représentations de faible dimension, peu nombreuses et indépendantes. Cette classe d'apprentissage n'a pas été utilisée dans ce stage,nous ne la détaillerons donc pas plus.

Parmi les algorithmes les plus couramment utilisés dans l'apprentissage non supervisé, citons [Wiki] :

1. Clustering :
   - Classification hiérarchique
   - K-means
   - Modèles de mélange
   - DBSCAN
   - OPTICS
2. Détection d' anomalies
3. Les réseaux de neurones
   - Autoencodeurs
   - Deep Belief Nets
   - Hebbian Learning
   - Generative Adversarial Networks
   - Self-organizing map
4. Approches d'apprentissage de modèles à variables latentes telles que :
   - Expectation–maximization algorithme (EM)
   - Méthode des moments
   - Analyse des composants principaux (ACP)
   - Décomposition en valeurs singulières

L'apprentissage en profondeur, ou réseaux de neurones profonds, est un schéma d'apprentissage automatique particulier, généralement destiné à un apprentissage supervisé ou non supervisé, et pouvant être intégré à l'apprentissage par renforcement, généralement en tant qu'approximateur de fonctions. L'apprentissage supervisé et non supervisé est généralement ponctuel, myope, compte tenu d'une récompense immédiate ; tandis que l'apprentissage par renforcement est séquentiel, clairvoyant et considère une récompense accumulée à long terme.

Nous allons maintenant aborder le modèle qui nous intéresse dans notre recherche : les réseaux de neurones.



Un neurone formel

3.4.1 Réseaux de neurones

Nous nous basons sur [doc]. Pour plus de détails avec un aspect mathématique, on refère qu chapitre []. On introduit d'abord la définition d'un modèle linéaire avec fonctions de base fixées :

Modèle linéaire avec fonctions de base fixées

C'est un modèle paramétrique, potentiellement discret sur $\mathcal{X}$. La sortie globale peut être non linéaire, néanmoins les opérations avec le paramètre θ restent linéaires. Ces modèles sont très utilisés, mais nécessitent de définir un ensemble de fonctions de base a priori. Formellement :

Definition 3.4.1. Soit $\Psi_{\theta,\phi} : \mathbb{R}^n \to \mathbb{R}$ un modèle linéaire avec fonctions de base fixées $\phi_i : \mathbb{R}^n \to \mathbb{R}$:

$$\forall x \in \mathbb{R}^n : \Psi_{\theta,\phi}(x) := \sum_{i=1}^{m} \theta_i \phi_i(x), \tag{3.13}$$

avec $m = |\theta|$: le nombre de paramètres.

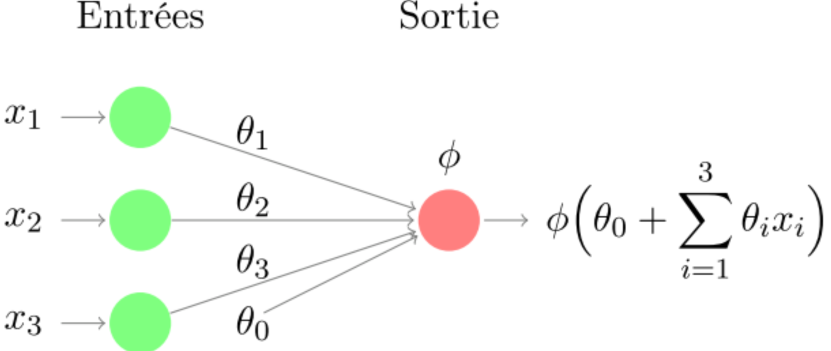
Nous allons maintenant aborder le modèle qui intéresse : les réseaux de neurones. Ce sont des modèles paramétriques continus. Contrairement aux modèles linéaires avec des fonctions de base fixes, il n'est pas nécessaire de fournir des fonctions de base. Nous allons toujours définir des opérations de base, également appelées fonctions primitives ou fonctions d'activation, qui seront utilisées pour construire automatiquement l'équivalent des fonctions de base du réseau de neurones. De plus, ils ne seront pas simplement corrigés, mais devront être améliorés : leurs paramètres sont intégrés dans θ. Plutôt que de fournir les fonctions de base, nous fournissons une structure et des opérateurs élémentaires. Il existe de nombreuses structures de réseaux de neurones. On choisit généralement de travailler avec des perceptrons multicouches pour plus de simplicité et de bonnes performances. (Hornik et al., 1989), ils peuvent théoriquement approximer n'importe quelle fonction avec un nombre suffisant de neurones. Le réseau est organisé en couches où chaque neurone est connecté à tous les neurones de la couche suivante. Cette organisation ne fait pas a priori d'hypothèses sur les données d'entrée, ce qui est intéressant. Le réseau est composé d'unités simples, les neurones formels (Fig'3.4). Le neurone formel est un modèle paramétrique, continu et statique. En combinant plusieurs neurones sur plusieurs couches, nous obtenons un perceptron multicouche (fig). C'est un modèle non linéaire à la fois en sortie et en paramètre.

Definition 3.4.2. Soit $\Psi_{\theta,\phi} : \mathbb{R}^n \to \mathbb{R}$ un perceptron multicouche avec m fonctions d'activation $\phi_i : \mathbb{R} \to \mathbb{R}$ fixées par couche, et m couches cachées composées de $(h_k)_{\{1,\dots,m\}} \in \mathbb{N}^m$ neurones chacune, alors pour tout $x \in \mathbb{R}^n$:

$$\Psi_{\theta,\phi}(x) = \Psi_{\theta,\phi}(x, 1, m+1),$$

avec $\Psi_{\theta,\phi}(x, j, k)$ déni récursivement comme étant :

$$\Psi_{\theta,\phi}(x, j, k) = \phi_k \left(\theta_{0,j,k} + \sum_{i=1}^{h_{k-1}} \theta_{i,j,k} \Psi_{\theta,\phi}(x, j, k-1) \right)$$



Un perceptron multicouche avec $X = \mathbb{R}^4$ et une couche cachée de 3 neurones.

$$\Psi_{\theta,\phi}(x, j, 1) = \phi_1 \left(\theta_{0,j,1} + \sum_{i=1}^{n} \theta_{i,j,1} x_i \right)$$

où $k \in \{1, \ldots, m+1\}$ indice les couches, $j \in \{1, \ldots, h_k\}$ indice les neurones des couches et i indice les poids d'un perceptron. Le nombre de paramètres $|\theta| = (n+1) \cdot h_1 + \sum_{i=2}^{m} (h_{i-1}+1) \cdot h_i + h_m + 1$

Pour éviter le calcul répetitif de l'activation d'un même neurone , on fait un calcul couche par couche en partant de la couche d'entrée et en gardant en mémoire chaque activation pour la couche suivante.

Dans cet exemple [Fig. 3.5], la fonction calculée est la suivante :

$$\Psi_{\theta,\phi}(x) = \phi_2 \left(\theta_{0,1,2} + \sum_{j=1}^{3} \left(\theta_{j,1,2} \phi_1 \left(\theta_{0,j,1} + \sum_{i=1}^{4} \theta_{i,j,1} x_i \right) \right) \right)$$

Pour plus de détails voir [10].

Chapitre 4

Apprentissage par renforcement

Je me base dans ce chapitre sur [4],[11] et [3].

4.1 Introduction

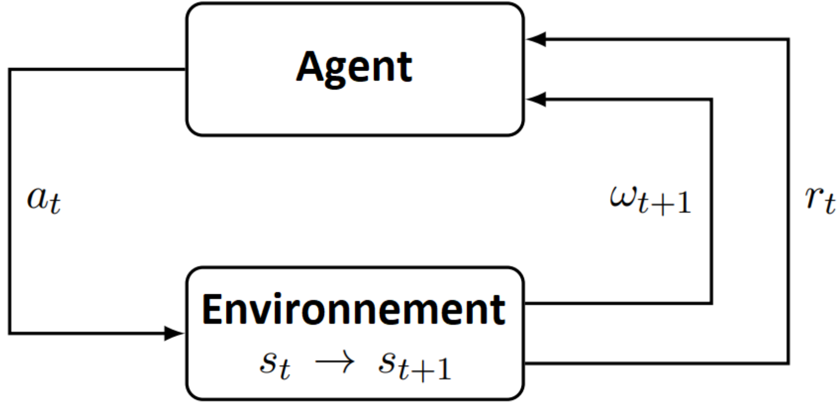
L'apprentissage par renforcement est un domaine de l'apprentissage automatique qui s'intéresse à la façon dont les agents logiciels doivent prendre des mesures dans un environnement afin de maximiser la notion de récompense cumulative. L'apprentissage par renforcement est considéré comme l'un des trois paradigmes d'apprentissage automatique, aux côtés de l'apprentissage supervisé et de l'apprentissage non supervisé.

Il diffère de l'apprentissage supervisé en ce que les paires d'entrée / sortie étiquetées ne doivent pas être présentées et que les actions sous-optimales ne doivent pas être explicitement corrigées. Au lieu de cela, l'objectif est de trouver un équilibre entre l'exploration (d'un territoire inexploré) et l'exploitation (du savoir actuel).

L'environnement est généralement formulé sous la forme d'un processus de décision de Markov (MDP), car de nombreux algorithmes d'apprentissage par renforcement utilisés dans ce contexte utilisent des techniques de programmation dynamique. La principale différence entre les méthodes classiques de programmation dynamique et les algorithmes d'apprentissage par renforcement réside dans le fait que ces derniers ne supposent pas la connaissance d'un modèle mathématique exact du MDP et ciblent de grands MDP où les méthodes exactes deviennent irréalisables.

Un aspect clé de RL est qu'un agent apprend un bon comportement. Cela signifie qu'il modifie ou acquiert de nouveaux comportements et de nouvelles compétences. Un autre aspect important de RL est qu'il utilise l'expérience des essais et des erreurs (par exemple, une programmation dynamique qui suppose une connaissance totale de l'environnement a priori). Ainsi, l'agent RL ne nécessite pas une connaissance ou un contrôle complet de l'environnement ; il doit seulement être capable d'interagir avec l'environnement et de collecter des informations. En mode offline, l'expérience est acquise a priori, puis elle est utilisée en tant que lot d'apprentissage (le paramètre en mode offline est également appelé lot RL).

Ceci est en contraste avec le mode online où les données deviennent disponibles dans un ordre séquentiel et sont utilisées pour mettre à jour progressivement le comportement de l'agent. Dans les deux cas, les algorithmes d'apprentissage de base sont essentiellement les mêmes, mais la principale différence est que, dans un environnement en ligne, l'agent peut influencer la manière dont il rassemble l'expérience pour qu'il soit le plus utile pour l'apprentissage. C'est un défi supplémentaire, principalement parce que l'agent doit faire face aux dilemme exploration / exploitation pendant l'apprentissage. Mais l'apprentissage en ligne peut également constituer un avantage, car l'agent est capable de collecter des informations spécifiques sur la partie la plus intéressante de l'environnement. Pour cette raison, même lorsque l'environnement est parfaitement connu, les approches RL peuvent constituer l'approche la plus efficace sur le plan des calculs par rapport à certaines méthodes de programmation dynamiques qui seraient inefficaces en raison de ce manque de spécificité.



Interaction agent-environnement dans RL

4.2 Formalisation

4.2.1 Le cadre d'apprentissage du renforcement

Le problème RL général est formalisé en tant que processus de contrôle stochastique temporel discret dans lequel un agent interagit avec son environnement de la manière suivante : l'agent démarre, dans un état donné dans son environnement $s_0 \in \mathcal{S}$, en recueillant une première observation $\omega_0 \in \Omega$.A chaque pas de temps t,l'agent doit prendre une action $a_t \in \mathcal{A}$. Comme illustré à la figure **??**, il en résulte trois conséquences :

1. L'agent obtient une récompense $r_t \in \mathcal{R}$.
2. La transiton de l'état $s_t \in \mathcal{S}$ à $s_{t+1} \in \mathcal{S}$
3. L'agent obtient une observation $\omega_{t+1} \in \Omega$.

Cette configuration a été proposée par Bellman, 1957b, puis étendu à l'apprentissage par Barto et al., 1983. Sutton et Barto, 2017, traitent de manière exhaustive les principes fondamentaux de RL. Nous en présentons ici les principaux éléments de RL.

4.2.2 La propriété de Markov

Par souci de simplicité, considérons d'abord le cas des processus de contrôle stochastiques markoviens .

Definition 4.2.1. Un processus de contrôle stochastique temporel discret est markovien (c'est-à-dire qu'il a la propriété de Markov) si :

$$\bullet\ \mathbb{P}(\omega_{t+1}|\omega_t, a_t) = \mathbb{P}(\omega_{t+1}|\omega_t, a_t, \dots, \omega_0, a_0)\,, \text{ et}$$
$$\bullet\ \mathbb{P}(r_t|\omega_t, a_t) = \mathbb{P}(r_t|\omega_t, a_t, \dots, \omega_0, a_0)$$

La propriété de Markov signifie que l'avenir du processus dépend uniquement de l'observation actuelle et que l'agent n'a aucun intérêt à consulter l'historique complet.

Un processus de décision de Markov (MDP) (Bellman, 1957a) est un processus de contrôle stochastique temporel discret défini comme suit :

Definition 4.2.2. Un MDP est un 5-tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$ avec :

— $\mathcal{S}$ est l'espace d'état,
— $\mathcal{A}$ est l'espace d'action,
— $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ est la fonction de transition (ensemble de probabilités de transition conditionnelles entre états),
— $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ une fonction de récompense avec :

$$\mathcal{R}(s, a) = \mathcal{R}_{sa} = \mathbb{E}\{r_{t+1}|s_t = s, a_t = a\}$$

— $\gamma \in [0, 1[$ est le facteur discount.

Le système est entièrement observable dans un MDP, ce qui signifie que l'observation est la même que l'état de l'environnement : $\omega_t = s_t$. A chaque pas de temps t, la probabilité de passer à $\mathcal{S}_{t+1}$ est donnée par la fonction de transition d'état $\mathcal{T}(s_t, a_t, s_{t+1})$ et la récompense est donnée par une fonction de récompense bornée $\mathcal{R}(s_t, a_t)$. Ceci est illustré à la figure [fig].

On définit formellement un problème d'apprentissage par renforcement comme un PDM où $\mathcal{T}$ et $\mathcal{R}$ sont à priori inconnus.

4.2.3 Différentes catégories de politiques

Une politique (ou stratégie) définit la manière dont un agent sélectionne les actions. Les politiques peuvent être classées sous le critère d'être stationnaire ou non stationnaire. Une stratégie non stationnaire dépend du pas de temps et est utile dans le contexte réseau fini, dans lequel les récompenses cumulées que l'agent cherche à optimiser se limitent à un nombre fini de pas de temps futurs (Bertsekas et al., 1995). Dans cette introduction à RL profond, les horizons infinis sont considérés et les politiques sont stationnaires.

Les politiques peuvent également être classées selon un deuxième critère, soit déterministe, soit stochastique :

— Dans le cas déterministe, la politique est décrite par : $\pi(s) : \mathcal{S} \to \mathcal{A}$

— Dans le cas stochastique, la politique est décrite par : $\pi(s, a) : \mathcal{S} \times \mathcal{A} \to [0, 1]$ ou $\pi(s, a)$ dénote la probabilité qu'une action a puisse être choisie dans l'état s.

4.2.4 Le retour attendu et fonction valeur

Le retour attendu est le cumul des récompenses obtenues à partir d'un instant t :

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \tag{4.1}$$

où T est l'instant final si la vie de l'agent peut être découpée en épisodes de plusieurs pas de temps , et on dit que les taches sont épisodiques.Autrement, on a des taches continues, dans ce cas l'instant final serait $T = \infty$ et le retour R_t pourrait être infini. Donc pour borner ce dernier , on introduit la notion de retour déprécié :

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} = r_{t+1} + \gamma R_{t+1} \tag{4.2}$$

où $\gamma \in [0, 1]$ est le facteur de dépréciation qui permet de régler l'importance accordée aux récompenses futures vis-à-vis des récompenses immédiates.Le plus souvent, on choisit $\gamma \in]0, 1[$.

Nous considérons le cas d'un agent RL dont le but est de trouver une politique $\pi(s, a) \in \Pi$, afin d'optimiser le retour attendu $V^\pi(s) : \mathcal{S} \to \mathbb{R}$ appelée fonction valeur (V-value) qui estime à quel point il est bon pour l'agent d'être dans un état particulier. On a :

$$V^\pi(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, \pi\right] \tag{4.3}$$

avec :

— $r_t = \underset{a \sim \pi(s_t, \cdot)}{\mathbb{E}} R(s_t, a, s_{t+1})$

— $\mathbb{P}(s_{t+1} | s_t, a_t) = \mathcal{T}(s_t, a_t, s_{t+1})$ avec $a_t \sim \pi(s_t, \cdot)$

Cette équation peut être récrite récursivement dans le cas d'un MDP :

$$
\begin{aligned}
V^{\pi}(s) &= E_{\pi}\{R_t|s_t = s\} \\
&= E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\right\} \\
&= E_{\pi}\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t = s\right\} \\
&= \sum_{a\in\mathcal{A}(s)}\pi(s,a)\left[\mathcal{R}_{sa} + \gamma\sum_{s'\in\mathcal{S}^+}\mathcal{T}_{sa}^{s'}V^{\pi}(s')\right]
\end{aligned}
\tag{4.4}
$$

avec $\mathcal{T}_{sa}^{s'} = \mathcal{T}(s,a,s')$. Cette dernière, s'appelle l'équation de Bellman pour V^{π}.

De plus, le retour optimal est défini par :

$$V^*(s) = \max_{\pi\in\Pi} V^{\pi}(s) \tag{4.5}$$

4.2.5 Fonction action-valeur

On introduit une fonction aussi intéressante : la fonction action-valeur (Q-value) qui définit le retour attendu en partant de l'état s, en émettant l'action a puis en suivant la politique π par la suite $Q^{\pi}(s,a)$: $\mathcal{S}\times A \to \mathbb{R}$ définie comme suit :

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k}|s_t = s, a_t = a, \pi\right] \tag{4.6}$$

De même façon , on obtient l'équation de Belleman pour Q^{π} :

Comme pour la fonction valeur, la fonction optimale action-valeur Q^* peut également être défini par :

$$Q^*(s,a) = \max_{\pi\in\Pi} Q^{\pi}(s,a) \tag{4.7}$$

La particularité de la fonction action-valeur Q par rapport à la fonction valeur V est que la politique optimale peut être obtenue directement à partir de $Q^*(s,a)$:

$$\pi^*(s) = \underset{a\in\mathcal{A}}{\mathrm{argmax}} Q^*(s,a) \tag{4.8}$$

La fonction valeur optimale V^* est la récompense escomptée attendue dans un état donné s tout en suivant la politique π par la suite. La valeur optimale $Q^*(s,a)$ est le retour actualisé attendu dans un état donné s et pour une action donnée a tout en suivant la politique π par la suite.

On cherche à exprimer récursivement la fonction valeur optimale V^* , on a :

$$
\begin{aligned}
V^{\star}(s) &= \max_{a\in\mathcal{A}(s)} Q^{\pi^{\star}}(s,a) \\
&= \max_{a\in\mathcal{A}(s)} E_{\pi^*}\{R_t|s_t = s, a_t = a\} \\
&= \max_{a\in\mathcal{A}(s)} E_{\pi^*}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s, a_t = a\right\} \\
&= \max_{a\in\mathcal{A}(s)} E_{\pi^*}\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t = s, a_t = a\right\} \\
&= \max_{a\in\mathcal{A}(s)} E\{r_{t+1} + \gamma V^{\star}(s_{t+1})\,|s_t = s, a_t = a\} \\
&= \max_{a\in\mathcal{A}(s)}\left[\mathcal{R}_{sa} + \gamma\sum_{s'\in\mathcal{S}^+}\mathcal{T}_{sa}^{s'}V^{\star}(s')\right]
\end{aligned}
\tag{4.9}
$$

De même façon on obtient pour la fonction action-valeur optimale :

$$Q^{\star}(s,a) = \mathcal{R}_{sa} + \gamma\max_{a\in\mathcal{A}(s)}\sum_{s'\in\mathcal{S}^+}\mathcal{T}_{sa}^{s'}Q^{\star}(s',a') \tag{4.10}$$

Il est également possible de définir la fonction avantage :

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s) \tag{4.11}$$

qui décrit la qualité de l'action a par rapport au rendement attendu lors de suivi direct de la politique π.

4.2.6 Rétropropagation de la valeur

D'après l'équation de Bellman , la valeur est retropropagée vers les états précédents : [fig]

4.3 Différents composants pour apprendre une politique

Un agent RL comprend un ou plusieurs des composants suivants :

— une représentation d'une fonction de valeur qui fournit une prédiction de la qualité de chaque état ou de chaque paire état/action,

— une représentation directe de la politique $\pi(s)$ ou $\pi(s, a)$, ou

— un modèle de l'environnement (fonction de transition estimée et fonction de récompense estimée) associé à un algorithme de planification.

Les deux premiers composants sont liés à ce que l'on appelle *model-free* RL. Lorsque ce dernier composant est utilisé, l'algorithme est appelé *model-based* RL. Un schéma avec toutes les approches possibles est fourni à la figure [].

Pour la plupart des problèmes abordant la complexité du monde réel, l'espace d'états est de grande dimension (et peut-être continu). Pour apprendre une estimation du modèle, de la fonction de valeur ou de la politique, les algorithmes RL présentent deux principaux avantages : s'appuyer sur l'apprentissage en profondeur :

— Les réseaux de neurones sont bien adaptés au traitement des entrées sensorielles de grande dimension (telles que les séries chronologiques, etc.) et, dans la pratique, ils ne nécessitent pas une augmentation exponentielle des données lors de l'ajout de dimensions supplémentaires à l'espace d'état ou d'action.

— En outre, ils peuvent être formés progressivement et utiliser des échantillons supplémentaires obtenus au fur et à mesure de l'apprentissage.

Un schéma avec toutes les approches possibles est fourni dans la figure [fig]

4.4 Différentes configurations pour apprendre une politique à partir de données

4.4.1 Offline & online learning

L'apprentissage d'une tâche de décision séquentielle apparaît dans deux cas : (i) dans le cas d'apprentissage hors ligne où seules des données limitées sur un environnement donné est disponible et (ii) dans un cas d'apprentissage en ligne où, parallèlement, l'agent acquiert progressivement de l'expérience environnement. Dans les deux cas, les principaux algorithmes d'apprentissage présentés dans [4.5]. La spécificité du paramètre de traitement par lots (batch setting) est que l'agent doit apprendre des données limitées sans possibilité d'interaction supplémentaire avec l'environnement. Dans ce cas, la notion de généralisation introduite au chapitre 7 est au centre des préoccupations. En mode en ligne, le problème d'apprentissage est plus complexe et l'apprentissage sans nécessiter une grande quantité de données (efficacité de l'échantillon) n'est pas uniquement influencé par la capacité de l'algorithme d'apprentissage à bien généraliser à partir de l'expérience limitée. En effet, l'agent a la possibilité de rassembler de l'expérience via une stratégie d'exploration / exploitation. En outre, il peut utiliser areplaymemorypour stocker son expérience afin de pouvoir la traiter à nouveau. La mémoire d'exploration et de rejeu sera abordée au chapitre 8). Dans les configurations par lot (batch) et en ligne, une considération supplémentaire est

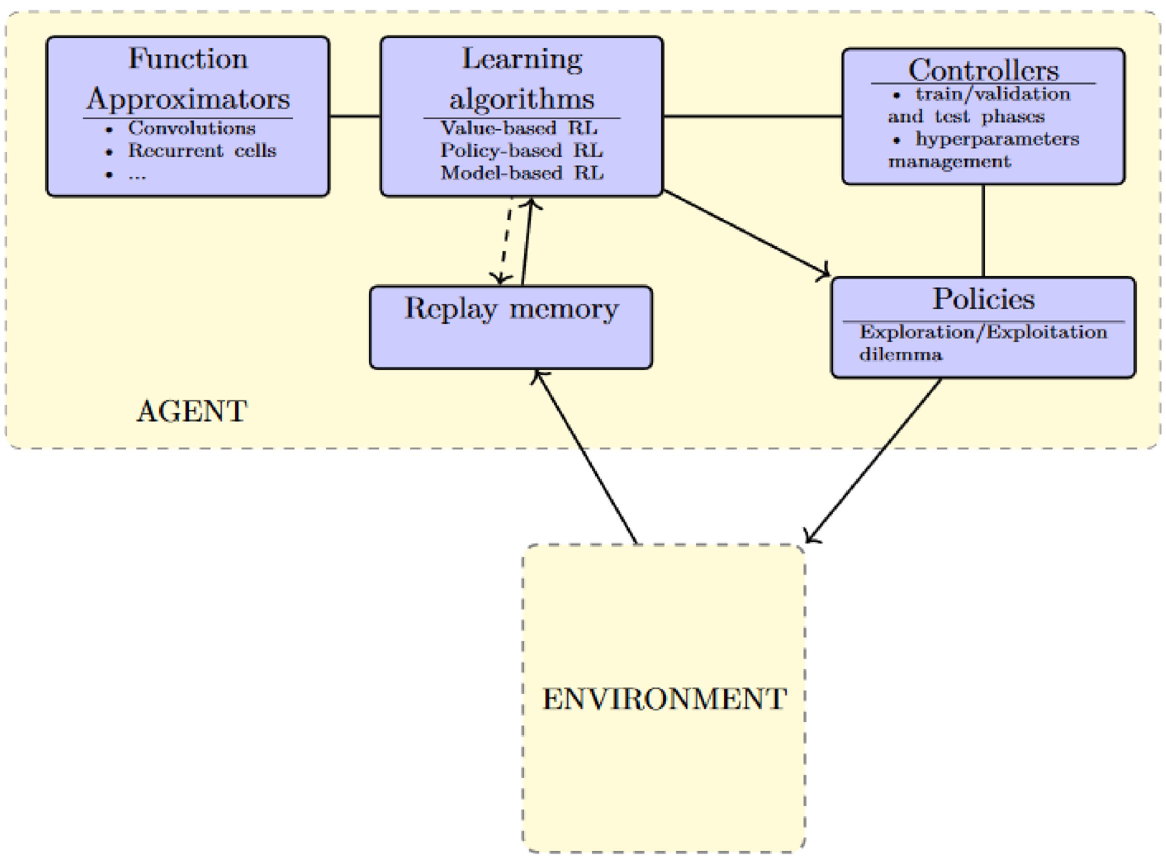


Schéma général des méthodes RL profond

également l'efficacité du calcul, qui dépend, entre autres choses, de l'efficacité d'un pas de descente de gradient donné. Tous ces éléments seront introduits avec plus de détails dans les chapitres suivants. La figure 4.2 présente un schéma général des différents éléments que l'on peut trouver dans la plupart des algorithmes RL profond.

4.4.2 Comparaison entre l'off-policy et l'on-policy learning

Selon Sutton et Barto, 2017, « Les méthodes on-policy tentent d'évaluer ou d'améliorer la politique utilisée pour prendre des décisions, alors que les méthodes off-policy évaluent ou améliorent une politique différente de celle utilisée pour générer les données». Dans les méthodes basées sur l'off-policy, l'apprentissage est simple lorsque vous utilisez des trajectoires qui ne sont pas nécessairement obtenues selon la stratégie actuelle, mais à partir d'une stratégie de comportement différente $\beta(s, a)$. Dans ces cas, la répétition d'expérience permet de réutiliser des échantillons provenant d'une stratégie de comportement différente. Au contraire, les méthodes basées sur l'on-policy introduisent généralement un biais lorsqu'elles sont utilisées avec un tampon de rejeu (replay buffer), car les trajectoires ne sont généralement pas obtenues uniquement avec la politique actuelle π. Cela donne un avantage aux méthode off-policy, car elles permettent de tirer parti de toute expérience. En revanche, les méthodes on-policy introduiraient un biais lorsqu'on utilisait des trajectoires off-policy, si l'on n'y prêtait pas une attention particulière.

4.5 Méthodes basées sur la valeur pour deep RL

La classe d'algorithmes basée sur la valeur vise à construire une fonction de valeur, ce qui nous permet par la suite de définir une politique. Nous discutons ci-après l'un des algorithmes de valeur les plus simples et les plus répandus, l'algorithme Q-learning (Watkins, 1989) et sa variante, le Q-learning ajusté, qui utilise des approximateurs de fonctions paramétrés (Gordon, 1996). Nous abordons également les

principaux éléments de l'algorithme DQN (deep Q-network) (Mnihet al., 2015) qui a permis un contrôle de niveau surhumain lorsqu'il joue à des jeux ATARI à partir des pixels en utilisant des réseaux naturels comme approximateurs de fonctions. Nous passons ensuite en revue diverses améliorations de l'algorithme DQN et fournissons des ressources pour des détails supplémentaires. À la fin de ce chapitre et dans le chapitre suivant, nous discutons du lien ultime entre les méthodes basées sur la valeur et les méthodes basées sur les politiques.

4.5.1 Q-learning

La version de base de Q-learning conserve une table de correspondance de valeurs $Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k}|s_t = s, a_t = a, \pi\right]$ avec une entrée pour chaque paire état-action. Afin de mieux comprendre la fonction optimale de la valeur Q, l'algorithme Q-learning utilise l'équation de Bellman pour la fonction valeur Q (Bellman et Dreyfus, 1962) dont la solution unique est $Q^*(\mathcal{S},a)$:

$$Q^*(s,a) = (\mathcal{B}Q^*)(s,a) \tag{4.12}$$

où $\mathcal{B}$ est l'opérateur de Bellman mappant une fonction quelconque $K : \mathcal{S}\times\mathcal{A} \to \mathbb{R}$ dans une autre fonction $\mathcal{S}\times\mathcal{A} \to \mathbb{R}$ et est défini comme suit :

$$(\mathcal{B}K)(s,a) = \sum_{s'\in S} T(s,a,s')\left(R(s,a,s') + \gamma \max_{a'\in\mathcal{A}} K(s',a')\right) \tag{4.13}$$

L'opérateur Bellman est une contraction car il peut être démontré que pour toute paire de fonctions bornées $K, K' : S\times A \to \mathbb{R}$ (l'espace des fonctions bornées est un espace métrique complet) on a :

$$|(BK - BK')(s,a)| = \left|\gamma \sum_{s'\in S} T(s,a,s')\left(\max_{a'\in\mathcal{A}} K(s',a') - \max_{a'\in\mathcal{A}} K'(s',a')\right)\right| \leqslant \gamma \|K-K'\|_\infty \tag{4.14}$$

Donc,

$$\|BK - BK'\|_\infty \leqslant \gamma \|K-K'\|_\infty \tag{4.15}$$

Donc, d'après le théorème de point fixe de Banach ,le point fixe de l'opérateur Bellman B existe.

Dans la pratique, il existe une preuve générale de la convergence vers la fonction valeur optimale (Watkins et Dayan, 1992) dans les conditions suivantes :

— les paires état-action sont représentées discrètement, et

— toutes les actions sont échantillonnées de manière répétée dans tous les états (ce qui garantit une exploration suffisante, ne nécessitant donc pas d'accès au modèle de transition).

Cette méthode simple est souvent inapplicable en raison de l'espace d'action d'états de grande dimension (éventuellement continu). Dans ce contexte, une fonction de valeur paramétrée $Q(s,a;\theta)$ est nécessaire, où θ fait référence à certains paramètres qui définissent les Q-valeurs.

4.5.2 Q-learning ajusté

Les expériences sont rassemblées dans un jeu de données D donné sous la forme de tuples $< s,a,r,s' >$ où l'état lors du pas de temps suivant est tiré de $T(s,a,\cdot)$ et des récompenses données par $R(s,a,s')$. Dans l'apprentissage Q-learning ajusté (Gordon, 1996), l'algorithme commence par une initialisation aléatoire des valeurs $Q(s,a;\theta_0)$ où θ_0 se réfère aux paramètres initiaux (généralement telles que les valeurs Q initiales doivent être relativement proches de 0 afin d'éviter un apprentissage lent). Ensuite, une approximation des valeurs Q à la k ème itération $Q(s,a;\theta_k)$ est mise à jour en direction de la valeur cible :

$$Y_k^Q = r + \gamma \max_{a'\in\mathcal{A}} Q(s',a';\theta_k) \tag{4.16}$$

où θ_k fait référence à certains paramètres qui définissent les valeurs Q à la k-ème itération. Dans l'apprentissage neuronal ajusté (Neural fitted Q-learning : NFQ) (Riedmiller, 2005), l'état peut être fourni en tant qu'entrée au Q-réseau et une sortie différente est donnée pour chacune des actions possibles. Ceci fournit une structure efficace qui présente l'avantage d'obtenir le calcul de $\max_{a'\in A} Q(s',a';\theta_k)$ en un

seul passage en avant dans le réseau neuronal pour un donnée s'. Les Q-valeurs sont paramétrées avec un réseau de neurones $Q(s, a; \theta_k)$ où les paramètres θ_k sont mis à jour par descente de gradient stochastique (ou une variante) en minimisant la perte :

$$L_{DQN} = \left(Q(s, a; \theta_k) - Y_k^Q\right)^2 \tag{4.17}$$

Ainsi, la mise à jour de Q-learning revient à mettre à jour les paramètres :

$$\theta_{k+1} = \theta_k + \alpha \left(Y_k^Q - Q(s, a; \theta_k)\right) \nabla_{\theta_k} Q(s, a; \theta_k) \tag{4.18}$$

où α est le taux d'apprentissage. Notez que l'utilisation de la perte carrée n'est pas arbitraire. En effet, il garantit que $Q(s, a; \theta_k)$ tendra sans biais à la valeur attendue de la variable aléatoire Y_k^Q.He, il s'assure que $Q(s, a; \theta_k)$ tendra à $Q^*(\mathcal{S}, a)$ après plusieurs hypothèses dans l'hypothèse selon laquelle le réseau de neurones est bien adapté à la tâche et que l'expérience acquise dans le jeu de données D est suffisante (plus de détails à ce sujet au chapitre 7). Lors de la mise à jour des poids, on modifie également la cible. En raison des capacités de généralisation et d'extrapolation des réseaux de neurones, cette approche peut générer de grosses erreurs à différents endroits de l'espace d'état. Par conséquent, la propriété de cartographie de contraction du Bellman de l'opérateur dans l'équation 4.2 n'est pas suffisant pour garantir la convergence. Il est vérifié expérimentalement que ces erreurs peuvent se propager avec cette règle actualisée et que, par conséquent, la convergence peut être lente, voire instable (Baird, 1995 ; Tsitsiklis et Van Roy, 1997 ; Gordon, 1999 ; Riedmiller, 2005). L'utilisation d'approximateurs de fonction est un autre effet secondaire néfaste lié au fait que les valeurs Q ont tendance à être surestimées en raison de l'opérateur max (Van Hasseltet al., 2016). En raison des instabilités et du risque de surestimation, un soin particulier a été apporté pour assurer un apprentissage correct.

4.5.3 Deep Q-networks

En s'appuyant sur les idées de NFQ, l'algorithme deep Q-network (DQN) introduit par Mnihet et al. (2015) permet d'obtenir de fortes performances en mode en ligne pour une variété de jeux ATARI, directement en apprenant à partir des pixels. Il utilise deux heuristiques pour limiter les instabilités :

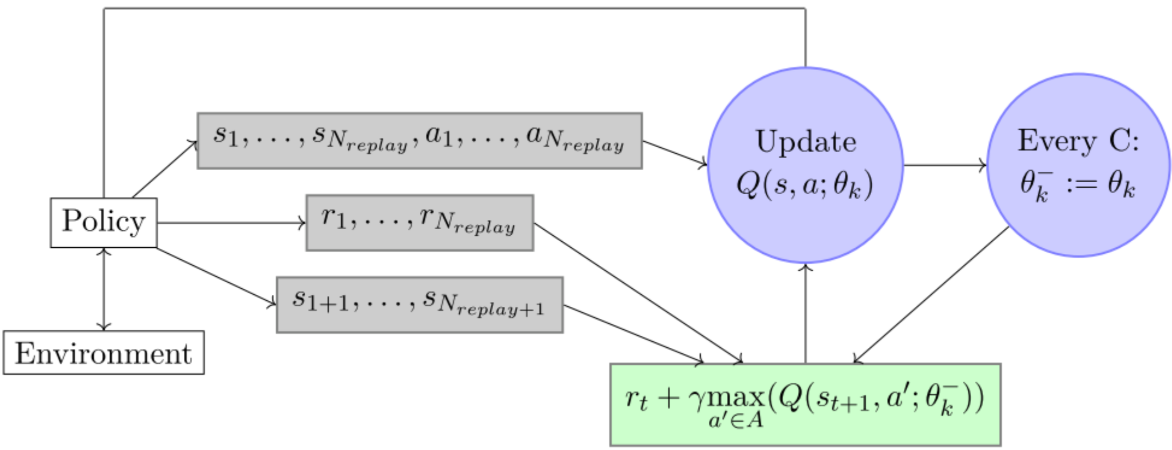
1. Le Q-réseau cible de l'équation 4.3 est remplacé par $Q\left(s', a'; \theta_k^-\right)$ où ses paramètres θ_k^- ne sont mis à jour que toutes les itérations $C \in \mathbb{N}$ avec l'affectation suivante :$\theta_k^- = \theta_k$. Cela empêche les instabilités de se propager rapidement et réduit le risque de divergence car les valeurs cibles Y_k^Q sont maintenues fixes pour les itérations C. L'idée des réseaux cibles peut être vue comme une instanciation d'un Q-learning ajusté, où chaque période entre les mises à jour du réseau cible correspond à une seule Q-itération ajustée.
2. En mode en ligne, la mémoire de rejeu (replay memory) (Lin, 1992) conserve tous les informations pour les derniers $N_{\text{replay}} \in \mathbb{N}$ pas de temps, où l'expérience est recueillie en suivant une politique ϵ-greedy [1] .

   Les mises à jour sont ensuite effectuées sur un ensemble de tuples (appelé mini-batch) sélectionnés de manière aléatoire dans la mémoire de lecture. Cette technique permet des mises à jour qui couvrent un large éventail d'espace d'états. De plus, une mise à jour de mini-lots a moins de variance par rapport à une simple mise à jour de tuples. Par conséquent, il offre la possibilité d'effectuer une mise à jour plus importante des paramètres, tout en permettant une parallélisation efficace de l'algorithme[4.3].

4.5.4 Perspective distributionnelle de RL

Dans cet partie, nous plaidons pour l'importance fondamentale de la distribution des valeurs : la distribution du retour aléatoire reçue par un agent d'apprentissage par renforcement. Cela contraste avec l'approche commune de l'apprentissage par renforcement qui modélise l'attente de ce retour. Bien qu'un corpus documentaire bien établi étudie la répartition des valeurs, il a jusqu'à présent été utilisé à des

1. Il effectue une action aléatoire avec probabilité ϵ et suit la politique donnée par $\mathrm{argmax}_{a \in \mathcal{A}} Q(s, a; \theta_k)$ avec probabilité $1 - \epsilon$



Esquisse de l'algorithme DQN

fins spécifiques telles que la mise en œuvre du risque comportement conscient. Nous commençons par des résultats théoriques à la fois dans les paramètres d'évaluation et de contrôle des politiques, ce qui révèle une instabilité distributive importante dans ces derniers. Ensuite, on va détailler la perspective distributionnelle pour concevoir un nouvel algorithme qui applique l'équation de Bellman à l'apprentissage des distributions de valeurs approximatives faite dans l'article [11].

Comme on l'a déja expliqué, l'un des principes majeurs de l'apprentissage par renforcement est qu'un agent doit viser à maximiser son utilité $\mathcal{Q}$ ou valeur attendue (Sutton Barto, 1998). L'équation de Bellman décrit succinctement cette valeur en termes de récompense attendue et de résultat attendu de la transition aléatoire $(x, a) \rightarrow (X', A')$:

$$Q(x, a) = \mathbb{E}R(x, a) + \gamma \mathbb{E}Q\left(X', A'\right) \tag{4.19}$$

Dans la suite , nous visons à aller au-delà de la notion de valeur et d'arguments en faveur d'une perspective distributionnelle de l'apprentissage par renforcement. Plus précisément, nous voulons étudier le retour aléatoire $\mathcal{Z}$ plutot que so éspérence qui est la valeur $\mathcal{Q}$. Ce retour aléatoire est également décrit par une équation récursive, mais de nature distributive :

$$Z(x, a) \overset{D}{=} R(x, a) + \gamma Z\left(X', A'\right) \tag{4.20}$$

Notations

On note la norme L_p d'un vecteur aléatoire $U : \Omega \rightarrow \mathbb{R}^{\mathcal{X}}$ (ou $\mathbb{R}^{\mathcal{X} \times \mathcal{A}}$) , avec $1 \leqslant p \leqslant \infty$ par : $\|U\|_p := [\mathbb{E}\,[\|U(\omega)\|_p^p]]^{1/p}$. Pour $p = \infty$, on note $\|U\|_\infty = \operatorname{ess\,sup} \|U(\omega)\|_\infty$.On note aussi la fonction de répartition (f.d.r) de U par $F_U(y) := \Pr\{U \leqslant y\}$ et son inverse par $F_U^{-1}(q) := \inf\{y : F_U(y) \geqslant q\}$. Une équation distributionnelle $U :\overset{D}{=} V$. $U :\overset{D}{=} V$ indique que la variable aléatoire U est distribuée selon la même loi de V.Les équations de distribution ont été utilisées dans l'apprentissage par renforcement par Engel et al. (2005) ; Morimura et al. (2010a), entre autres, et dans la recherche opérationnelle de White (1988).

La métrique de Wasserstein

L'outil principal de notre analyse est la métrique de Wasserstein d_p entre les fonctions de distribution cumulatives (voir par exemple Bickel Freedman, 1981, où il s'appelle la métrique de Mallows). Pour F,G deux f.d.r sur les réels, il est défini comme :

$$d_p(F, G) := \inf_{U,V} \|U - V\|_p \tag{4.21}$$

où l'inmum est pris sur toutes les paires de variables aléatoires (U, V) avec les distributions cumulatives respectives F et G. L'inmum est atteint par l'inverse f.d.r transformation d'une variable aléatoire U uniformément répartie sur $[0, 1]$:

$$d_p(F, G) = \left\| F^{-1}(\mathcal{U}) - G^{-1}(\mathcal{U}) \right\|_p \tag{4.22}$$

Pour $p < \infty$, ceci est plus explicitement écrit comme :

$$d_p(F, G) = \left(\int_0^1 \left| F^{-1}(u) - G^{-1}(u) \right|^p du \right)^{1/p} \tag{4.23}$$

Étant donné deux variables aléatoires U, V avec f.d.r F_U, F_V, nous écrirai :

$$d_p(U, V) := d_p(F_U, F_V) \tag{4.24}$$

Nous trouverons commode de concilier les variables aléatoires considérées avec leurs versions sous inf, en écrivant :

$$d_p(U, V) = \inf_{U,V} \|U - V\|_p \tag{4.25}$$

chaque fois sans ambiguïté ; Nous croyons que la plus grande lisibilité justifie l'inexactitude technique. Enfin, nous étendons cette métrique aux vecteurs de variables aléatoires, telles que les distributions de valeurs, en utilisant la norme L_p correspondante.

La métrique d_p a les propriétés suivantes :

$$\begin{aligned} d_p(aU, aV) &\leqslant |a| d_p(U, V) \\ d_p(A + U, A + V) &\leqslant d_p(U, V) \\ d_p(AU, AV) &\leqslant \|A\|_p d_p(U, V) \end{aligned} \tag{4.26}$$

Nous aurons besoin de la propriété supplémentaire suivante dans la suite, qui ne fait aucune hypothèse d'indépendance sur ses variables.

Lemme 4.5.1. *Soit $A_1, A_2, \ldots$ des variables aléatoires décrivant une partition de Ω, c'est-à-dire $A_i(\omega) \in \{0, 1\}$ et pour tout ω, il existe exactement un A_i avec $A_i(\omega) = 1$. Soit U, V deux variables aléatoires.Alors :*

$$d_p(U, V) \leqslant \sum_i d_p(A_i U, A_i V) \tag{4.27}$$

Démonstration. Nous allons donner la preuve pour $p < \infty$,notant qu'il en va de même pour $p = \infty$.Soit $Y_i :\overset{D}{=} A_i U$ et $Z_i :\overset{D}{=} A_i V$.D'abord noter que :

$$\begin{aligned} d_p^p(A_i U, A_i V) &= \inf_{Y_i, Z_i} \mathbb{E}\left[|Y_i - Z_i|^p\right] \\ &= \inf_{Y_i, Z_i} \mathbb{E}\left[\mathbb{E}\left[|Y_i - Z_i|^p \,|A_i\right]\right] \end{aligned} \tag{4.28}$$

Maintenant $|A_i U - A_i V|^p = 0$ chaque fois que $A_i = 0$. Il s'ensuit que nous pouvons choisir Y_i, Z_i pour que aussi $|Y_i - Z_i|^p = 0$ à chaque fois que $A_i = 0$, sans augmenter la norme . Par conséquent :

$$d_p^p(A_i U, A_i V) = \qquad \inf\nolimits_{Y_i, Z_i} \S e \Pr\{A_i = 1\} \mathbb{E}\left[|Y_i - Z_i|^p \,|A_i = 1\right] \tag{4.29}$$

Ensuite,

$$\inf_{U,V} \sum_i \Pr\{A_i = 1\} \mathbb{E}\left[|A_i U - A_i V|^p \,|A_i = 1\right] \leqslant \inf_{Y_1, V_2, \ldots, Z_1, Z_2, \ldots} \sum_i \Pr\{A_i = 1\} \mathbb{E}\left[|Y_i - Z_i|^p \,|A_i = 1\right] \tag{4.30}$$

Plus précisément, le membre de gauche de l'équation est un infinimum sur toutes les variables aléatoires dont les distributions cumulatives sont F_U et F_V, respectivement, tandis que le côté droit est un in

finimum sur les suites de variables aléatoires $Y_1, Y_2, \ldots$ et $Z_1, Z_2, \ldots$ dont les distributions cumulatives sont $F_{A_i U}, F_{A_i V}$, respectivement. Pour prouver cette limite supérieure, considérons la f.d.r de U :

$$\begin{aligned} F_U(y) &= \Pr\{U \leqslant y\} \\ &= \sum_i \Pr\{A_i = 1\} \Pr\{U \leqslant y | A_i = 1\} \\ &= \sum_i \Pr\{A_i = 1\} \Pr\{A_i U \leqslant y | A_i = 1\} \end{aligned} \tag{4.31}$$

D'autre part, la f.d.r de $Y_i \stackrel{D}{=} A_i U$ est :

$$\begin{aligned} F_{A_i U}(y) = &\Pr\{A_i = 1\} \Pr\{A_i U \leqslant y | A_i = 1\} + \Pr\{A_i = 0\} \Pr\{A_i U \leqslant y | A_i = 0\} \\ = &\Pr\{A_i = 1\} \Pr\{A_i U \leqslant y | A_i = 1\} + \Pr\{A_i = 0\} \mathbb{I}[y \geqslant 0] \end{aligned} \tag{4.32}$$

Ensuite ,

$$\begin{aligned} & d_p^p(U, V) = \inf_{U,V} \|U - V\|_p \\ & = \inf_{U,V} \mathbb{E}\left[|U - V|^p\right] \\ & \stackrel{(a)}{=} \inf_{U,V} \textstyle\sum_i \Pr\{A_i = 1\} \mathbb{E}\left[|U - V|^p \,|A_i = 1\right] \\ & = \inf_{U,V} \textstyle\sum_i \Pr\{A_i = 1\} \mathbb{E}\left[|A_i U - A_i V|^p \,|A_i = 1\right] \end{aligned} \tag{4.33}$$

où (a) suit parce que $A_1, A_2, \ldots$ est une partition.En utilisant 4.30, cela implique :

$$\begin{aligned} & d_p^p(U, V) \\ & = \inf_{U,V} \textstyle\sum_i \Pr\{A_i = 1\} \mathbb{E}\left[|A_i U - A_i V|^p \,|A_i = 1\right] \\ & \leqslant \inf_{Y_1, Z_2, \ldots} \textstyle\sum_i \Pr\{A_i = 1\} \mathbb{E}\left[|Y_i - Z_i|^p \,|A_i = 1\right] \\ & \stackrel{(b)}{=} \textstyle\sum_{i Y_i, Z_i} \Pr\{A_i = 1\} \mathbb{E}\left[|Y_i - Z_i|^p \,|A_i = 1\right] \\ & \stackrel{(c)}{=} \textstyle\sum_i d_p\left(A_i U, A_i V\right) \end{aligned} \tag{4.34}$$

parce que dans (b) les composants individuels de la somme sont indépendamment minimisé ; et (c) de (4.29). □

Soit $\mathcal{Z}$ l'espace des distributions de valeur avec borné des moments. Pour deux distributions de valeurs $Z_1, Z_2 \in \mathcal{Z}$, nous utiliserons une forme maximale de la métrique de Wasserstein :

$$\bar{d}_p\left(Z_1, Z_2\right) := \sup_{x,a} d_p\left(Z_1(x, a), Z_2(x, a)\right) \tag{4.35}$$

Nous utiliserons $\bar{d}_p$ pour établir la convergence des opérateurs Bellman de distribution.

Lemme 4.5.2. $\bar{d}_p$ *est une métrique sur les distributions de valeur.*

Démonstration. La seule propriété non triviale est l'inégalité triangulaire. Pour toute distribution de valeurs $Y \in \mathcal{Z}$, on a :

$$\begin{aligned} & \bar{d}_p\left(Z_1, Z_2\right) = \sup_{x,a} d_p\left(Z_1(x, a), Z_2(x, a)\right) \\ & \quad \stackrel{(a)}{\leqslant} \sup\left[d_p\left(Z_1(x, a), Y(x, a)\right) + d_p\left(Y(x, a), Z_2(x, a)\right)\right] \\ & \quad \leqslant \sup_{x,a} d_p\left(Z_1(x, a), Y(x, a)\right) + \sup_{x,a} d_p\left(Y(x, a), Z_2(x, a)\right) \\ & = \bar{d}_p\left(Z_1, Y\right) + \bar{d}_p\left(Y, Z_2\right) \end{aligned} \tag{4.36}$$

où dans (a) nous avons utilisé l'inégalité du triangulaire de le distance d_p. □

Évaluation de la politique

Dans le cadre de l'évaluation de la politique (Sutton Barto, 1998), nous nous intéressons à la fonction de valeur V^π associée à une politique donnée π. L'analogue ici est la distribution de la valeur Z^π. Dans cette section, nous caractérisons Z^π et étudions le comportement de l'opérateur d'évaluation de politique T^π. Nous soulignons que Z^π décrit le caractère aléatoire intrinsèque des interactions de l'agent avec son environnement, plutôt qu'une mesure de l'incertitude concernant l'environnement lui-même.

Nous considérons la fonction de récompense comme un vecteur aléatoire $R \in \mathcal{Z}$ et définissons l'opérateur de transition $P^\pi : \mathcal{Z} \to \mathcal{Z}$:

$$\begin{aligned} P^\pi Z(x, a) &:\overset{D}{=} Z(X', A') \\ &X' \sim P(\cdot|x, a), A' \sim \pi(\cdot|X') \end{aligned} \tag{4.37}$$

où nous utilisons des majuscules pour souligner le caractère aléatoire du prochain couple action-état (X', A'). Nous définissons l'opérateur de distribution Bellman $\mathcal{T}^\pi : \mathcal{Z} \to \mathcal{Z}$:

$$\mathcal{T}^\pi Z(x, a) := R(x, a) + \gamma P^\pi Z(x, a) \tag{4.38}$$

Bien que $\mathcal{T}^\pi$ présente une ressemblance superficielle avec l'opérateur habituel de Bellman (2), il est fondamentalement différent. En particulier, trois sources d'aléatoire définissent la distribution du composé $\mathcal{T}^\pi Z$:

— Le caractère aléatoire de la récompense R,

— Le hasard dans la transition P^π, et

— La distribution de valeur d'état suivant $Z(X', A')$.

Contraction en d_p Considérons le processus $Z_{k+1} := \mathcal{T}^\pi Z_k$, en partant de quelque $Z_0 \in \mathcal{Z}$. On peut s'attendre à ce que l'espérance limite de $\{Z_k\}$ converge de manière exponentielle rapidement, comme d'habitude, vers Q^π. Comme nous le montrons maintenant, le processus converge dans un sens plus fort : $\mathcal{T}^\pi$ est une contraction dans $\bar{d}_p$, ce qui implique que tous les moments convergent également de manière exponentielle et rapide.

Lemme 4.5.3. $\mathcal{T}^\pi : \mathcal{Z} \to \mathcal{Z}$ *est une* $\gamma-$*contraction dans* $(\mathcal{Z}, \bar{d}_p)$.

En utilisant le lemme 4.5.3, nous concluons en utilisant le théorème du point fixe de Banach que $\mathcal{T}^\pi$ a un point fixe unique. Lors de l'inspection, ce point fixe doit être $\mathcal{Z}^\pi$ tel que défini précedement. Comme nous supposons que tous les moments sont bornés, il suffit de conclure que la suite $\{Z_k\}$ converge vers Z avec $\bar{d}_p$ pour $1 \leqslant p \leqslant \infty$. Pour conclure, nous remarquons que toutes les métriques de distribution ne sont pas égales ; Par exemple, Chung Sobel (1987) ont montré que $\mathcal{T}^\pi$n'est pas une contraction de la distance de variation totale. Des résultats similaires peuvent être obtenus pour la divergence de Kullback-Leibler et la distance de Kolmogorov.

Apprentissage distributionnel approximatif

Dans cette section, nous proposons un algorithme basé sur l'opérateur d'optimalité de distribution de Bellman. Cela nécessitera en particulier de choisir une distribution approximative. Bien que le cas gaussien ait été précédemment considéré (Morimura et al., 2010a ; Tamar et al., 2016), on va utiliser une riche classe de distributions paramétriques considéré par (https ://arxiv.org/abs/1707.06887).

Distribution paramétrique Nous allons modéliser la distribution des valeurs en utilisant une distribution discrète paramétrée par $N \in \mathbb{N}$ et $V_{MIN}, V_{MAX} \in \mathbb{R}$, et dont le support est l'ensemble des atomes $\{z_i = V_{MIN} + i\triangle z : 0 \leqslant < N\}, \Delta z := \frac{V_{MAX} - V_{MIN}}{N-1}$. En un sens, ces atomes sont les «retours canoniques» de notre distribution. Les probabilités atomiques sont données par un modèle paramétrique $\theta : \mathcal{X} \times \mathcal{A} \to \mathbb{R}^N$:

$$Z_\theta(x, a) = z_i \quad \text{w.p.} \quad p_i(x, a) := \frac{e^{\theta_i(x,a)}}{\sum_j e^{\theta_j(x,a)}} \tag{4.39}$$

La distribution discrète présente les avantages d'être hautement expressif et convivial sur le plan computationnel (voir par exemple Van den Oord et al., 2016).

Mise à jour projetée de Bellman L'utilisation d'une distribution discrète pose un problème : la mise à jour de Bellman $\mathcal{T}Z_\theta$ et la paramétrisation Z_θ ont presque toujours des appuis disjoints. Il semblerait naturel de minimiser la distance de Wasserstein (considéré comme une perte) entre $\mathcal{T}Z_\theta$ et Z_θ, ce qui est également commodément robuste aux écarts dans le support. Cependant, un deuxième problème empêche cela : dans la pratique, nous sommes généralement limités à l'apprentissage à partir de transitions d'échantillon, ce qui n'est pas possible avec la perte de Wasserstein.

Au lieu de cela, nous projetons l'exemple de mise à jour de Bellman $\hat{\mathcal{T}}Z_\theta$ sur le support de Z_θ (figure 1, algorithme 1), ce qui réduit efficacement la mise à jour de Bellman à la classification à plusieurs classes. Soit π la politique gourmande (greedy policy) relativement à $\mathbb{E}Z_\theta$. Étant donné un exemple de transition (x, a, r, x'), on calcule la mise à jour de Bellman $\hat{\mathcal{T}}z_j := r + \gamma z_j$ pour chaque atome z_j, puis on distribue sa probabilité $p_j(x', \pi(x'))$ à les voisins immédiats de $\hat{\mathcal{T}}z_j$. La i-ème composante de la mise à jour projetée $\Phi\hat{\mathcal{T}}Z_\theta(x, a)$ est :

$$\left(\Phi\hat{\mathcal{T}}Z_\theta(x,a)\right)_i = \sum_{j=0}^{N-1}\left[1 - \frac{\left|\left[\hat{\mathcal{T}}z_j\right]_{V_{\mathrm{MIN}}}^{V_{\mathrm{MAX}}} - z_i\right|}{\triangle z}\right]_0^1 p_j\left(x', \pi(x')\right) \tag{4.40}$$

où $[\cdot]_a^b$ limite son argument dans l'intervalle $[a, b]$. Comme d'habitude, nous considérons la distribution de l'état suivant comme paramétrée par un paramètre xé $\tilde{\theta}$. La perte d'échantillon $\mathcal{L}_{x,a}(\theta)$ est le terme d'entropie croisée (cross-entropy) de la divergence de KL :

$$D_{\mathrm{KL}}\left(\Phi\hat{\mathcal{T}}Z_{\tilde{\theta}}(x,a) \| Z_\theta(x,a)\right) \tag{4.41}$$

qui est facilement minimisé, par exemple en utilisant la descente de gradient. Nous appelons ce choix de distribution et de perte l'algorithme catégorique [algo]. Lorsque N = 2, une alternative simple à un paramètre est :

$$\Phi\hat{\mathcal{T}}Z_\theta(x,a) := \left[\mathbb{E}\left[\hat{\mathcal{T}}Z_\theta(x,a)\right] - V_{\mathrm{MMN}}\right) / \triangle z]_0^1 \tag{4.42}$$

Nous appelons cela l'algorithme de Bernoulli. Nous notons que, bien que ces algorithmes ne semblent pas liés à la métrique de Wasserstein, des travaux récents (Bellemare et al., 2017) suggèrent une connexion plus profonde. Voir annexe [].

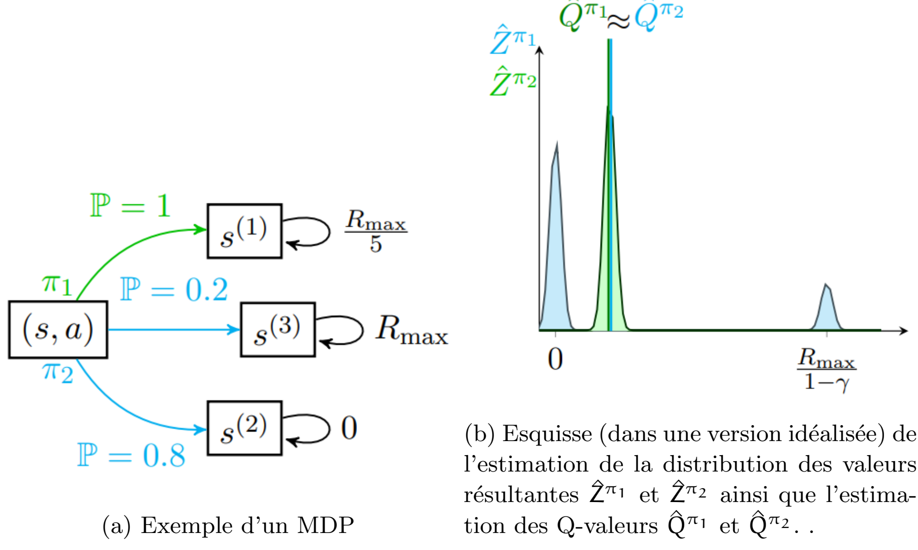
Data: Une transition $x_t, a_t, r_t, x_{t+1}, \gamma_t \in [0, 1]$
Result: Cross-entropy loss $:- \sum_i m_i \log p_i(x_t, a_t)$
$Q(x_{t+1}, a) := \sum_i z_i p_i(x_{t+1},$
$a^* \leftarrow \arg\max_a Q(x_{t+1}, a)$
$m_i = 0, \quad i \in 0, \ldots, N-1$
for $j \in 0, \ldots, N-1$ **do**
Calculer la projection de $\hat{\mathcal{T}}z_j$ *sur le support* $\{z_i\}$
$\hat{\mathcal{T}}z_j \leftarrow [r_t + \gamma_t z_j]_{V_{\mathrm{MAX}}}^{V_{\mathrm{MAX}}}$
$b_j \leftarrow \left(\hat{\mathcal{T}}z_j - V_{\mathrm{MIN}}\right) / \Delta z$ *avec* $b_j \in [0, N-1]$
$l \leftarrow \lfloor b_j \rfloor, u \leftarrow \lceil b_j \rceil$
Distribution de la probabilité de $\hat{\mathcal{T}}z_j$
$m_l \leftarrow m_l + p_j(x_{t+1}, a^*)(u - b_j)$
$m_u \leftarrow m_u + p_j(x_{t+1}, a^*)(b_j - l)$
end

Algorithm 4: L'algorithme catégorique de Bernoulli

Un exemple d'implémentation des algorithmes de RL distributionnel est dans l'annexe **??**.

Bilan : Cette approche présente les avantages suivants :

- Il est possible de mettre en œuvre un comportement conscient du risque (voir, par exemple, Morimura et al., 2010).
- Cela conduit à un apprentissage plus performant dans la pratique. Cela peut sembler surprenant puisque DQN et le DQN de distribution visent tous deux à maximiser le rendement attendu (comme illustré à la figure 4.3). L'un des éléments principaux est que la perspective de distribution fournit naturellement un ensemble de signaux d'apprentissage plus riche qu'une fonction de valeur scalaire.



(a) Exemple d'un MDP

(b) Esquisse (dans une version idéalisée) de l'estimation de la distribution des valeurs résultantes $\hat{Z}^{\pi_1}$ et $\hat{Z}^{\pi_2}$ ainsi que l'estimation des Q-valeurs $\hat{Q}^{\pi_1}$ et $\hat{Q}^{\pi_2}$. .

Pour deux politiques illustrées sur la figure (a), l'illustration sur la figure (b) donne la distribution de $Z^{(\pi)}(s, a)$ comparée à la valeur attendue $Q^{\pi}(s, a)$. Sur la figure de gauche, on peut voir que π_1 passe avec certitude à un état absorbant avec une récompense à chaque pas $\frac{R_{\max}}{5}$, tandis que π_2 se déplace avec une probabilité de 0,2 et 0,8 dans des états absorbants avec des récompenses à chaque pas respectivement $R_{\max}$ et 0. À partir de la paire (s, a), les politiques π_1 et π_2 ont le même rendement attendu mais des distributions de valeurs différentes.

4.5.5 Multi-step learning

Dans DQN, la valeur cible utilisée pour mettre à jour les paramètres du réseau Q (donnée dans l'équation 4.16) est estimée comme étant la somme de la récompense immédiate et une contribution des étapes suivantes de la déclaration. Cette contribution est estimée sur la base de sa propre estimation de valeur au prochain pas de temps. Pour cette raison, on dit que l'algorithme d'apprentissage est bootstrap puisqu'il utilise ses propres estimations de valeur de manière récursive (Sutton, 1988). Cette méthode d'estimation d'une valeur cible n'est pas la seule possibilité. Les méthodes sans amorçage (Non-bootstrapping) apprennent directement des déclarations (Monte Carlo) et une solution intermédiaire consiste à utiliser une cible à plusieurs étapes (Sutton, 1988 ; Watkins, 1989 ; Peng et Williams, 1994 ; Singh et Sutton, 1996). Une telle variante dans le cas de DQN peut être obtenue en utilisant la valeur cible à n échelons donnée par :

$$Y_k^{Q,n} = \sum_{t=0}^{n-1} \gamma^t r_t + \gamma^n \max_{a' \in A} Q(s_n, a'; \theta_k) \tag{4.43}$$

où $(s_0, a_0, r_0, \cdots, s_{n-1}, a_{n-1}, r_{n-1}, s_n)$ est toute trajectoire de $n+1$ pas de temps avec $s = s_0$ et $a = a_0$. Une combinaison de différentes cibles multi-étapes peut également être utilisée :

$$Y_k^{Q,n} = \sum_{i=0}^{n-1} \lambda_i \left(\sum_{t=0}^{i} \gamma^t r_t + \gamma^{i+1} \max_{a' \in A} Q(s_{i+1}, a'; \theta_k) \right) \tag{4.44}$$

avec $\sum_{i=0}^{n-1} \lambda_i = 1$. Dans la méthode appelée $TD(\lambda)$ (Sutton, 1988),$n \to \infty$ et λ_i suivent une loi géométrique : $\lambda_i \propto \lambda^i$ où 0 $0 \leqslant \lambda \leqslant 1$. Vous trouverez un exemple d'implémentation de cette méthode dans A.

Chapitre 5

L'apprentissage profond

5.1 Approche

L'apprentissage en profondeur repose sur une fonction $f : \mathcal{X} \to \mathcal{Y}$ paramétrée par $\theta \in \mathbb{R}^{n_\theta}$ ($n_\theta \in \mathbb{N}$) :

$$y = f(x, \theta) \tag{5.1}$$

Un réseau neuronal profond est caractérisé par une succession de multiples couches de traitement. Chaque couche consiste en une transformation non linéaire et la séquence de ces transformations conduit à l'apprentissage de différents niveaux d'abstraction (Erhan et al., 2009 ; Olah et al., 2017).

Tout d'abord, décrivons un réseau de neurones très simple avec une couche cachée entièrement connectée (fig 5.1). La première couche reçoit les valeurs d'entrée (c'est-à-dire les entités en entrée) x sous la forme d'un vecteur colonne de taille n_x . Les valeurs de la couche cachée suivante sont une transformation de ces valeurs par une fonction paramétrique non linéaire, qui est une multiplication par une matrice W_1 de taille $n_h \times n_x$,plus un terme de biais b_1 de taille n_h, suivi d'une transformation non linéaire :

$$h = A\left(W_1 \cdot x + b_1\right) \tag{5.2}$$

où A est la fonction d'activation.Cette fonction d'activation non linéaire est ce qui rend la transformation au niveau de chaque couche non linéaire, ce qui fournit finalement l'expressivité du réseau de neurones. Les choix les plus populaires de la fonction d'activation sont [fig] :

- la tangente hyperbolique : $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$. Elle sert principalement à la classification entre deux classes.
- la sigmoïde (ou fonction logistique) : $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$.La principale raison pour laquelle nous utilisons cette fonction est à valeurs dans $[0; 1]$. Par conséquent, il est particulièrement utilisé pour les modèles dans lesquels nous devons prédire la probabilité en tant que sortie.
- softmax 2.3 : est une fonction d'activation logistique plus généralisée utilisée pour la classification multiclass.
- rectified linear unit (ReLU) : $f(x) = x^+ = \max(0, x)$. Elle est actuellement la fonction d'activation la plus utilisée dans le monde. Depuis, elle est utilisée dans presque tous les réseaux de neurones convolutifs ou d'apprentissage en profondeur.
- softplus : $f(x) = \log\left(1 + e^x\right)$

La couche cachée h peut à son tour être transformé en d'autres ensembles de valeurs jusqu'à la dernière transformation fournissant les valeurs de sortie y. Dans ce cas :

$$y = \left(W_2 \cdot h + b_2\right) \tag{5.3}$$

Exemple de réseau de neurones avec une couche cachée.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Fonctions d'activations

Toutes ces couches sont formées pour minimiser l'erreur empirique $I_S[f]$. La méthode la plus courante d'optimisation des paramètres d'un réseau neuronal est basée sur la descente de gradient via l'algorithme de rétro-propagation (Rumelhart et al., 1988). Dans le cas le plus simple, à chaque itération, l'algorithme modifie ses paramètres internes θ afin de les adapter à la fonction souhaitée :

$$\theta \leftarrow \theta - \alpha \nabla_\theta I_S[f] \tag{5.4}$$

où α est le taux d'apprentissage. Je me base pricipalement dans les 2 suivantes sections sur [12].

5.2 Optimalité globale en apprentissage profond

Cette section étudie le problème de l'apprentissage des paramètres $W = \{W^k\}_{k=1}^K$ d'un réseau profond de N exemples d'apprentissage (X, Y). Notons $X \in \mathbb{R}^{N \times D}$ les données d'entrée avec D dimension de chaque ligne (par exemple, une image en niveaux de gris avec D pixels). Soit $W^k \in \mathbb{R}^{d_{k-1} \times d_k}$ une matrice représentant une transformation linéaire appliquée à la sortie de la couche $k-1$,$X_{k-1} \in \mathbb{R}^{N \times d_{k-1}}$, pour obtenir une d_k-représentation $X_{k-1}W^k$ à la couche k. Par exemple, chaque colonne de W^k pourrait représenter une convolution avec un filtre (comme dans les réseaux de neurones à convolution) ou l'application d'un classificateur linéaire (comme dans des réseaux entièrement connectés). Soit ψ_k : $\mathbb{R} \to \mathbb{R}$ une foction d'activation non linéraire (voir ...).On l'applique à chaque couche tel que $X_k = \psi_k\left(X_{k-1}W^k\right)$.La sortie X_k du réseau est donnée par :

$$\Phi\left(X, W^1, \ldots, W^K\right) = \psi_K\left(\psi_{K-1}(\cdots\ \psi_2\left(\psi_1\left(XW^1\right)W^2\right)\cdots W^{K-1})W^K\right) \tag{5.5}$$

Dans un cadre de classification, chaque ligne de $X \in \mathbb{R}^{N \times D}$ désigne un point de données dans $\mathbb{R}^D$ et chaque ligne de $Y \in \{0,1\}^{N \times C}$ indique l'appartenance de chaque point de données à une de C classes, c'est-à-dire $Y_{jc} = 1$ si la j-ième rangée de X appartient à la classe $c \in \{1, \ldots, C\}$ et $Y_{jc} = 0$ sinon. Dans une configuration de régression, les lignes de $Y \in \mathbb{R}^{N \times C}$ désignent les variables dépendantes des lignes de X. Le problème de l'apprentissage des pondérations du réseau W est formulé comme le problème d'optimisation suivant :

$$\min_{\{W^k\}_{k=1}^K} \ell\left(Y, \Phi\left(X, W^1, \ldots, W^K\right)\right) + \lambda \Theta\left(W^1, \ldots, W^K\right) \tag{5.6}$$

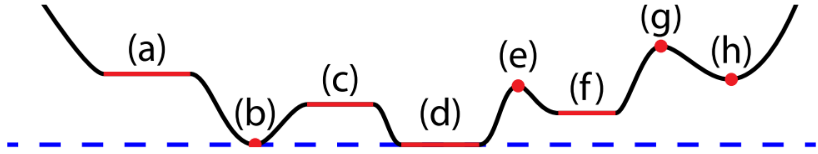
où $\ell(Y, \Phi)$ est une fonction de perte (loss) qui mesure la concordance entre la sortie réelle, Y, et la sortie prédite,$\Phi(X, W)$,Θ est une fonction de régularisation conçue pour empêcher le surajustement, par exemple : la décroissance du poids via la régularisation ℓ_2,$\Theta(W) = \sum_{k=1}^K \left\|W^k\right\|_F^2$, et $\lambda > 0$ est un paramètre d'équilibrage.

5.2.1 Le défi de la non convexité dans l'apprentissage en réseau de neurones

Un défi majeur dans l'entraînement au réseau neuronal est que le problème d'optimisation dans 5.6 est non convexe car, même si la perte $\ell(Y, \Phi)$ est typiquement une fonction convexe en Φ , par exemple, la perte au carré $\ell(Y, \Phi) = \|Y - \Phi\|_F^2$, $W \mapsto \Phi(X, W)$ est généralement une fonction non convexe due au produit des variables W^k et des non-linéarités des fonctions ψ_k . Cela pose des défis importants aux algorithmes d'optimisation existants (descente de gradient, descente de gradient stochastique, méthodes de minimisation en alternance, de descente en coordonnées de blocs, de propagation en arrière et de méthodes quasi-Newton). Cependant, pour les problèmes non convexes, l'ensemble des points critiques comprend non seulement les minima globaux, mais également les minima locaux, les maxima locaux, les points de selle et les plateaux de selle, comme l'illustre la Fig (5.3).

Par conséquent, la non convexité du Le problème laisse le modèle un peu mal posé en ce sens que ce n'est pas seulement la formulation du modèle qui importe, mais aussi les détails de mise en œuvre, tels que la façon dont le modèle est initialisé et les particularités de l'algorithme d'optimisation, qui peuvent avoir un impact significatif sur les performances du modèle.

Pour résoudre le problème de la non convexité, une stratégie courante utilisée dans l'apprentissage en profondeur consiste à initialiser les poids du réseau de manière aléatoire, à mettre à jour ces poids à l'aide de la descente locale, à vérifier si l'erreur de formation diminue suffisamment rapidement et,



Exemple de points critiques d'une fonction non convexe (indiqués en rouge). (a, c) Plateaux. (b, d) minima globaux. (e, g) maxima locaux. (f, h) Minimums locaux.

dans le cas contraire, à choisir une autre initialisation. En pratique, il a été observé que si la taille du réseau était suffisamment grande, cette stratégie pouvait conduire à des solutions très différentes pour les pondérations du réseau, qui donnaient presque les mêmes valeurs objectives et les mêmes performances de classification [22]. Il a également été observé que lorsque la taille du réseau est suffisante et que la fonction ReLU est choisie, de nombreux poids sont nuls, un phénomène appelé neurones morts (dead neurons) et la performance de la classification s'améliore considérablement [37], [38], [39], [40]. Bien que ceci suggère empiriquement que lorsque la taille du réseau est suffisante et que les non-linéarités ReLU sont utilisées, tous les minima locaux puissent être globaux, il n'existe actuellement aucune théorie rigoureuse fournissant une explication mathématique précise à ces phénomènes observés expérimentalement. [https ://arxiv.org/abs/1712.04741]

5.3 Stabilité géométrique en apprentissage profond

Une question importante sur le chemin de la compréhension des modèles d'apprentissage profond consistent à caractériser mathématiquement son biais inductif ; c'est-à-dire, définir la classe de tâches de régression / classification pour lesquelles elles sont prédéfinies de manière à bien exécuter, ou du moins à obtenir de meilleures performances que les alternatives classiques. Dans le cas particulier des tâches de vision par ordinateur, les archictectures convolutionnelles fournissent un biais inductif fondamental à l'origine des modèles de vision par apprentissage en profondeur les plus réussis. Comme nous l'expliquons ensuite, la notion de stabilité géométrique fournit un cadre possible pour comprendre son succès.

Soit $\Omega = [0,1]^d \subset \mathbb{R}^d$ un compact de $\mathbb{R}^d$. Dans une tâche d'apprentissage supervisé, une fonction inconnue $f : L^2(\Omega) \to \mathcal{Y}$ est observée sur un ensemble d'entraînement $\mathcal{D}$:

$$\mathcal{D} := \left\{X_i \in L^2(\Omega), Y_i = f(X_i)\right\}_{i \in \mathcal{I}} \tag{5.7}$$

où $\mathcal{Y}$ est l'espace cible qui peut être considéré comme discret dans une configuration de classification standard ($C = |\mathcal{Y}|$ étant le nombre de classes), ou $\mathcal{Y} = \mathbb{R}^C$ dans une tâche de régression. Dans la grande majorité des tâches de vision par ordinateur et d'analyse de la parole, la fonction inconnue f satisfait généralement les hypothèses cruciales suivantes :

1. Stationnarité : Considérons un opérateur de translation

$$\mathcal{T}_v X(u) = X(u - v), \quad \forall\, u, v \in \Omega, \forall\, X \in L^2(\Omega) \tag{5.8}$$

   En fonction de la tâche, nous supposons que la fonction f est soit invariante, soit équivariante par rapport aux translations. Dans le premier cas, nous avons $f(\mathcal{T}_v X) = f(X)$ pour tout $X \in L^2(\Omega)$ et $v \in \Omega$. C'est généralement le cas dans les tâches de classification d'objets. Dans ce dernier cas, nous avons $f(\mathcal{T}_v X) = \mathcal{T}_v f(X)$,ce qui est bien défini lorsque la sortie du modèle est un espace dans lequel les translations peuvent agir (par exemple, en cas de problèmes de localisation d'objet, de segmentation sémantique ou d'estimation de mouvement). La définition de l'invariance ici ne doit pas être confondue avec la notion traditionnelle de systèmes invariants de translation dans le traitement du signal, qui correspond à l'équivariance de translation dans notre langue (car la sortie traduit chaque fois que l'entrée est translatée).

2. Déformations locales et la séparation à l'échelle : De même, une déformation $\mathcal{L}_\tau$ définit sur $L^2(\Omega)$ par $\mathcal{L}_\tau X(u) = X(u - \tau(u))$, où $\tau : \Omega \to \Omega$ est un champ vectoriel lisse. X (u) = X (u - (u)).

Les déformations peuvent modéliser des translations locales, des changements de point de vue, des rotations et des transpositions de fréquence [9]. La plupart des tâches étudiées dans la vision par ordinateur ne sont pas seulement invariantes / équivariantes en translation, mais surtout stables en ce qui concerne les déformations locales [52], [9]. Dans les tâches invariantes à la translation, nous avons :

$$|f(\mathcal{L}_\tau X) - f(X)| \approx \|\nabla\tau\| \tag{5.9}$$

pour tout X, τ, où $\|\nabla\tau\|$ mesure la régularité d'un champ de déformation donné. En d'autres termes, la quantité à prédire ne change pas beaucoup si l'image d'entrée est légèrement déformée. Dans les tâches qui sont équivariantes en translation, nous avons

$$|f(\mathcal{L}_\tau X) - \mathcal{L}_\tau f(X)| \approx \|\nabla\tau\| \tag{5.10}$$

Cette propriété est beaucoup plus forte que la stationnarité, car l'espace des déformations locales a une grande dimensionnalité - de l'ordre de $\mathbb{R}^D$ lorsque nous discrétisons des images avec D pixels, par opposition au groupe de translation d-dimensionnel qui n'a que d = 2 dimensions dans le cas d'images.

Les hypothèses (5.9) - (5.10) peuvent être exploitées pour se rapprocher de f à partir des entités $\Phi(X)$ qui réduisent progressivement la résolution spatiale. En effet, extraire, démoduler et sous-échantillonner les réponses de filtre localisées crée des résumés locaux insensibles aux traductions locales, mais cette perte de perte de sensibilité n'affecte pas notre capacité à approcher f, grâce à (5.9) - (5.10). Pour illustrer ce principe, notons

$$Z(a_1, a_2; \nu) = \text{Prob}\left(X(u) = a_1 \text{ et } X(u+\nu) = a_2\right) \tag{5.11}$$

la distribution conjointe de deux pixels d'image décalés de ν l'un de l'autre. En présence de dépendances statistiques à long terme, cette distribution conjointe ne sera séparable d'aucun v. Cependant, la stabilité de la déformation antérieure indique que $Z(a_1, a_2; \nu) \approx Z(a_1, a_2; \nu(1+\epsilon))$ pour les ϵ petits. En d'autres termes, alors que les dépendances à long terme existent bien dans la nature et sont essentielles à la reconnaissance des objets, elles peuvent être capturées et sous-échantillonnées à différentes échelles. Bien que ce principe de stabilité aux déformations locales ait été exploité dans la communauté de la vision par ordinateur dans des modèles autres que les CNN, par exemple, les modèles de pièces déformables [53], les CNN trouvent un bon équilibre en termes de pouvoir d'approximation, d'optimisation et d'invariance.

En effet, la stationnarité et la stabilité vis-à-vis des traductions locales sont toutes deux mises à profit dans les réseaux de neurones à convolution (CNN). Un CNN est constitué de plusieurs couches convolutives de la forme $\tilde{X} = C_W(X)$ agissant sur une entrée de $p-$dimension $X(u) = (X_1(u), \ldots, X_p(u))$ en appliquant une banque de filtres $W = (w_{l,l'}), l = 1, \ldots, q, l' = 1, \ldots, p$ et une fonction non-linéaire ponctuelle ψ :

$$\tilde{X}_l(u) = \psi\left(\sum_{l'=1}^{p} (X_{l'} \star w_{l,l'})(u)\right) \tag{5.12}$$

et en produisant une sortie q-dimensionnelle $\tilde{X}(u) = \left(\tilde{X}_1(u), \ldots, \tilde{X}_q(u)\right)$ souvent appelé cartes de caractéristiques (feature maps). Ici,

$$(X \star w)(u) = \int_\Omega X(u-u')\, w(u')\, du' \tag{5.13}$$

désigne la convolution standard. Selon la déformation locale préalable, les filtres W ont un support spatial compact. De plus, une couche de sous-échantillonnage (downsampling) ou de regroupement (pooling) $\tilde{X} = P(X)$ peut être utilisé, défini comme :

$$\tilde{X}_l(u) = P\left(\{X_l(u') : u' \in \mathcal{N}(u)\}\right), \quad l = 1, \ldots, q \tag{5.14}$$

où $\mathcal{N}(u) \subset \Omega$ est un voisinage autour de u et P est une fonction invariante de la permutation telle qu'un pooling moyen, énergétique ou maximal).

Un réseau de convolution est construit en composant plusieurs couches de convolution et éventuellement de regroupement, obtenant une représentation hiérarchique générique.

$$\Phi_{\boldsymbol{W}}(X) = (C_{W^{(K)}} \cdots P \cdots \circ C_{W^{(2)}} \circ C_{W^{(1)}})(X) \quad (5.15)$$

où $\boldsymbol{W} = \{W^{(1)}, \ldots, W^{(K)}\}$ est l'hyper-vecteur des paramètres de réseau (tous les coefficients de filtre). Un avantage clé des CNN expliquant leur succès dans de nombreuses tâches est que les a priori géométriques sur lesquels les CNN sont basées sur un échantillon de complexité qui évite la malédiction de la dimensionnalité. Grâce à la stationnarité et aux prieurs déformations locales, les opérateurs linéaires à chaque couche ont un nombre constant de paramètres, indépendamment de la taille d'entrée D (nombre de pixels dans une image). De plus, grâce au propriété hiérarchique multi-échelles, le nombre de couches augmente à un taux $\mathcal{O}(\log D)$, ce qui entraîne une complexité d'apprentissage totale de $\mathcal{O}(\log D)$ paramètres.

Enfin, récemment, des efforts ont été déployés pour étendre les priorités de stabilité géométrique aux données qui ne sont pas définies sur un domaine euclidien, où le groupe de translation n'est généralement pas défini. En particulier, les chercheurs exploitent la géométrie de graphes généraux via le spectre de graphes laplaciens et ses équivalents spatiaux ; voir [13] pour une enquête récente sur ces avancées.

5.4 Théorie basée sur la structure pour l'apprentissage profond

5.4.1 Structure des données dans un réseau de neurones

Un aspect important pour comprendre un meilleur apprentissage profond est la relation entre la structure des données et le réseau profond. Pour une analyse formelle, considérons le cas d'un réseau comportant des poids i.i.d.gaussiens, qui est une initialisation commune dans la formation des réseaux profonds. Des travaux récents [14] montrent que de tels réseaux avec des poids aléatoires préservent la structure métrique des données lors de leur propagation le long des couches, permettant ainsi une récupération stable des données d'origine à partir des entités calculées par le réseau - une propriété souvent rencontrée en réseaux profonds [15],[16].

Plus précisément, le travail de [14] montre que l'entrée dans le réseau peut être récupérée à partir des caractéristiques du réseau à une certaine couche si leur taille est proportionnelle à la dimension intrinsèque des données d'entrée. Ceci est similaire à la reconstruction de données à partir d'un petit nombre de projections aléatoires [17], [18]. Cependant, bien que les projections aléatoires préservent la distance euclidienne entre deux entrées jusqu'à une faible distorsion, chaque couche d'un réseau profond avec des poids aléatoires distord cette distance proportionnellement à l'angle entre les deux entrées : plus l'angle est petit, plus le retrait de la distance. Par conséquent, plus le réseau est profond, plus fort le retrait atteint. Notez que cela ne contredit pas le fait qu'il est possible de récupérer l'entrée à partir de la sortie. même lorsque des propriétés telles que l'éclairage, la pose et l'emplacement sont supprimées d'une image (dans une certaine mesure), la ressemblance avec l'image d'origine est toujours conservée. La projection aléatoire étant une stratégie d'échantillonnage universelle pour les données de faible dimension [17], [18], [19], les réseaux profonds avec pondérations aléatoires constituent un système universel qui sépare toutes les données (appartenant à un modèle de faible dimension) selon angles entre les points de données, où l'hypothèse générale est qu'il existe de grands angles entre différentes classes [20], [21]. Au fur et à mesure que la formation de la matrice de projection s'adapte afin de mieux préserver des distances spécifiques par rapport aux autres, la formation d'un réseau donne la priorité aux angles intra-classe par rapport aux inter-classes. Cette relation est évoquée par les techniques de preuve dans [14] et se manifeste empiriquement par l'observation des angles et des distances euclidiennes à la sortie des réseaux formés.

5.5 Etat de l'art

Dans les applications actuelles, de nombreux types de couches de réseaux neuronaux sont apparus au-delà des simples réseaux à action directe qui viennent d'être présentés. Chaque variante offre des

avantages spécifiques, en fonction de l'application (par exemple, un bon compromis entre biais et surajustement dans un environnement d'apprentissage supervisé). De plus, dans un réseau de neurones donné, un nombre arbitrairement grand de couches est possible, et la tendance ces dernières années est d'avoir un nombre toujours croissant de couches, avec plus de 100 tâches d'apprentissage supervisées (Szegedy et al. ., 2017). Nous décrivons simplement ici deux types de couches présentant un intérêt particulier dans deep RL (et dans de nombreuses autres tâches).

Les couches convolutives (LeCun, Bengio et al., 1995) sont particulièrement bien adaptées aux images et aux données séquentielles (voir Fig.), principalement en raison de leur propriété d'invariance de traduction. Les paramètres de la couche consistent en un ensemble de filtres (ou noyaux) pouvant être appris, qui ont un petit champ de réception et qui appliquent une opération de convolution à l'entrée, transmettant le résultat à la couche suivante. En conséquence, le réseau apprend les filtres qui s'activent lorsqu'il détecte certaines fonctionnalités spécifiques. En classification d'image,les premières couches apprennent à détecter les arêtes, les textures et les motifs ; les couches suivantes sont ensuite capables de détecter des parties d'objets et des objets entiers (Erhan et al., 2009 ; Olah et al., 2017). En fait, une couche convolutive est un type particulier de couche à anticipation, avec la spécificité que de nombreuses pondérations sont définies sur 0 (non pouvant être apprises) et que d'autres pondérations sont partagées.

Les couches récurrentes sont particulièrement bien adaptées aux données séquentielles (voir la figure). Plusieurs variantes offrent des avantages particuliers dans différents contextes. Un tel exemple est le réseau de mémoire à court terme (LSTM) (Hochreiter et Schmidhuber, 1997), capable de coder des informations à partir de séquences longues, contrairement à un réseau de neurones récurrent de base. Les machines de Turing neurales (MNT) (Graves et al., 2014) en sont un autre exemple. Dans de tels systèmes, une "mémoire externe" différenciable est utilisée pour déduire des dépendances même à plus long terme que les LSTM à faible dégradation.

Plusieurs autres architectures de réseaux de neurones spécifiques ont également été étudiées pour améliorer la généralisation dans l'apprentissage en profondeur. Par exemple, il est possible de concevoir une architecture de telle sorte qu'elle ne se concentre automatiquement que sur certaines parties des entrées avec un mécanisme appelé attention (Xu et al., 2015 ; Vaswani et al., 2017). D'autres approches visent à travailler avec des règles symboliques en apprenant à créer des programmes (Reed et De Freitas, 2015 ; Neelakantan et al., 2015 ; Johnson et al., 2017 ; Chen et al., 2017). Pour des informations sur des sujets tels que l'importance des normalisations d'entrée, les techniques d'initialisation du poids, les techniques de régularisation et les différentes variantes des techniques de descente de gradient, le lecteur peut consulter plusieurs revues sur le sujet (LeCun et al., 2015 ; Schmidhuber, 2015 ; Goodfellow et al., 2016) ainsi que des références y figurant.

Dans la suite, l'accent est mis sur l'apprentissage par renforcement, en particulier sur les méthodes permettant d'approcher les approximateurs des fonctions de réseaux neuronaux en profondeur. Ces méthodes permettent d'apprendre une grande variété de tâches de prise de décision séquentielles complexes directement à partir d'intrants riches en dimensions.

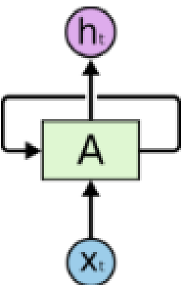
5.6 RNN standards

Je me base dans cette section sur l'article [22].

5.6.1 Introduction

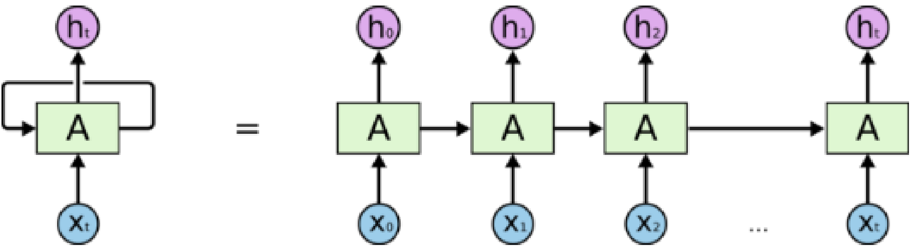
Les humains ne commencent pas à réfléchir à la seconde. En lisant ce rapport, vous comprenez chaque mot en fonction de votre compréhension des mots précédents. Vous n'éliminez pas tout et commencez à penser à nouveau. Vos pensées sont persévérantes. Les réseaux de neurones traditionnels ne peuvent pas faire cela, et cela semble être une lacune majeure. Par exemple, imaginons que vous souhaitiez classer le type d'événement qui se produit à chaque étape d'un film. On ignore comment un réseau de neurones traditionnel pourrait utiliser son raisonnement sur les événements précédents du film pour en informer les événements ultérieurs. Les réseaux de neurones récurrents répondent à ce problème. Ce sont des réseaux avec des boucles, permettant aux informations de persister.

Dans le diagramme ci-dessus (5.4) , un bloc de réseau neuronal, A : examine une entrée X_t et génère une valeur h_t. Une boucle permet aux informations d'être transmises d'une étape du réseau à la suivante.



Recurrent Neural Networks have loops.

Exemple d'un RNN : medium.com



An unrolled recurrent neural network.

Un réseau de neurones récurrent peut être considéré comme plusieurs copies du même réseau, chacune transmettant un message à un successeur. Considérez ce qui se passe si nous déroulons la boucle :

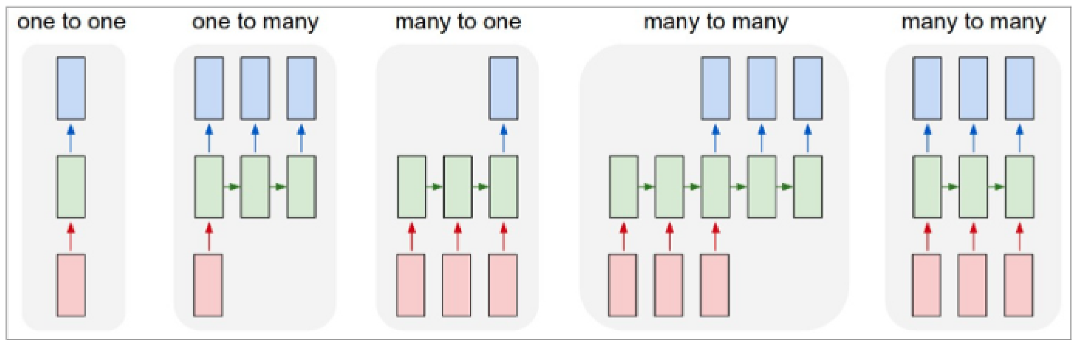
Cette nature en chaîne révèle que les réseaux de neurones récurrents sont intimement liés aux séquences et aux listes. C'est l'architecture naturelle du réseau de neurones à utiliser pour de telles données. Et ils sont certainement utilisés ! Au cours des dernières années, les applications RNN ont rencontré un succès incroyable pour une variété de problèmes : reconnaissance de la parole, modélisation du langage, traduction, sous-titrage d'images... La liste est longue. Bien que ce ne soit pas obligatoire, il serait bon que le lecteur comprenne ce que sont les WordVectors. Voici mon blog précédent sur Word2Vec, une technique permettant de créer des vecteurs Word.

5.6.2 Limitations et motivations

Une limitation flagrante des réseaux neuronaux Vanilla (et également des réseaux convolutifs) est que leur API est trop contrainte : ils acceptent un vecteur de taille fixe (par exemple, une image) et produisent un vecteur de taille fixe (par exemple, des probabilités de classes différentes).). De plus, ces modèles effectuent ce mappage en utilisant un nombre fixe d'étapes de calcul (par exemple, le nombre de couches dans le modèle). La raison principale pour laquelle les réseaux récurrents sont plus intéressants est qu'ils nous permettent d'opérer sur des séquences de vecteurs : séquences dans l'entrée, la sortie ou, dans le cas le plus général, les deux. Quelques exemples peuvent rendre cela plus concret (**??**) :

Chaque rectangle est un vecteur et les flèches représentent des fonctions (par exemple, une multiplication de matrice). Les vecteurs d'entrée sont en rouge, les vecteurs de sortie en bleu et les vecteurs verts indiquent l'état du RNN (plus de détails prochainement). De gauche à droite :

1. Mode "Vanilla" de traitement sans RNN, d'une entrée de taille fixe à une sortie de taille fixe (par exemple, classification d'image).
2. Sortie de séquence (par exemple, le sous-titrage d'image prend une image et génère une phrase de mots).



3. Entrée de séquence (par exemple, analyse de sentiment où une phrase donnée est classée comme exprimant un sentiment positif ou négatif)
4. Entrée de séquence et sortie de séquence (par exemple, traduction automatique : un RNN lit une phrase en anglais, puis en affiche une en français).
5. Entrée et sortie de séquence synchronisée (par exemple, classification vidéo où nous souhaitons étiqueter chaque image de la vidéo).

Notez que dans tous les cas, il n'existe pas de contraintes prédéfinies sur les séquences de longueurs, car la transformation récurrente (vert) est fixe et peut être appliquée autant de fois que nous le souhaitons.

Je me base dans la partie suivante sur [23].

5.6.3 Les racines de RNN

Dans cette section, nous allons dériver les réseaux de neurones récurrents (RNN) à partir d'équations différentielles. Même si les RNN sont exprimés sous forme d'équations aux différences, celles-ci sont indispensables à la modélisation des réseaux de neurones et continuent à avoir un impact profond sur la résolution de tâches pratiques de traitement de données à l'aide de méthodes d'apprentissage automatique. Les équations différentielles sont obtenues à partir des équations différentielles originales correspondantes par discrétisation des opérateurs différentiels agissant sur les fonctions sous-jacentes. S'appuyant sur les théories mathématiques établies à partir d'équations différentielles dans le domaine des données en temps continu permet souvent de mieux comprendre l'évolution des équations aux différences correspondantes . Soit $\vec{s}(t)$ la valeur du vecteur de signal d'état à d dimensions et considérons l'équation différentielle générale non linéaire non homogène du premier ordre, qui décrit l'évolution du signal d'état en fonction du temps,t :

$$\frac{d\vec{s}(t)}{dt} = \vec{f}(t) + \vec{\phi} \tag{5.16}$$

où $\vec{f}(t)$ est une fonction du temps à valeurs vectorielles d-dimensionnelle,$t \in \mathbb{R}^+$, et $\vec{\phi}$ est un vecteur constant de dimension d. Une forme canonique de $\vec{f}(t)$ est :

$$\vec{f}(t) = \vec{h}(\vec{s}(t), \vec{x}(t)) \tag{5.17}$$

où $\vec{x}(t)$ est le vecteur du signal d'entrée de dimension d et $\vec{h}(\vec{s}(t), \vec{x}(t))$ est une fonction à valeur vectorielle des arguments à valeur vectorielle. Le système résultant,

$$\frac{d\vec{s}(t)}{dt} = \vec{h}(\vec{s}(t), \vec{x}(t)) + \vec{\phi} \tag{5.18}$$

survient dans de nombreuses situations en physique, chimie, biologie et ingénierie. Dans certains cas, on commence avec s et x en tant que quantités entièrement «analogiques» (c'est-à-dire, fonctions non seulement du temps, t, mais aussi d'une autre variable continue indépendante,$\vec{\xi}$, désignant les coordonnées dans un espace multidimensionnel). En utilisant cette notation, l'intensité d'un signal vidéo d'entrée affiché sur un écran plat à 2 dimensions serait représentée par $x(\vec{\xi}, t)$ avec $\vec{\xi} \in \mathbb{R}^2$. L'échantillonnage de $x(\vec{\xi}, t)$ sur une grille uniforme à 2 dimensions convertit ce signal en représentation $x(\vec{i}, t)$, où $\vec{i}$ est un indice discret à 2 dimensions. Enfin, l'assemblage des valeurs de $x(\vec{i}, t)$ pour toutes les permutations

des composants de l'indice,$\vec{i}$, dans un vecteur colonne, produit $\vec{x}(t)$ comme initialement présenté dans l'équation 5.18 ci-dessus.

Une autre forme canonique de $\vec{f}(t)$ est :

$$\vec{f}(t) = \vec{a}(t) + \vec{b}(t) + \vec{c}(t) \tag{5.19}$$

dont les termes constitutifs,$\vec{a}(t), \vec{b}(t)$, et $\vec{c}(t)$, sont des fonctions à valeurs vectorielles d-dimensionnelles du temps t. Ils sont définis comme suit :

$$\begin{aligned}
\vec{a}(t) &= \sum_{k=0}^{K_s-1} \vec{a}_k \left(\vec{s}\left(t - \tau_s(k)\right)\right) \\
\vec{b}(t) &= \sum_{k=0}^{K_r-1} \vec{b}_k \left(\vec{r}\left(t - \tau_r(k)\right)\right) \\
\vec{r}\left(t - \tau_r(k)\right) &= G\left(\vec{s}\left(t - \tau_r(k)\right)\right) \\
\vec{c}(t) &= \sum_{k=0}^{K_x-1} \vec{c}_k \left(\vec{x}\left(t - \tau_x(k)\right)\right)
\end{aligned} \tag{5.20}$$

Notre objectif est d'encoder chaque mot en utilisant un vecteur dans $\mathbb{R}^d$, de sorte que les mots ayant une signification similaire soient proches.En raison de la disparition des gradients dans les RNN standards, les gradients ne se propagent pas bien à travers le réseau : impossible d'apprendre les dépendances à long terme. C'est pour cela on introduit les LSTM qui vont contournés ce problème parfaitement .

5.7 LSTM

Je me base dans cette section sur [24]

5.7.1 Introduction

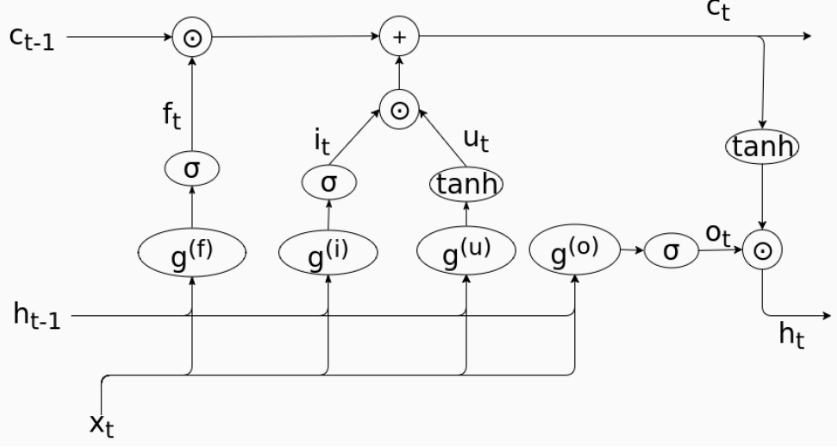
Long short-term memory est une architecture de réseau de neurones récurrents artificiels (RNN) utilisée dans le domaine de l'apprentissage en profondeur. Contrairement aux réseaux neuronaux à réaction standard, le LSTM dispose de connexions de retour qui en font un "ordinateur polyvalent" (c'est-à-dire qu'il peut calculer tout ce que peut une machine de Turing). Il peut non seulement traiter des points de données uniques (tels que des images), mais également des séquences complètes de données (telles que la parole ou la vidéo). Par exemple, LSTM est applicable à des tâches telles que la reconnaissance de l'écriture manuscrite connectée ou la reconnaissance vocale non segmentées . Bloomberg Business Week a écrit : "Ces pouvoirs font de LSTM le succès commercial le plus commercial de l'IA, utilisé pour tout, de la prévision des maladies à la composition musicale" .

Une unité LSTM commune est composée d'une cellule, d'une porte d'entrée, d'une porte de sortie et d'une porte d'oublie. La cellule se souvient des valeurs sur des intervalles de temps arbitraires et les trois portes régulent le flux d'informations entrant et sortant de la cellule.

Les réseaux LSTM sont bien adaptés à la classification, au traitement et à la prévision sur la base de données chronologiques, car il peut y avoir des décalages d'une durée inconnue entre les événements importants d'une série chronologique. Les LSTM ont été développés pour traiter les problèmes de gradient explosifs et évanescents que l'on peut rencontrer lors de la formation de RNN traditionnels. L'insensibilité relative à la longueur de l'espace est un avantage du LSTM par rapport aux RNN, aux modèles de Markov cachés et à d'autres méthodes d'apprentissage par séquence dans de nombreuses applications [citation requise].

5.7.2 Principe

En théorie, les RNN classiques (ou "à la vanilla") peuvent suivre les dépendances arbitraires à long terme dans les séquences d'entrée. Le problème des RNN vanillas est de nature informatique (ou pratique) : lors de la formation d'un RNN vanilla en back-propagation, les gradients qui sont rétro-propagés



LSTM Cell

peuvent "disparaître" (c'est-à-dire qu'ils peuvent tendre à zéro) ou "exploser" (c'est-à-dire qu'ils peuvent tendre vers l'infini), en raison des calculs impliqués dans le processus, qui utilisent des nombres à précision finie. Les RNN utilisant des unités LSTM résolvent partiellement le problème de gradient de fuite, car les unités LSTM permettent également aux gradients de circuler sans modification. Cependant, les réseaux LSTM peuvent toujours souffrir du problème de la dégradation du gradient. [Wikipedia]

5.7.3 Architecture

Il existe plusieurs architectures d'unités LSTM. Une architecture commune est composée d'une cellule (la partie mémoire de l'unité LSTM) et de trois "régulateurs", généralement appelés portes, du flux d'informations à l'intérieur de l'unité LSTM : une porte d'entrée, une porte de sortie et une porte d'oubli. Certaines variantes de l'unité LSTM n'ont pas une ou plusieurs de ces portes, voire d'autres. Par exemple, les unités récurrentes gated (GRU) n'ont pas de porte de sortie.

Intuitivement, la cellule est chargée de suivre les dépendances entre les éléments de la séquence d'entrée. La porte d'entrée contrôle dans quelle mesure une nouvelle valeur pénètre dans la cellule, la porte oublie contrôle dans quelle mesure une valeur reste dans la cellule et la porte de sortie détermine dans quelle mesure la valeur dans la cellule est utilisée pour calculer la sortie. activation de l'unité LSTM. La fonction d'activation des portes LSTM est souvent la fonction logistique.

Il existe des connexions dans et hors des portes du LSTM, dont certaines sont récurrentes. Le poids de ces connexions, qui doivent être apprises au cours de la formation, détermine le fonctionnement des portes.[Wikipedia]

On note pour la suite :

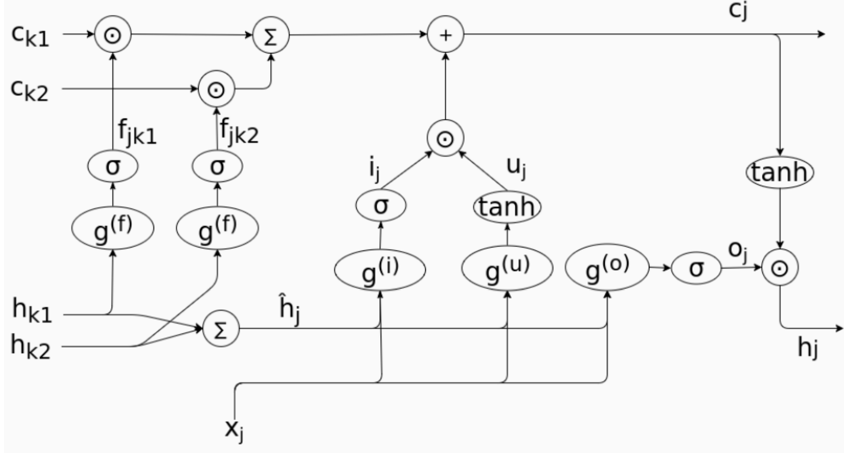
$$g^{n}\left(x_{t}, h_{t-1}\right) = W^{(n)}x_{t} + U^{(n)}h_{t-1} + b^{(n)}$$

5.7.4 Variantes

Les phrases ne sont pas une simple séquence linéaire.Notre objectif est de conserver le sens syntaxique de la phrase . Pour cela , il existe plusieurs variantes de LSTM pour gérer ce problème :

— Child-sum tree LSTM [5.6] : Somme sur tous les enfants d'un noeud : peut être utilisé pour n'importe quel nombre d'enfants. Ce modele permet de prendre en entrée des arbres d'analyse de phrases : alors que le modèle LSTM standard permet de propager des informations séquentiellement en conditionnant la cellule LTSM d'un mot x_t sur l'état du mot précédent x_{t-1}, l'unité de mot Child-Sum Tree-LSTM dépend des états de tous les mots enfants de ce mot.

  Mêmes vecteurs de déclenchement et mêmes unités de mémoire que la cellule LSTM, mais ces vecteurs dépendent des états de toutes les unités filles d'un mot.



Child-sum tree LSTM au noeud j avec les enfants k1 et k2

Pour un arbre et un noeud j, $C(j)$ est l'ensemble des noeuds enfants du noeud j. Les équations de transition sont alors les suivantes :

1. $\tilde{h}_j = \sum_{k \in C(j)} h_k$
2. une porte d'entrée (input gate) : $i_j = \sigma\left(W^i x_j + U^i \tilde{h}_j + b^i\right)$
3. une porte d'oubli (forgot gate) :$f_{jk} = \sigma\left(W^f x_j + U^f h_k + b^f\right)$
4. une porte de sortie (output gate) : $o_j = \sigma\left(W^o * x_j + U^o \tilde{h}_j + b^o\right)$
5. $u_j = \tanh\left(W^u x_j + U^u \tilde{h}_j + b^u\right)$
6. une cellule mémoire : $c_j = i_j \odot u_j + \sum_{k \in C(j)} f_{jk} \odot c_k$
7. l'état caché du noeud j : $h_j = o_j \odot \tanh(c_j)$

— N-ary tree LSTM [5.7] : Utilise différents paramètres pour chaque noeud : meilleure granularité, mais le nombre maximal d'enfants par noeud doit être fixé. La cellule de mémoire c_j et l'état caché h_j sont calculés comme suit :

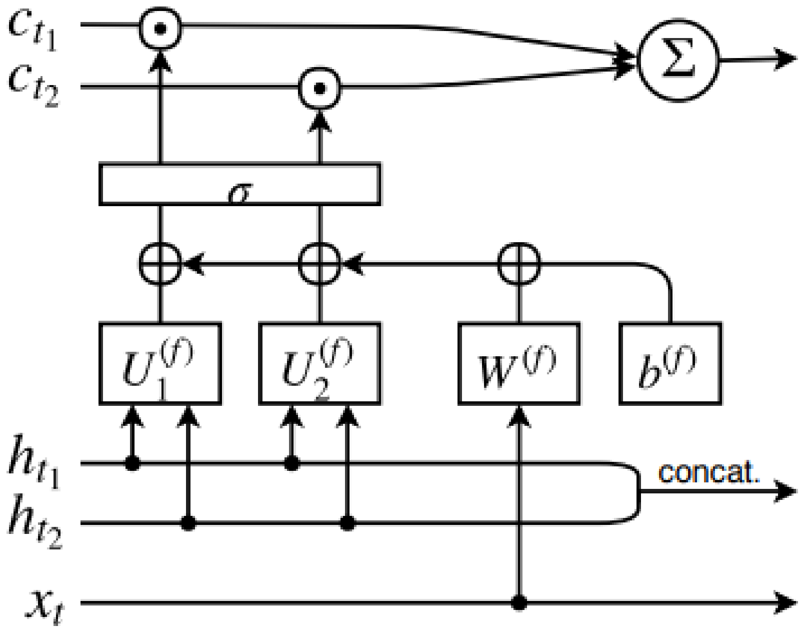
$$
\begin{aligned}
\hat{h_j} &= [h_{j_1}; \cdots; h_{j_n}] \\
f_{j_k} &= \sigma\left(W^{(f)} x_j + U_k^{(f)} \hat{h}_j + b^{(f)}\right) \\
u_j &= \tanh\left(W^{(u)} x_j + U^{(u)} \hat{h}_j + b^{(u)}\right) \\
i_j &= \sigma\left(W^{(i)} x_j + U^{(i)} \hat{h}_j + b^{(i)}\right) \\
o_j &= \sigma\left(W^{(o)} x_j + U^{(o)} \hat{h}_j + b^{(o)}\right) \\
c_j &= \sum_{k \in C(j)} c_k \odot f_{j_k} + i_j \odot u_j \\
h_j &= o_j \odot \tanh(c_j)
\end{aligned}
\tag{5.21}
$$

où $\hat{h_j} \in \mathbb{R}^{d_2 n_j}$ est le vecteur obtenu en concaténant les n_j vecteurs $h_{j_1}, \ldots, h_{j_{n_j}}$.

Contrairement à Child-sum Tree-LSTM, les paramètres $U_k^{(f)} \in \mathbb{R}^{d_2 \times d_2 n_j}$ ne sont pas partagés entre les enfants .

5.7.5 Contexte & résultats

Dans la partie de la géneration des questions dan 2.7, notre but était la géneration des questions pertinentes syntaxiquement en utilisant Tree-LSTM child sum comme fonction de transition. Dans un



N-ary Tree-LSTM

heightAlgorithme	Validation accuracy mean	Test accuracy	Test loss
LSTM Simple binaire	88.23	90.00	0.555
LSTM Child Sum binaire	90.00	88.03	0.628
LSTM Simple Multiclass	51.14	40.00	1.76
LSTM Child Sum Multiclass	62.23	51.00	1.49

premier temps , nous implémentons cet algorithme (Qgen) [voir B] avec les données NLP Stanford [`https://nlp.stanford.edu/sentiment`] pour la classification de sentiments,puis pour le problème de géneration de questions vu comme un porblème de classification multiclass sur l'ensemble de vocabulaire (1 194 mots unique).

Nous présentons dans le tableau **??**, les résultats obtonues de nos implémentation des algorithmes LSTM simple et Tree-LSTM child sum pour une classification binaire et multiclasse [voir B].

Chapitre 6

Conclusion

Dans ce rapport, nous avons proposé de créer un environnement de formation à partir des modèles supervisés profonds afin de former un agent DeepRL afin de résoudre un problème de dialogue multimodal axé sur les objectifs. Nous montrons la promesse de cette approche sur les données GuessWhat ?! , et on observe quantitativement et qualitativement une amélioration encourageante par rapport à un modèle de référence supervisé. Alors que les modèles d'apprentissage supervisé ne génèrent pas de stratégie de dialogue cohérente, notre méthode apprend à quel moment s'arrêter après avoir généré une séquence de questions pertinentes.

Bibliographie

[1] Florian Strub, Harm de Vries, Jeremie Mary, Bilal Piot, Aaron Courville, and Olivier Pietquin. End-to-end optimization of goal-driven and visually grounded dialogue systems, 2017.

[2] Harm de Vries, Florian Strub, Sarath Chandar, Olivier Pietquin, Hugo Larochelle, and Aaron Courville. Guesswhat ?! visual object discovery through multi-modal dialogue, 2016.

[3] Yuxi Li. Deep reinforcement learning : An overview, 2017.

[4] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. 2018.

[5] Matthieu Zimmer. Apprentissage par renforcement développemental, 2018.

[6] Sarah Filippi. Stratégies optimistes en apprentissage par renforcement, 2011.

[7] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning.* MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[8] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[9] Ashwin K Vijayakumar, Michael Cogswell, Ramprasath R. Selvaraju, Qing Sun, Stefan Lee, David Crandall, and Dhruv Batra. Diverse beam search : Decoding diverse solutions from neural sequence models, 2016.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* The MIT Press, 2016.

[11] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning, 2017.

[12] Rene Vidal, Joan Bruna, Raja Giryes, and Stefano Soatto. Mathematics of deep learning, 2017.

[13] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning : going beyond euclidean data. 2016.

[14] Raja Giryes, Guillermo Sapiro, and Alex M. Bronstein. Deep neural networks with random gaussian weights : A universal classification strategy ? 2015.

[15] Joan Bruna Estrach, Arthur Szlam, and Yann LeCun. Signal recovery from pooling representations. In *31st International Conference on Machine Learning, ICML 2014*, volume 2, pages 1585–1598. International Machine Learning Society (IMLS), 2014.

[16] Aravindh Mahendran and Andrea Vedaldi. Understanding deep image representations by inverting them, 2014.

[17] E. J. Candes and T. Tao. Near-optimal signal recovery from random projections : Universal encoding strategies ? *IEEE Transactions on Information Theory*, 52(12) :5406–5425, Dec 2006.

[18] Venkat Chandrasekaran, Benjamin Recht, Pablo A. Parrilo, and Alan S. Willsky. The convex geometry of linear inverse problems. 2010.

[19] Raja Giryes, Yonina C. Eldar, Alex M. Bronstein, and Guillermo Sapiro. Tradeoffs between convergence speed and reconstruction accuracy in inverse problems, 2016.

[20] Lior Wolf and Amnon Shashua. Learning over sets using kernel principal angles. *J. Mach. Learn. Res.*, 4 :913–931, December 2003.

[21] E. Elhamifar and R. Vidal. Sparse subspace clustering : Algorithm, theory, and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(11) :2765–2781, Nov 2013.

[22] Suvro Banerjee. An introduction to recurrent neural networks.

[23] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network, 2018.

[24] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. Automatic source code summarization with extended tree-lstm, 2019.

Annexe A

Implémentation des algorithmes utilisés dans RL

A.1 Exemple d'implémentation de l'algorithme N-Step TD

In [0]:
```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

class RandomWalkMDR:
    """ Defines the Markov reward process
    States are [0, 1, 2, 3, 4, 5, 6] = [Terminal state, A, B, C, D, E, Terminal State]
    Actions are [-1, 1] for left and right steps
    Returns are 0 everywhere except for landing at the right terminal state (state 6)
    """
    def __init__(self):
        self.all_states = np.arange(7)
        self.start_state = 3 # all episodes start at the center state C (here 3)
        self.reset_state()

    def reset_state(self):
        self.state = self.start_state
        self.states_visited = [self.state]
        self.rewards_received = []
        return self.state

    def get_states(self):
        return self.all_states

    def get_reward(self, state):
        # return +1 when an episode terminates on the right
        return int(state == self.all_states[-1])

    def step(self):
        action = [-1, 1][np.random.rand() >= 0.5]  # go left or right with equal probability
        next_state = self.state + action
        reward = self.get_reward(next_state)
        self.rewards_received.append(reward)

        if not self.is_terminal(next_state):
```

```
                self.state = next_state
                self.states_visited.append(next_state)

            return next_state, reward

        def is_terminal(self, state):
            # the two ends of the random walk path are the terminal states
            return (state == self.all_states[0]) or (state == self.all_states[-1])
```

```
In [0]: import numpy as np
        import matplotlib.pyplot as plt
        from tqdm import tqdm
```

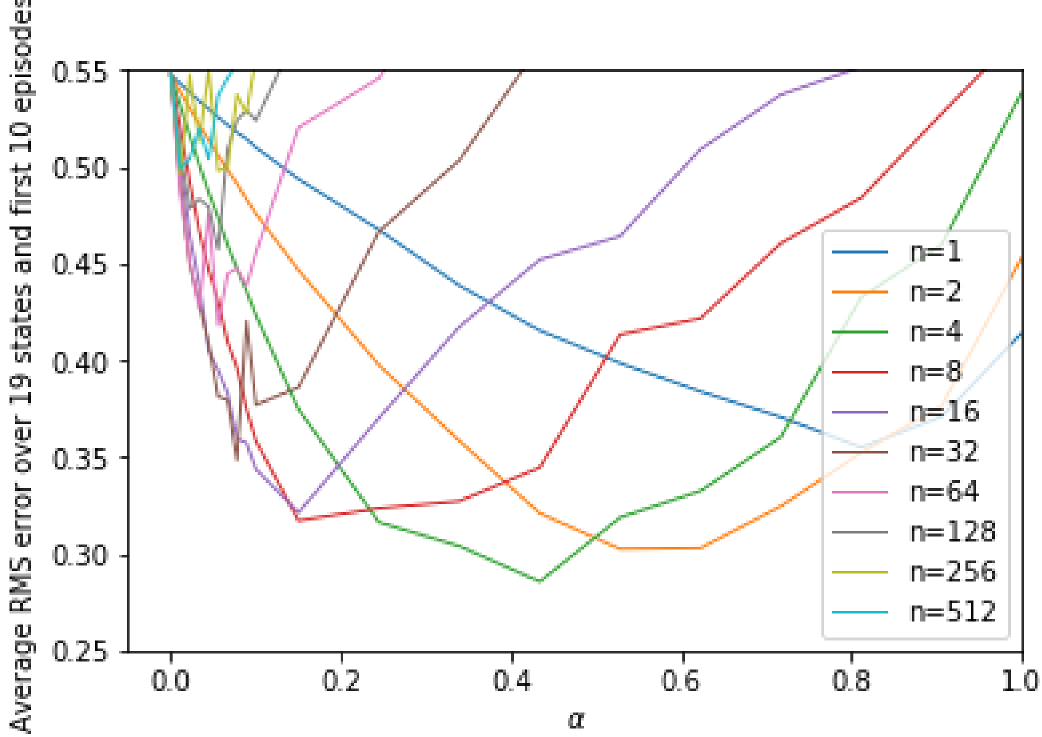
```
In [0]: class RandomWalk(RandomWalkMDR):
            def __init__(self, n_states=21):
                self.all_states = np.arange(n_states)
                self.start_state = max(self.all_states)//2
                self.reset_state()

            def get_reward(self, state):
                if state == self.all_states[0]:
                    return -1
                elif state == self.all_states[-1]:
                    return 1
                else:
                    return 0
```

```
In [0]: def estimate_v(mdr, n_episodes, n, alpha, gamma=1):
            """ Estimate the value function using n-step TD method.
            This maintains a running estimate of the value function for each episode in range(n_epis
            """

            # Initialize records for episode values (v) and values over episodes
            v = np.zeros(len(mdr.get_states()))
            v_over_episodes = np.empty((n_episodes+1, len(mdr.get_states())))
            v_over_episodes[0] = v.copy()

            # Implements Algorithm in Section 7.1 -- n-step TD for estimating v_pi
            for episode in range(1, n_episodes+1):
                # initialize and store S0, T, t
                state = mdr.reset_state()
                T = float('inf')
                t = 0  # time step inside of episode

                # loop for each step of episode, t = 0, 1, 2, ...
                while True:
                    # if we haven't reached the terminal state, take an action
                    if t < T:
                        state, step_reward = mdr.step()
                        if mdr.is_terminal(state):
                            T = t + 1

                    # update state estimate at time tau
                    tau = t - n + 1
                    if tau >= 0:
```

```python
                G = sum(gamma**(i - tau) * mdr.rewards_received[i] for i in range(tau, min(ta
                if tau + n < T:
                    state_tpn =  mdr.states_visited[tau+n]  # state at time step tau + na
                    G += gamma**n * v[state_tpn]
                state_tau = mdr.states_visited[tau]  # state at time step tau
                v[state_tau] += alpha * (G - v[state_tau])

            # episode step
            t += 1
            if tau == T - 1:
                break

        # at the end of each episode, add value estimate for current episode to the aggregat
        v_over_episodes[episode] = v.copy()

    # return average over the episodes for only the non-terminal states
    return v_over_episodes[:,1:-1]


# --------------------
# Figure : Performance of n-step TD methods as a function of , for various values of n,
# on a 19-state random walk task
# --------------------


def fig():
    mdr = RandomWalk()
    true_values = np.linspace(-1, 1, 21)[1:-1]

    n_runs = 10
    n_episodes = 10
    ns = 2**np.arange(10)
    alphas = np.hstack((np.linspace(0, 0.1, 10), np.linspace(0.15, 1, 10)))

    rms_error = np.zeros((n_runs, len(ns), len(alphas)))

    for rep in tqdm(range(n_runs)):
        for i, n in enumerate(ns):
            for j, alpha in enumerate(alphas):
                v = estimate_v(mdr, n_episodes, n, alpha)
                # The performance measure for each parameter setting, shown on the vertical
                # the square-root of the average squared error between the predicitons at th
                # for the 19 states and their true values, then averaged over the first 10 e
                # the whole experiement.
                rms_error[rep, i, j] += np.mean(np.sqrt(np.mean((v - true_values)**2, axis=1

    rms_error = np.mean(rms_error, axis=0)  # avg over runs

    for i, n in enumerate(ns):
        plt.plot(alphas, rms_error[i], label='n={}'.format(n), lw=1)
    plt.xlabel(r'$\alpha$')
    plt.xlim(plt.gca().get_xlim()[0], max(alphas))
    plt.ylim(0.25, 0.55)
    plt.ylabel('Average RMS error over {} states and first {} episodes'.format(len(mdr.all_st
```

```
    plt.legend()

if __name__ == '__main__':
        np.random.seed(1)
        fig()
```

```
100%|| 10/10 [01:10<00:00,  7.03s/it]
```



In [0]:

A.2 Une implémentation fonctionnelle du DQN catégorique (distributional RL).

A.2.1 Utils :

In [0]:
```
import logging
import gym
import gym_fast_envs  # noqa
from termcolor import colored as clr

from utils import EvaluationMonitor
from utils import PreprocessFrames
from utils import SqueezeRewards
from utils import DoneAfterLostLife


def env_factory(cmdl, mode):
```

```python
    # Undo the default logger and configure a new one.
    gym.undo_logger_setup()
    logger = logging.getLogger()
    logger.setLevel(logging.WARNING)

    print(clr("[Main] Constructing %s environment." % mode, attrs=['bold']))
    env = gym.make(cmdl.env_name)

    if hasattr(cmdl, 'rescale_dims'):
        state_dims = (cmdl.rescale_dims, cmdl.rescale_dims)
    else:
        state_dims = env.observation_space.shape[0:2]

    env_class, hist_len, cuda = cmdl.env_class, cmdl.hist_len, cmdl.cuda

    if mode == "training":
        env = PreprocessFrames(env, env_class, hist_len, state_dims, cuda)
        if hasattr(cmdl, 'reward_clamp') and cmdl.reward_clamp:
            env = SqueezeRewards(env)
        if hasattr(cmdl, 'done_after_lost_life') and cmdl.done_after_lost_life:
            env = DoneAfterLostLife(env)
        print('-' * 50)
        return env

    elif mode == "evaluation":
        if cmdl.eval_env_name != cmdl.env_name:
            print(clr("[%s] Warning! evaluating on a different env: %s"
                      % ("Main", cmdl.eval_env_name), 'red', attrs=['bold']))
            env = gym.make(cmdl.eval_env_name)

        env = PreprocessFrames(env, env_class, hist_len, state_dims, cuda)
        env = EvaluationMonitor(env, cmdl)
        print('-' * 50)
        return env


def not_implemented(obj):
    import inspect
    method_name = inspect.stack()[1][3]
    raise RuntimeError(
        clr(("%s.%s not implemented nor delegated." %
(obj.name, method_name)), 'white', 'on_red'))


#### Parsing

""" Functions and classes for parsing config files and command line arguments.
"""
import argparse
import yaml
import os
from termcolor import colored as clr
```

```python
def parse_cmd_args():
    """ Return parsed command line arguments.
    """
    p = argparse.ArgumentParser(description='')
    p.add_argument('-l', '--label', type=str, default="default_label",
                   metavar='label_name::str',
                   help='Label of the current experiment')
    p.add_argument('-id', '--id', type=int, default=0,
                   metavar='label_name::str',
                   help='Id of this instance running within the current' +
                        'experiment')
    p.add_argument('-cf', '--config', type=str, default="catch_dev",
                   metavar='path::str',
                   help='Path to the config file.')
    p.add_argument('-r', '--results', type=str, default="./experiments",
                   metavar='path::str',
                   help='Path of the results folder.')
    args = p.parse_args()
    return args


def to_namespace(d):
    """ Convert a dict to a namespace.
    """
    n = argparse.Namespace()
    for k, v in d.items():
        setattr(n, k, to_namespace(v) if isinstance(v, dict) else v)
    return n


def inject_args(n, args):
    # inject some of the cmdl args into the config namespace
    setattr(n, "experiment_id", args.id)
    setattr(n, "results_path", args.results)
    return n


def check_paths(cmdl):
    if not os.path.exists(cmdl.results_path):
        print(
            clr("%s path for saving results does not exist. Please create it."
                % cmdl.results_path, 'red', attrs=['bold']))
        raise IOError
    else:
        print(clr("Warning, data in %s will be overwritten."
               % cmdl.results_path, 'red', attrs=['bold']))


def parse_config_file(path):
    f = open(path)
    config_data = yaml.load(f, Loader=yaml.SafeLoader)
    f.close()
    return to_namespace(config_data)
```

```
def get_config():
    args = parse_cmd_args()
    cmdl = parse_config_file(args.config)
    cmdl = inject_args(cmdl, args)
    check_paths(cmdl)
return cmdl

#### wrappers
import unittest
import logging
import torch
import numpy as np
import gym
from gym import Wrapper
from gym import ObservationWrapper
from gym import RewardWrapper
from PIL import Image
from termcolor import colored as clr
from collections import OrderedDict
from utils.torch_types import TorchTypes

logger = logging.getLogger(__name__)


class SqueezeRewards(RewardWrapper):
    def __init__(self, env):
        super(SqueezeRewards, self).__init__(env)
        print("[Reward Wrapper] for clamping rewards to -+1")

    def _reward(self, reward):
        return float(np.sign(reward))


class PreprocessFrames(ObservationWrapper):
    def __init__(self, env, env_type, hist_len, state_dims, cuda=None):
        super(PreprocessFrames, self).__init__(env)

        self.env_type = env_type
        self.state_dims = state_dims
        self.hist_len = hist_len
        self.env_wh = self.env.observation_space.shape[0:2]
        self.env_ch = self.env.observation_space.shape[2]
        self.wxh = self.env_wh[0] * self.env_wh[1]

        # need to find a better way
        if self.env_type == "atari":
            self._preprocess = self._atari_preprocess
        elif self.env_type == "catch":
            self._preprocess = self._catch_preprocess
        print("[Preprocess Wrapper] for %s with state history of %d frames."
              % (self.env_type, hist_len))

        self.cuda = False if cuda is None else cuda
```

```
        self.dtype = dtype = TorchTypes(self.cuda)
        self.rgb = dtype.FT([.2126, .7152, .0722])

        # torch.size([1, 4, 24, 24])
        """
        self.hist_state = torch.FloatTensor(1, hist_len, *state_dims)
        self.hist_state.fill_(0)
        """

        self.d = OrderedDict({i: torch.FloatTensor(1, 1, *state_dims).fill_(0)
                              for i in range(hist_len)})

    def _observation(self, o):
        return self._preprocess(o)

    def _reset(self):
        # self.hist_state.fill_(0)
        self.d = OrderedDict(
            {i: torch.FloatTensor(1, 1, *self.state_dims).fill_(0)
                for i in range(self.hist_len)})
        observation = self.env.reset()
        return self._observation(observation)

    def _catch_preprocess(self, o):
        return self._get_concatenated_state(self._rgb2y(o))

    def _atari_preprocess(self, o):
        img = Image.fromarray(self._rgb2y(o).numpy())
        img = np.array(img.resize(self.state_dims, resample=Image.NEAREST))
        th_img = torch.from_numpy(img)
        return self._get_concatenated_state(th_img)

    def _rgb2y(self, o):
        o = torch.from_numpy(o).type(self.dtype.FT)
        s = o.view(self.wxh, 3).mv(self.rgb).view(*self.env_wh) / 255
        return s.cpu()

    def _get_concatenated_state(self, o):
        hist_len = self.hist_len
        for i in range(hist_len - 1):
            self.d[i] = self.d[i + 1]
        self.d[hist_len - 1] = o.unsqueeze(0).unsqueeze(0)
        return torch.cat(list(self.d.values()), 1)

    """
    def _get_concatenated_state(self, o):
        hist_len = self.hist_len  # eg. 4
        # move frames already existent one position below
        if hist_len > 1:
            self.hist_state[0][0:hist_len - 1] = self.hist_state[0][1:hist_len]
        # concatenate the newest frame to the top of the augmented state
        self.hist_state[0][self.hist_len - 1] = o
        return self.hist_state
    """
```

```python
class DoneAfterLostLife(gym.Wrapper):
    def __init__(self, env):
        super(DoneAfterLostLife, self).__init__(env)

        self.no_more_lives = True
        self.crt_live = env.unwrapped.ale.lives()
        self.has_many_lives = self.crt_live != 0

        if self.has_many_lives:
            self._step = self._many_lives_step
        else:
            self._step = self._one_live_step
        not_a = clr("not a", attrs=['bold'])

        print("[DoneAfterLostLife Wrapper]  %s is %s many lives game."
              % (env.env.spec.id, "a" if self.has_many_lives else not_a))

    def _reset(self):
        if self.no_more_lives:
            obs = self.env.reset()
            self.crt_live = self.env.unwrapped.ale.lives()
            return obs
        else:
            return self.__obs

    def _many_lives_step(self, action):
        obs, reward, done, info = self.env.step(action)
        crt_live = self.env.unwrapped.ale.lives()
        if crt_live < self.crt_live:
            # just lost a live
            done = True
            self.crt_live = crt_live

        if crt_live == 0:
            self.no_more_lives = True
        else:
            self.no_more_lives = False
            self.__obs = obs
        return obs, reward, done, info

    def _one_live_step(self, action):
        return self.env.step(action)


class EvaluationMonitor(Wrapper):
    def __init__(self, env, cmdl):
        super(EvaluationMonitor, self).__init__(env)

        self.freq = cmdl.eval_frequency  # in steps
        self.eval_steps = cmdl.eval_steps
        self.cmdl = cmdl
```

```
        if self.cmdl.display_plots:
            import Visdom
            self.vis = Visdom()
            self.plot = self.vis.line(
                Y=np.array([0]), X=np.array([0]),
                opts=dict(
                    title=cmdl.label,
                    caption="Episodic reward per %d steps." % self.eval_steps)
            )

        self.eval_cnt = 0
        self.crt_training_step = 0
        self.step_cnt = 0
        self.ep_cnt = 1
        self.total_rw = 0
        self.max_mean_rw = -1000

        no_of_evals = cmdl.training_steps // cmdl.eval_frequency \
            - (cmdl.eval_start-1) // cmdl.eval_frequency

        self.eval_frame_idx = torch.LongTensor(no_of_evals).fill_(0)
        self.eval_rw_per_episode = torch.FloatTensor(no_of_evals).fill_(0)
        self.eval_rw_per_frame = torch.FloatTensor(no_of_evals).fill_(0)
        self.eval_eps_per_eval = torch.LongTensor(no_of_evals).fill_(0)

    def get_crt_step(self, crt_training_step):
        self.crt_training_step = crt_training_step

    def _reset_monitor(self):
        self.step_cnt, self.ep_cnt, self.total_rw = 0, 0, 0

    def _step(self, action):
        # self._before_step(action)
        observation, reward, done, info = self.env.step(action)
        done = self._after_step(observation, reward, done, info)
        return observation, reward, done, info

    def _reset(self):
        observation = self.env.reset()
        self._after_reset(observation)
        return observation

    def _after_step(self, o, r, done, info):
        self.total_rw += r
        self.step_cnt += 1

        # Evaluation ends here
        if self.step_cnt == self.eval_steps:
            self._update()
            self._reset_monitor()
        return done

    def _after_reset(self, observation):
        if self.step_cnt != self.eval_steps:
```

```
            self.ep_cnt += 1

    def _update(self):
        mean_rw = self.total_rw / (self.ep_cnt - 1)
        max_mean_rw = self.max_mean_rw
        self.max_mean_rw = mean_rw if mean_rw > max_mean_rw else max_mean_rw

        self._update_plot(self.crt_training_step, mean_rw)
        self._display_logs(mean_rw, max_mean_rw)
        self._update_reports(mean_rw)
        self.eval_cnt += 1

    def _update_reports(self, mean_rw):
        idx = self.eval_cnt

        self.eval_frame_idx[idx] = self.crt_training_step
        self.eval_rw_per_episode[idx] = mean_rw
        self.eval_rw_per_frame[idx] = self.total_rw / self.step_cnt
        self.eval_eps_per_eval[idx] = (self.ep_cnt - 1)

        torch.save({
            'eval_frame_idx': self.eval_frame_idx,
            'eval_rw_per_episode': self.eval_rw_per_episode,
            'eval_rw_per_frame': self.eval_rw_per_frame,
            'eval_eps_per_eval': self.eval_eps_per_eval
        }, self.cmdl.results_path + "/eval_stats.torch")

    def _update_plot(self, crt_training_step, mean_rw):
        if self.cmdl.display_plots:
            self.vis.line(
                X=np.array([crt_training_step]),
                Y=np.array([mean_rw]),
                win=self.plot,
                update='append'
            )

    def _display_logs(self, mean_rw, max_mean_rw):
        bg_color = 'on_magenta' if mean_rw > max_mean_rw else 'on_blue'
        print(clr("[Evaluator] done in %5d steps. " % self.step_cnt,
              attrs=['bold'])
              + clr(" rw/ep=%3.2f " % mean_rw, 'white', bg_color,
                    attrs=['bold']))


class VisdomMonitor(Wrapper):
    def __init__(self, env, cmdl):
        super(VisdomMonitor, self).__init__(env)

        self.freq = cmdl.report_freq  # in steps
        self.cmdl = cmdl

        if self.cmdl.display_plots:
            from visdom import Visdom
            self.vis = Visdom()
```

```
            self.plot = self.vis.line(
                Y=np.array([0]), X=np.array([0]),
                opts=dict(
                    title=cmdl.label,
                    caption="Episodic reward per 1200 steps.")
            )

        self.step_cnt = 0
        self.ep_cnt = -1
        self.ep_rw = []
        self.last_reported_ep = 0

    def _step(self, action):
        # self._before_step(action)
        observation, reward, done, info = self.env.step(action)
        done = self._after_step(observation, reward, done, info)
        return observation, reward, done, info

    def _reset(self):
        self._before_reset()
        observation = self.env.reset()
        self._after_reset(observation)
        return observation

    def _after_step(self, o, r, done, info):
        self.ep_rw[self.ep_cnt] += r
        self.step_cnt += 1
        if self.step_cnt % self.freq == 0:
            self._update_plot()
        return done

    def _before_reset(self):
        self.ep_rw.append(0)

    def _after_reset(self, observation):
        self.ep_cnt += 1
        # print("[%2d][%4d]  RESET" % (self.ep_cnt, self.step_cnt))

    def _update_plot(self):
        # print(self.last_reported_ep, self.ep_cnt + 1)
        completed_eps = self.ep_rw[self.last_reported_ep:self.ep_cnt + 1]
        ep_mean_reward = sum(completed_eps) / len(completed_eps)
        if self.cmdl.display_plots:
            self.vis.line(
                X=np.array([self.step_cnt]),
                Y=np.array([ep_mean_reward]),
                win=self.plot,
                update='append'
            )
        self.last_reported_ep = self.ep_cnt + 1


class TestAtariWrappers(unittest.TestCase):
```

```
    def _test_env(self, env_name):
        env = gym.make(env_name)
        env = DoneAfterLostLife(env)

        o = env.reset()

        for i in range(10000):
            o, r, d, _ = env.step(env.action_space.sample())
            if d:
                o = env.reset()
                print("%3d, %s, %d" % (i, env_name, env.unwrapped.ale.lives()))

    def test_pong(self):
        print("Testing Pong")
        self._test_env("Pong-v0")

    def test_frostbite(self):
        print("Testing Frostbite")
        self._test_env("Frostbite-v0")


if __name__ == "__main__":
    import unittest
  unittest.main()


#### Torch Types
import torch


class TorchTypes(object):

    def __init__(self, cuda=False):
        self.set_cuda(cuda)

    def set_cuda(self, use_cuda):
        if use_cuda:
            self.FT = torch.cuda.FloatTensor
            self.LT = torch.cuda.LongTensor
            self.BT = torch.cuda.ByteTensor
            self.IT = torch.cuda.IntTensor
            self.DT = torch.cuda.DoubleTensor
        else:
            self.FT = torch.FloatTensor
            self.LT = torch.LongTensor
            self.BT = torch.ByteTensor
            self.IT = torch.IntTensor
            self.DT = torch.DoubleTensor
```

A.2.2 Agents :

```
In [0]: ## Base
        import time
        from termcolor import colored as clr
```

```python
from utils import not_implemented


class BaseAgent(object):
    def __init__(self, env_space):
        self.actions = env_space[0]
        self.action_no = self.actions.n
        self.state_dims = env_space[1].shape[0:2]

        self.step_cnt = 0
        self.ep_cnt = 0
        self.ep_reward_cnt = 0
        self.ep_reward = []
        self.max_mean_rw = -100

    def evaluate_policy(self, obs):
        not_implemented(self)

    def improve_policy(self, _state, _action, reward, state, done):
        not_implemented(self)

    def gather_stats(self, reward, done):
        self.step_cnt += 1
        self.ep_reward_cnt += reward
        if done:
            self.ep_cnt += 1
            self.ep_reward.append(self.ep_reward_cnt)
            self.ep_reward_cnt = 0

    def display_setup(self, env, config):
        emph = ["env_name", "agent_type", "label", "batch_size", "lr",
                "hist_len"]
        print("-------------------------------------------------")
        for k in config.__dict__:
            if config.__dict__[k] is not None:
                v = config.__dict__[k]
                space = "." * (32 - len(k))
                config_line = "%s:  %s  %s" % (k, space, v)
                for e in emph:
                    if k == e:
                        config_line = clr(config_line, attrs=['bold'])
                print(config_line)
        print("-------------------------------------------------")
        custom = {"no_of_actions": self.action_no}
        for k, v in custom.items():
            space = "." * (32 - len(k))
            print("%s:  %s  %s" % (k, space, v))
        print("-------------------------------------------------")

    def display_stats(self, start_time):
        fps = self.cmdl.report_frequency / (time.perf_counter() - start_time)

        print(clr("[%s] step=%7d, fps=%.2f " % (self.name, self.step_cnt, fps),
                  attrs=['bold']))
```

```
            self.ep_reward.clear()

        def display_final_report(self, ep_cnt, step_cnt, global_time):
            elapsed_time = time.perf_counter() - global_time
            fps = step_cnt / elapsed_time
            print(clr("[  %s   ] finished after %d eps, %d steps.  "
                  % ("Main", ep_cnt, step_cnt), 'white', 'on_grey'))
            print(clr("[  %s   ] finished after %.2fs, %.2ffps.   "
                  % ("Main", elapsed_time, fps), 'white', 'on_grey'))

        def display_model_stats(self):
               pass
```

In [0]:
```
#### categorical_dqn_agent
from agents.dqn_agent import DQNAgent
from estimators import get_estimator as get_model
from policy_evaluation import CategoricalPolicyEvaluation
from policy_improvement import CategoricalPolicyImprovement


class CategoricalDQNAgent(DQNAgent):
    def __init__(self, action_space, cmdl):
        DQNAgent.__init__(self, action_space, cmdl)
        self.name = "Categorical_agent"
        self.cmdl = cmdl

        hist_len, action_no = cmdl.hist_len, self.action_no
        self.policy = policy = get_model(cmdl.estimator, 1, hist_len,
                                         (action_no, cmdl.atoms_no),
                                         hidden_size=cmdl.hidden_size)
        self.target = target = get_model(cmdl.estimator, 1, hist_len,
                                         (action_no, cmdl.atoms_no),
                                         hidden_size=cmdl.hidden_size)
        if self.cmdl.cuda:
            self.policy.cuda()
            self.target.cuda()

        self.policy_evaluation = CategoricalPolicyEvaluation(policy, cmdl)
        self.policy_improvement = CategoricalPolicyImprovement(
                policy, target, cmdl)

    def improve_policy(self, _s, _a, r, s, done):
        h = self.cmdl.hist_len - 1
        self.replay_memory.push(_s[0, h], _a, r, done)

        if len(self.replay_memory) < self.cmdl.start_learning_after:
            return

        if (self.step_cnt % self.cmdl.update_freq == 0) and (
                len(self.replay_memory) > self.cmdl.batch_size):

            # get batch of transitions
            batch = self.replay_memory.sample()
```

```
                # compute gradients
                self.policy_improvement.accumulate_gradient(*batch)
                self.policy_improvement.update_model()

            if self.step_cnt % self.cmdl.target_update_freq == 0:
                self.policy_improvement.update_target_net()

    def display_model_stats(self):
        self.policy_improvement.get_model_stats()
        print("MaxQ=%2.2f.  MemSz=%5d.  Epsilon=%.2f." % (
self.max_q, len(self.replay_memory), self.epsilon))
```

In [0]:
```
### dqn_agent
from numpy.random import uniform
from agents.base_agent import BaseAgent
from estimators import get_estimator as get_model
from policy_evaluation import DeterministicPolicy as DQNEvaluation
from policy_evaluation import get_schedule as get_epsilon_schedule
from policy_improvement import DQNPolicyImprovement as DQNImprovement
from data_structures import ExperienceReplay
from utils import TorchTypes


class DQNAgent(BaseAgent):
    def __init__(self, env_space, cmdl):
        BaseAgent.__init__(self, env_space)
        self.name = "DQN_agent"
        self.cmdl = cmdl
        eps = self.cmdl.epsilon
        e_steps = self.cmdl.epsilon_steps

        self.policy = policy = get_model(cmdl.estimator, 1, cmdl.hist_len,
                                         self.action_no, cmdl.hidden_size)
        self.target = target = get_model(cmdl.estimator, 1, cmdl.hist_len,
                                         self.action_no, cmdl.hidden_size)
        if self.cmdl.cuda:
            self.policy.cuda()
            self.target.cuda()
        self.policy_evaluation = DQNEvaluation(policy)
        self.policy_improvement = DQNImprovement(policy, target, cmdl)

        self.exploration = get_epsilon_schedule("linear", eps, 0.05, e_steps)
        self.replay_memory = ExperienceReplay.factory(cmdl, self.state_dims)

        self.dtype = TorchTypes(cmdl.cuda)
        self.max_q = -1000

    def evaluate_policy(self, state):
        self.epsilon = next(self.exploration)
        if self.epsilon < uniform():
            qval, action = self.policy_evaluation.get_action(state)
            self.max_q = max(qval, self.max_q)
            return action
```

```
            else:
                return self.actions.sample()

        def improve_policy(self, _s, _a, r, s, done):
            h = self.cmdl.hist_len - 1
            self.replay_memory.push(_s[0, h], _a, r, done)

            if len(self.replay_memory) < self.cmdl.start_learning_after:
                return

            if (self.step_cnt % self.cmdl.update_freq == 0) and (
                    len(self.replay_memory) > self.cmdl.batch_size):

                # get batch of transitions
                batch = self.replay_memory.sample()

                # compute gradients
                self.policy_improvement.accumulate_gradient(*batch)
                self.policy_improvement.update_model()

            if self.step_cnt % self.cmdl.target_update_freq == 0:
                self.policy_improvement.update_target_net()

        def display_model_stats(self):
            self.policy_improvement.get_model_stats()
            print("MaxQ=%2.2f.  MemSz=%5d.  Epsilon=%.2f." % (
            self.max_q, len(self.replay_memory), self.epsilon))
            self.max_q = -1000
```

In [0]:
```
#### evaluation_agent
from numpy.random import uniform
from estimators import get_estimator as get_model
from policy_evaluation import DeterministicPolicy
from policy_evaluation import CategoricalPolicyEvaluation


class EvaluationAgent(object):
    def __init__(self, env_space, cmdl):
        self.name = "Evaluation"

        self.actions = env_space[0]
        self.action_no = action_no = self.actions.n
        self.cmdl = cmdl
        self.epsilon = 0.05

        if cmdl.agent_type == "dqn":
            self.policy = policy = get_model(cmdl.estimator, 1, cmdl.hist_len,
                                             self.action_no, cmdl.hidden_size)
            if self.cmdl.cuda:
                self.policy.cuda()
            self.policy_evaluation = DeterministicPolicy(policy)
        elif cmdl.agent_type == "categorical":
            self.policy = policy = get_model(cmdl.estimator, 1, cmdl.hist_len,
                                             (action_no, cmdl.atoms_no),
```

```
                                                    hidden_size=cmdl.hidden_size)
                if self.cmdl.cuda:
                    self.policy.cuda()
                self.policy_evaluation = CategoricalPolicyEvaluation(policy, cmdl)
            print("[%s]  Evaluating %s agent." % (self.name, cmdl.agent_type))

            self.max_q = -1000

        def evaluate_policy(self, state):
            if self.epsilon < uniform():
                qval, action = self.policy_evaluation.get_action(state)
                self.max_q = max(qval, self.max_q)
                return action
            else:
    return self.actions.sample()
```

```
In [0]: #### random_agent
        from .base_agent import BaseAgent


        class RandomAgent(BaseAgent):
            def __init__(self, action_space, cmdl):
                BaseAgent.__init__(self, action_space)

                self.name = "RND_agent"

            def evaluate_policy(self, state):
                return self.action_space.sample()

            def improve_policy(self, _state, _action, reward, state, done):
        pass
```

A.2.3 Estimateurs : Exemple - Jeu Aatari

```
In [0]: """ Architecture de réseau de neurones pour les jeux Atari.
        """
        import torch
        import torch.nn as nn
        import torch.nn.functional as F


        class AtariNet(nn.Module):
            def __init__(self, input_channels, hist_len, out_size, hidden_size=256):
                super(AtariNet, self).__init__()
                self.input_channels = input_channels
                self.hist_len = hist_len
                self.input_depth = input_depth = hist_len * input_channels
                if type(out_size) is tuple:
                    self.is_categorical = True
                    self.action_no, self.atoms_no = out_size
                    self.out_size = self.action_no * self.atoms_no
                else:
                    self.is_categorical = False
                    self.out_size = out_size
```

```
        self.hidden_size = hidden_size

        self.conv1 = nn.Conv2d(input_depth, 32, kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
        self.lin1 = nn.Linear(64 * 7 * 7, self.hidden_size)
        self.head = nn.Linear(self.hidden_size, self.out_size)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.lin1(x.view(x.size(0), -1)))
        out = self.head(x.view(x.size(0), -1))
        if self.is_categorical:
            splits = out.chunk(self.action_no, 1)
            return torch.stack(list(map(lambda s: F.softmax(s), splits)), 1)
        else:
            return out

    def get_attributes(self):
        return (self.input_channels, self.hist_len, self.action_no,self.hidden_size)
```

A.2.4 Evaluation catégorique de la politique

```
In [0]: import torch
        from torch.autograd import Variable
        from utils import TorchTypes


        class CategoricalPolicyEvaluation(object):
            def __init__(self, policy, cmdl):
                """Assumes policy returns an autograd.Variable"""
                self.name = "CP"
                self.cmdl = cmdl
                self.policy = policy

                self.dtype = dtype = TorchTypes(cmdl.cuda)
                self.support = torch.linspace(cmdl.v_min, cmdl.v_max, cmdl.atoms_no)
                self.support = self.support.type(dtype.FT)

            def get_action(self, state):
                """ Takes best action based on estimated state-action values."""
                state = state.type(self.dtype.FT)
                probs = self.policy(Variable(state, volatile=True)).data
                support = self.support.expand_as(probs)
                q_val, argmax_a = torch.mul(probs, support).squeeze().sum(1).max(0)
                return (q_val[0], argmax_a[0])
```

Annexe B

Implémentation des variantes LSTM

B.1 Implémentation : LSTM simple pour la classification binaire

```
In [0]: #### Import libraries

        from string import digits ##text
        import re
        import pandas as pd
        import csv
        import numpy
        import numpy as np
        import torch
        import ast
        import torch
        from torch.utils.data import DataLoader, TensorDataset
```

B.1.1 Fonctions auxiliaires

```
In [0]: def _label_node_index(node, n=0):
            node['index'] = n
            for child in node['children']:
                n += 1
                _label_node_index(child, n)


        def _gather_node_attributes(node, key):
            features = node[key]
            for child in node['children']:
                features.extend(_gather_node_attributes(child, key))
            return features

        def tree_to_tensors(tree, device=torch.device('cpu')):
            _label_node_index(tree)

            features = _gather_node_attributes(tree, 'Features')
            labels = _gather_node_attributes(tree, 'Labels')


            return {
                'Features': features,
```

```
        'Labels': labels
    }

def string_to_dict(s):
    return ast.literal_eval(s)
```

```
In [0]: ### Fonctions : prédection et calcul de l'accuracy

        def predicted(h):
              t=[]
              for i in range(len(h)):
                    if h[i]>= 0.5 :
                        t.append(1.)
                    else :
                        t.append(0.)
              return torch.FloatTensor(t)

        def get_accuracy(lab,pred):
            correct = (pred== lab).float().sum().item()
            accuracy = 100.*correct / len(lab)

            return accuracy

        def get_label_tensor(t):
          l=[]
          for i in range(len(t)):
              l.append([t[i][-1]])

          return torch.tensor(l)
```

B.1.2 Data

```
In [0]: ### Loading Data ; trees_bin and vocabulary

        def load():
          temp = open("trees_new_bin.txt",'r').readlines()
          trees=[string_to_dict(l.strip('\n'+'\t')) for l in temp]
          v=open("vocab_build.txt",'r').readlines()
          vocab=[l.strip('\n'+'\t') for l in v]
          return trees,vocab

        trees,vocab=load()

        #### Extract features & labels

        def get_data():
          t= open("data_lstm.txt",'r').readlines()
          data=[l.strip('\n'+'\t') for l in t]
          features=data[0]
          labels=data[1]
          return ast.literal_eval(features),ast.literal_eval(labels)


        Features,Labels=get_data()
```

```
In [0]: ### Fonctions : padding features & labels

        pad_dim=20  ### pading dimension
        def pad_features(reviews_int, seq_length):
              features = np.zeros((len(reviews_int), seq_length), dtype = int)
              for i, review in enumerate(reviews_int):
                  review_len = len(review)
                  if review_len <= seq_length:
                      zeroes = list(np.zeros(seq_length-review_len))
                      new = zeroes+review
                  elif review_len > seq_length:
                      new = review[0:seq_length]

                  features[i,:] = np.array(new)

              return features

        def pad_labels(Labels, seq_length):
              labels = np.zeros((len(Labels), seq_length), dtype = int)
              for i, label in enumerate(Labels):
                  Labels_len = len(label)
                  if Labels_len <= seq_length:
                      zeroes = list(np.zeros(seq_length-Labels_len))
                      new = zeroes+label
                  elif Labels_len > seq_length:
                      new = label[0:seq_length]

                  labels[i,:] = np.array(new)

              return labels



        n=1000 ## restriction
        features=pad_features(Features[:n], pad_dim)

        labels=pad_labels(Labels[:n],pad_dim)

In [0]: def split_data(split_frac):
          len_feat=len(features)
          len_feat=len(features)
          train_x = features[0:int(split_frac*len_feat)]
          train_y = labels[0:int(split_frac*len_feat)]
          remaining_x = features[int(split_frac*len_feat):]
          remaining_y = labels[int(split_frac*len_feat):]
          valid_x = remaining_x[0:int(len(remaining_x)*0.5)]
          valid_y = remaining_y[0:int(len(remaining_y)*0.5)]
          test_x = remaining_x[int(len(remaining_x)*0.5):]
          test_y = remaining_y[int(len(remaining_y)*0.5):]
          return train_x,train_y,valid_x,valid_y,test_x,test_y


        split_frac=0.8
        train_x,train_y,valid_x,valid_y,test_x,test_y=split_data(split_frac)
```

```
In [0]: # create Tensor datasets
        train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
        valid_data = TensorDataset(torch.from_numpy(valid_x), torch.from_numpy(valid_y))
        test_data = TensorDataset(torch.from_numpy(test_x), torch.from_numpy(test_y))
        # dataloaders
        batch_size = 10
        # make sure to SHUFFLE your data
        train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
        valid_loader = DataLoader(valid_data, shuffle=True, batch_size=batch_size)
        test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size)
```

```
In [0]: # data iterator
        # obtain one batch of training data
        dataiter = iter(train_loader)
        sample_x, sample_y = dataiter.next()
        print('Sample input size: ', sample_x.size()) # batch_size, seq_length
        print('Sample input: \n', sample_x)
        print()
        print('Sample label size: ', sample_y.size()) # batch_size
        print('Sample label: \n', sample_y)
```

```
Sample input size:  torch.Size([10, 20])
Sample input:
 tensor([[    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,  5529,  2285,  7110, 19705, 12766,  2489,  4484],
        [ 6209,  9707, 16118,  2574, 19365,  2180,  6706, 14128, 10450, 10132,
         19130, 14170, 10450,  2187,  6945,  6361,  9501,  2180, 16704,  5646],
        [14432,  2656, 12990, 14091,  4284, 14237,  8940, 15080,   117,  2323,
         13570, 17902, 11732,  7265,  5306,  3400,   680,   885,  8745,  6103],
        [10447,  9939,  7951,  9939, 15124, 16118,  4912, 19643,  2167,  6361,
          7110,  7117,  4912, 16627,  6090,  8456,  8248,  7695, 14128,  6462],
        [ 3781, 13448, 11237, 11264, 11732, 11770, 14432, 11259,  4468, 17558,
          6361,  9501, 12766, 10947,  3518, 11569,  9939, 16627,  7878,  2285],
        [    0,     0,     0, 15660,  7110, 14091,  7117, 11237, 14170,  7951,
           767,  5932,  6361, 12766,  7117,  5916,  1814,  2020, 14802,  4484],
        [ 7514,  8403, 13570, 10105,  7951, 11259, 13302, 17620,  7117,  5135,
         14170,    18,  2265,  8456, 17086,  8745,  1381,  6361,  6522, 17388],
        [    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0, 17897,  7439,  9939, 13712, 16326, 11055,  2020,  4213,  4484],
        [    0,     0,     0,     0,     0,  8253,  9393,  1874,  9939, 18724,
          9939, 14128,  9939,  4179, 11016,  6090, 10330, 12707,  3280,  4484],
        [    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0, 16322, 16327, 10301, 15328, 12593,  4484]])

Sample label size:  torch.Size([10, 20])
Sample label:
 tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]])
```

B.1.3 LSTM Network Architecture

```
In [0]: ### LSTM Network Architecture

        import torch.nn as nn

        class SentimentLSTM(nn.Module):
            """
            The RNN model that will be used to perform Sentiment analysis.
            """

            def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers, drop_pr
                """
                Initialize the model by setting up the layers.
                """
                super().__init__()

                self.output_size = output_size
                self.n_layers = n_layers
                self.hidden_dim = hidden_dim

                # embedding and LSTM layers
                self.embedding = nn.Embedding(vocab_size, embedding_dim)
                self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                                    dropout=drop_prob, batch_first=True)

                # dropout layer
                self.dropout = nn.Dropout(0.3)

                # linear and sigmoid layers
                self.fc = nn.Linear(hidden_dim, output_size)
                self.sig = nn.Sigmoid()
                self.soft=nn.Softmax(dim=1)


            def forward(self, x, hidden):
                """
                Perform a forward pass of our model on some input and hidden state.
                """
                batch_size = x.size(0)

                # embeddings and lstm_out
                embeds = self.embedding(x)
                lstm_out, hidden = self.lstm(embeds, hidden)

                # stack up lstm outputs
                #lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)

                # dropout and fully-connected layer
                out = self.dropout(lstm_out)
                out = self.fc(out)
```

```
        # sigmoid function

        sig_out = self.sig(out)

        # reshape to be batch_size first
        sig_out = sig_out.view(batch_size, -1)
        sig_out = sig_out[:, -1] # get last batch of labels

        # return last sigmoid output and hidden state
        return sig_out, hidden


    def init_hidden(self, batch_size):
        ''' Initializes hidden state '''
        # Create two new tensors with sizes n_layers x batch_size x hidden_dim,
        # initialized to zero, for hidden state and cell state of LSTM
        weight = next(self.parameters()).data

        hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_(),
                  weight.new(self.n_layers, batch_size, self.hidden_dim).zero_())

        return hidden
```

In [0]:
```
# Initier e modèle et les hyperparamètres
vocab_size = len(vocab) # +1 for the 0 padding
output_size = 1
embedding_dim = 400
hidden_dim = 256
n_layers = 2
def model(vocab_size,output_size,embedding_dim,hidden_dim,n_layers):
  net = SentimentLSTM(vocab_size, output_size, embedding_dim, hidden_dim, n_layers)
  print(net)


model(vocab_size,output_size,embedding_dim,hidden_dim,n_layers)
```

```
SentimentLSTM(
  (embedding): Embedding(20216, 400)
  (lstm): LSTM(400, 256, num_layers=2, batch_first=True, dropout=0.5)
  (dropout): Dropout(p=0.3)
  (fc): Linear(in_features=256, out_features=1, bias=True)
  (sig): Sigmoid()
  (soft): Softmax()
)
```

In [0]:
```
# loss and optimization functions
lr=0.001

criterion = nn.BCELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
val_acc=0
val_acc_list=[]
```

```
# training params

epochs =5 # 3-4 is approx where I noticed the validation loss stop decreasing

counter = 0
print_every = 100
clip=5 # gradient clipping
num_correct = 0


net.train()
# train for some number of epochs
for e in range(epochs):
    # initialize hidden state
    h = net.init_hidden(batch_size)

    # batch loop
    i=0
    for inputs, labels in train_loader:
        counter += 1

             # Creating new variables for the hidden state, otherwise
        # we'd backprop through the entire training history
        h = tuple([each.data for each in h])

        # zero accumulated gradients
        net.zero_grad()

        # get the output from the model
        inputs = inputs.type(torch.LongTensor)
        output,h = net(inputs, h)
        lab=get_label_tensor(labels).float().squeeze()
        loss = criterion(output.squeeze(),  lab)
        loss.backward()
        # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
        nn.utils.clip_grad_norm_(net.parameters(), clip)
        optimizer.step()

        # loss stats
        if counter % print_every == 0:
            # Get validation loss
            val_h = net.init_hidden(batch_size)
            val_losses = []
            net.eval()
            i=0
            for inputs, labels in valid_loader:

                # Creating new variables for the hidden state, otherwise
                # we'd backprop through the entire training history
                val_h = tuple([each.data for each in val_h])

                inputs = inputs.type(torch.LongTensor)
                output, val_h = net(inputs, val_h)
                val_loss = criterion(output.squeeze(),lab)
```

```
                    val_losses.append(val_loss.item())
                    #pred = torch.round(output.squeeze())  # rounds to the nearest integer
                    #correct_tensor = pred.eq(get_label_tensor(labels).float().squeeze().view_as
                    out=output.squeeze()
                    #pred = torch.round(output)
                    pred=predicted(out)
                    #correct =  np.squeeze(correct_tensor.cpu().numpy())
                    #num_correct += np.sum(correct)
                    print(lab,pred)
                    acc_val=get_accuracy(lab,pred)
                    print(acc_val)
                    val_acc+=acc_val
                    i+=1
                net.train()
                print("Epoch: {}/{}...".format(e+1, epochs),
                      "Step: {}...".format(counter),
                      "Loss: {:.6f}...".format(loss.item()),

                      "Val Loss: {:.6f}".format(np.mean(val_losses)))
                print("Val_acc : ",val_acc/i)
                val_acc_list.append(val_acc/i)
                val_acc=0.
               ## modifier label indivi
    print("Val_acc mean: ",np.mean(val_acc_list))
```

In [0]:
```
# Get test data loss and accuracy

test_losses = [] # track loss
num_correct = 0.
acc_mean=0.
# init hidden state
h = net.init_hidden(batch_size)

net.eval()
# iterate over test data
i=0
for inputs, labels in test_loader:


    # Creating new variables for the hidden state, otherwise
    # we'd backprop through the entire training history
    h = tuple([each.data for each in h])


    # get predicted outputs
    inputs = inputs.type(torch.LongTensor)
    output, h = net(inputs, h)
    lab=get_label_tensor(labels).float().squeeze()
    out=output.squeeze()
    # calculate loss
    test_loss = criterion(out,lab)
    test_losses.append(test_loss.item())
```

```
            # convert output probabilities to predicted class (0 or 1)
            #pred = torch.round(output.squeeze())  # rounds to the nearest integer



            #pred = torch.round(output)
            pred=predicted(out)
            acc=get_accuracy(lab,pred)
            # compare predictions to true label
            #correct_tensor = pred.eq(get_label_tensor(labels).float().squeeze().view_as(pred))
            #correct =  np.squeeze(correct_tensor.cpu().numpy())
            #num_correct += np.sum(correct)
            acc_mean+=acc
            i+=1

        # -- stats! -- ##
        # avg test loss
        print("Test loss: {:.3f}".format(np.mean(test_losses)))
        #print(acc_mean/i,i,len(test_loader.dataset))
        # accuracy over all test data
        #test_acc =100.* num_correct/len(test_loader.dataset)
        print("Test accuracy: {:.3f}".format(acc_mean/i))
```

```
Test loss: 0.555
Test accuracy: 90.000
```

In [0]:

B.2 Implémentation : LSTM simple pour la classification multiclass

In [0]:
```
#### import libraries

import numpy as np
import numpy
import torch
from tqdm import tqdm
import random
import ast
import torch.optim as optim
from sklearn.metrics import confusion_matrix
from  statistics import mean
import seaborn as sns
import matplotlib.pyplot as plt

import pandas as pd
```

In [0]:
```
def _label_node_index(node, n=0):
    node['index'] = n
    for child in node['children']:
        n += 1
        _label_node_index(child, n)
```

```
def _gather_node_attributes(node, key):
    features = node[key]
    for child in node['children']:
        features.extend(_gather_node_attributes(child, key))
    return features
```

```
In [0]: from string import digits ##text
        import re
        import pandas as pd
        import csv
        #### import libraries
        import numpy
        import numpy as np
        import torch
        import ast
        from tqdm import tqdm


        from torch.utils.data import DataLoader, TensorDataset
```

```
In [0]: def tree_to_tensors(tree, device=torch.device('cpu')):
            _label_node_index(tree)

            features = _gather_node_attributes(tree, 'Features')
            labels = _gather_node_attributes(tree, 'Labels')


            return {
                'Features': features,
                'Labels': labels
            }
```

```
In [0]: import ast

        def string_to_dict(s):
            return ast.literal_eval(s)

        def load():
          temp = open("trees_new_bin.txt",'r').readlines()
          trees=[string_to_dict(l.strip('\n'+'\t')) for l in temp]
          v=open("vocab_build.txt",'r').readlines()
          vocab=[l.strip('\n'+'\t') for l in v]
          return trees,vocab

        trees,vocab=load()
```

```
In [0]: def get_data():
          t= open("data_lstm_multi.txt",'r').readlines()
          data=[l.strip('\n'+'\t') for l in t]
          features=data[0]
          labels=data[1]
          return ast.literal_eval(features),ast.literal_eval(labels)
```

```
In [0]: Features,Labels=get_data()
        pad_dim=20
```

```
In [0]: def get_accuracy(lab,pred):
            correct = (pred== lab).float().sum().item()
            accuracy = 100.*correct / len(lab)

            return accuracy
```

```
In [0]: def get_accuracy_v2(lab,pred):
            correct = (pred== lab).float().sum().item()
            accuracy = 100.*correct / len(lab)

            return accuracy
```

```
In [0]: def predicted_v2(h):
          l=[]
          for i in range(len(h)):
            _, ind= h[i].max(0)
            l.append(ind)
          return torch.tensor(l).squeeze()
```

```
In [0]: def predicted(h):
              t=[]
              for i in range(len(h)):
                    if h[i]>= 0.5 :
                        t.append(1.)
                    else :
                        t.append(0.)
              return torch.FloatTensor(t)
```

```
In [0]: def pad_features(reviews_int, seq_length):
              features = np.zeros((len(reviews_int), seq_length), dtype = int)
              for i, review in enumerate(reviews_int):
                  review_len = len(review)
                  if review_len <= seq_length:
                      zeroes = list(np.zeros(seq_length-review_len))
                      new = zeroes+review
                  elif review_len > seq_length:
                      new = review[0:seq_length]

                  features[i,:] = np.array(new)

              return features
```

```
In [0]: n=1000
        features=pad_features(Features[:n], pad_dim)
```

```
In [0]: features.shape
```

```
Out[0]: (1000, 20)
```

```
In [0]: def pad_labels(Labels, seq_length):
              labels = np.zeros((len(Labels), seq_length), dtype = int)
              for i, label in enumerate(Labels):
                  Labels_len = len(label)
```

```
            if Labels_len <= seq_length:
                zeroes = list(np.zeros(seq_length-Labels_len))
                new = zeroes+label
            elif Labels_len > seq_length:
                new = label[0:seq_length]

            labels[i,:] = np.array(new)

        return labels
```

In [0]:
```
labels=pad_labels(Labels[:n],pad_dim)
labels.shape
```

Out[0]: (1000, 20)

In [0]:
```
labels.shape
```

Out[0]: (1000, 20)

In [0]:
```
def get_label_tensor(t):
  l=[]
  for i in range(len(t)):
      l.append([t[i][-1]])

  return torch.tensor(l)
```

In [0]:
```
get_label_tensor(labels).size()
```

Out[0]: torch.Size([1000, 1])

In [0]:
```
split_frac = 0.8
len_feat=len(features)
train_x = features[0:int(split_frac*len_feat)]
train_y = labels[0:int(split_frac*len_feat)]
remaining_x = features[int(split_frac*len_feat):]
remaining_y = labels[int(split_frac*len_feat):]
r_x=int(len(remaining_x)*0.5)
valid_x = remaining_x[0:int(len(remaining_x)*0.5)]
valid_y = remaining_y[0:int(len(remaining_y)*0.5)]
test_x = remaining_x[int(len(remaining_x)*0.5):]
test_y = remaining_y[int(len(remaining_y)*0.5):]
```

In [0]:
```
import torch
from torch.utils.data import DataLoader, TensorDataset
# create Tensor datasets
train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
valid_data = TensorDataset(torch.from_numpy(valid_x), torch.from_numpy(valid_y))
test_data = TensorDataset(torch.from_numpy(test_x), torch.from_numpy(test_y))
# dataloaders
batch_size = 10
# make sure to SHUFFLE your data
train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
valid_loader = DataLoader(valid_data, shuffle=True, batch_size=batch_size)
test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size)
```

In [0]:
```
# data iterator
# obtain one batch of training data
```

```
dataiter = iter(train_loader)
sample_x, sample_y = dataiter.next()
print('Sample input size: ', sample_x.size()) # batch_size, seq_length
print('Sample input: \n', sample_x)
print()
print('Sample label size: ', sample_y.size()) # batch_size
print('Sample label: \n', sample_y)
```

```
Sample input size:  torch.Size([10, 20])
Sample input:
 tensor([[    0,   548,  8745, 12654,  6617, 12355,  6849,  6361, 16511,  8456,
          8291,  7730, 10044,  7117, 11237, 15019, 10642,  9501, 14264,  4484],
        [19793,  5646,  3799, 11732, 15030, 19617,  7563,  5976, 12856, 11259,
          3124,  8456,  1684, 15546, 13929,  7563,  9501,  9183, 12766,  1153],
        [    0,     0,     0,  2569,   285,  7951, 18655,  6361, 16689,  4912,
           553,  6361,  1321,  4034,  2989, 15802,   974, 14128, 17492,  4484],
        [14432, 18435, 16118, 13259, 18156,  6361,  8112,  7332,  7951, 14929,
         11016, 14128,  4912,  5183, 16677,  9939, 13842,  8456, 19023,  8745],
        [    0,     0,     0,     0, 14432,  4366,  6689,  6090, 16029, 15802,
          7951,   163, 14128,  3518,  9880, 13343,  9939,  2305, 11803,  4484],
        [13226, 11303, 15080, 16255,  6361, 14157,  9410,  3013,  4670,  7354,
          8456,  2391, 16118,  4366,  9344,  6361, 16118,  1421,  7117,  4483],
        [ 8917, 13570, 11054,  3217, 17993,  7332,  5264, 12159, 15303,  9183,
         11055,  1531, 15802,  1597,  6361, 18864,  3604,   672,  5232, 16118],
        [    0,     0,     0,     0,     0, 10557, 18906, 19130, 16745,   113,
          7082,   113, 11770,  9083, 11259, 12548,  8641,  3356, 12766,  1365],
        [    0,     0,     0,     0,  2243, 10171,  6361, 17845,  9939, 14128,
          9939, 12703, 13240,  6090, 19822, 18998, 16118,   844, 14929,  4484],
        [    0,     0,     0,     0,     0,     0,     0,     0,     0,  9144,
          1031,  9003,  5264, 18750, 14128, 20058,  6090,  4360, 11299,  4484]])

Sample label size:  torch.Size([10, 20])
Sample label:
 tensor([[0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
        [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [0, 0, 0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4],
        [0, 0, 0, 0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4],
        [0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
        [0, 0, 0, 0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]])
```

In [0]:
```
###LSTM Network Architecture

import torch.nn as nn

class SentimentLSTM(nn.Module):
    """
    The RNN model that will be used to perform Sentiment analysis.
    """

    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers, drop_pr
```

```
        """
        Initialize the model by setting up the layers.
        """
        super().__init__()

        self.output_size = output_size
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim

        # embedding and LSTM layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                            dropout=drop_prob, batch_first=True)

        # dropout layer
        self.dropout = nn.Dropout(0.3)

        # linear and sigmoid layers
        self.fc = nn.Linear(hidden_dim, output_size)
        self.sig = nn.Sigmoid()
        self.soft=nn.Softmax(dim=0)
        self.relu=nn.ReLU()


    def forward(self, x, hidden):
        """
        Perform a forward pass of our model on some input and hidden state.
        """
        batch_size = x.size(0)

        # embeddings and lstm_out
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)
        # stack up lstm outputs
        #lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)
        lstm_out=lstm_out[:, -1, :]
        # dropout and fully-connected layer
        out = self.dropout(lstm_out)
        out=self.relu(out)
        out = self.fc(out)
        # sigmoid function

        #sig_out = self.soft(out)
        sig_out=out
        # reshape to be batch_size first
        sig_out = sig_out.view(batch_size, -1)
        ln= nn.Linear(batch_size, 5)
        #sig_out = sig_out[:, -1] # get last batch of labels

        # return last sigmoid output and hidden state

        return sig_out, hidden
```

```
    def init_hidden(self, batch_size):
        ''' Initializes hidden state '''
        # Create two new tensors with sizes n_layers x batch_size x hidden_dim,
        # initialized to zero, for hidden state and cell state of LSTM
        weight = next(self.parameters()).data

        hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_(),
                      weight.new(self.n_layers, batch_size, self.hidden_dim).zero_())

        return hidden
```

```
In [0]: # Instantiate the model w/ hyperparams
        vocab_size = len(vocab) # +1 for the 0 padding
        output_size = 5
        embedding_dim = 256 # 400 : 33%
        hidden_dim = 256
        n_layers = 4
        net = SentimentLSTM(vocab_size, output_size, embedding_dim, hidden_dim, n_layers)
        print(net)
```

```
SentimentLSTM(
  (embedding): Embedding(20216, 256)
  (lstm): LSTM(256, 256, num_layers=4, batch_first=True, dropout=0.5)
  (dropout): Dropout(p=0.3)
  (fc): Linear(in_features=256, out_features=5, bias=True)
  (sig): Sigmoid()
  (soft): Softmax()
  (relu): ReLU()
)
```

```
In [0]: # loss and optimization functions
        lr=0.001

        criterion = nn.CrossEntropyLoss()

        #nn.BCELoss()
        optimizer = torch.optim.Adam(net.parameters(), lr=lr)
        val_acc=0
        val_acc_list=[]
        loss_list=[]

        # training params

        epochs = 12 # 3-4 is approx where I noticed the validation loss stop decreasing

        counter = 0
        print_every = 100
        clip=5 # gradient clipping
        num_correct = 0


        net.train()
        # train for some number of epochs
```

```
for e in tqdm(range(epochs)):
    # initialize hidden state
    h = net.init_hidden(batch_size)

    # batch loop
    i=0
    for inputs, labels in train_loader:
        counter += 1

             # Creating new variables for the hidden state, otherwise
        # we'd backprop through the entire training history
        h = tuple([each.data for each in h])

        # zero accumulated gradients
        net.zero_grad()

        # get the output from the model
        inputs = inputs.type(torch.LongTensor)
        output,h = net(inputs, h)
        lab=get_label_tensor(labels).to(dtype=torch.long).squeeze()
        loss = criterion(output,  lab)
        loss.backward()
        # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
        nn.utils.clip_grad_norm_(net.parameters(), clip)
        optimizer.step()
        # loss stats
        if counter % print_every == 0:
            # Get validation loss
            val_h = net.init_hidden(batch_size)
            val_losses = []
            net.eval()
            i=0
            for inputs, labels in valid_loader:

                # Creating new variables for the hidden state, otherwise
                # we'd backprop through the entire training history
                val_h = tuple([each.data for each in val_h])
                lab=get_label_tensor(labels).to(dtype=torch.long).squeeze()

                inputs = inputs.type(torch.LongTensor)
                output, val_h = net(inputs, val_h)
                val_loss = criterion(output,lab)

                val_losses.append(val_loss.item())
                #pred = torch.round(output.squeeze())  # rounds to the nearest integer
                #correct_tensor = pred.eq(get_label_tensor(labels).float().squeeze().view_as
                #pred = torch.round(output)
                pred=predicted_v2(output)
                print(pred)
                acc_val=get_accuracy(lab,pred)
                val_acc+=acc_val
                i+=1
            net.train()
            print(acc_val)
```

```python
                print("Epoch: {}/{}...".format(e+1, epochs),
                      "Step: {}...".format(counter),
                      "Loss: {:.6f}...".format(loss.item()),

                      "Val Loss: {:.6f}".format(np.mean(val_losses)))
                print("Val_acc : ",val_acc/i)
                val_acc_list.append(val_acc/i)
                val_acc=0.
        loss_list.append(loss)


                ## modifier label indivi
    print("Val_acc mean: ",np.mean(val_acc_list))
```

In [0]:
```python
import matplotlib.pyplot as plt
```

In [0]:
```python
df = pd.DataFrame({'epoch':[i for i in range(epochs)],'loss':loss_list})

plt.scatter('epoch', 'loss', data=df)
plt.xlabel('epoch ')
plt.ylabel('loss ')
plt.show()
```

In [0]:
```python
df = pd.DataFrame({'epoch':[i for i in range(len(val_losses))],'loss':val_losses})

plt.scatter('epoch', 'loss', data=df)
plt.xlabel('epoch ')
plt.ylabel('loss ')
plt.show()
```

In [0]:
```python
# Get test data loss and accuracy

test_losses = [] # track loss
num_correct = 0.
acc_mean=0.
# init hidden state
h = net.init_hidden(batch_size)

net.eval()
# iterate over test data
i=0
for inputs, labels in test_loader:


    # Creating new variables for the hidden state, otherwise
    # we'd backprop through the entire training history
    h = tuple([each.data for each in h])


    # get predicted outputs
    inputs = inputs.type(torch.LongTensor)
    output, h = net(inputs, h)
    lab=get_label_tensor(labels).to(dtype=torch.long).squeeze()
    out=output.squeeze()
```

```
            # calculate loss
            test_loss = criterion(out,lab)
            test_losses.append(test_loss.item())

            # convert output probabilities to predicted class (0 or 1)
            #pred = torch.round(output.squeeze())  # rounds to the nearest integer



            #pred = torch.round(output)
            pred=predicted_v2(out)
            acc=get_accuracy(lab,pred)
            # compare predictions to true label
            #correct_tensor = pred.eq(get_label_tensor(labels).float().squeeze().view_as(pred))
            #correct =  np.squeeze(correct_tensor.cpu().numpy())
            #num_correct += np.sum(correct)
            acc_mean+=acc
            i+=1

        # -- stats! -- ##
        # avg test loss
        print("Test loss: {:.3f}".format(np.mean(test_losses)))
        #print(acc_mean/i,i,len(test_loader.dataset))
        # accuracy over all test data
        #test_acc =100.* num_correct/len(test_loader.dataset)
        print("Test accuracy: {:.3f}".format(acc_mean/i))
```

In [0]:
```
resultats=[]
```

In [0]:
```
def evaluate(embedding_dim,hidden_dim,n_layers,lr,epochs):
    vocab_size = len(vocab) # +1 for the 0 padding
    net = SentimentLSTM(vocab_size, output_size, embedding_dim, hidden_dim, n_layers)
    # loss and optimization functions

    criterion = nn.CrossEntropyLoss()

    #nn.BCELoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    val_acc=0
    val_acc_list=[]
    loss_list=[]

    # training params

    counter = 0
    print_every = 100
    clip=5 # gradient clipping
    num_correct = 0


    net.train()
    # train for some number of epochs
    for e in tqdm(range(epochs)):
        # initialize hidden state
        h = net.init_hidden(batch_size)
```

```
# batch loop
i=0
for inputs, labels in train_loader:
    counter += 1

         # Creating new variables for the hidden state, otherwise
    # we'd backprop through the entire training history
    h = tuple([each.data for each in h])

    # zero accumulated gradients
    net.zero_grad()

    # get the output from the model
    inputs = inputs.type(torch.LongTensor)
    output,h = net(inputs, h)
    lab=get_label_tensor(labels).to(dtype=torch.long).squeeze()
    loss = criterion(output,  lab)
    loss.backward()
    # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
    nn.utils.clip_grad_norm_(net.parameters(), clip)
    optimizer.step()
    # loss stats
    if counter % print_every == 0:
        # Get validation loss
        val_h = net.init_hidden(batch_size)
        val_losses = []
        net.eval()
        i=0
        for inputs, labels in valid_loader:

            # Creating new variables for the hidden state, otherwise
            # we'd backprop through the entire training history
            val_h = tuple([each.data for each in val_h])
            lab=get_label_tensor(labels).to(dtype=torch.long).squeeze()

            inputs = inputs.type(torch.LongTensor)
            output, val_h = net(inputs, val_h)
            val_loss = criterion(output,lab)

            val_losses.append(val_loss.item())
            #pred = torch.round(output.squeeze())  # rounds to the nearest integer
            #correct_tensor = pred.eq(get_label_tensor(labels).float().squeeze().vie
            #pred = torch.round(output)
            pred=predicted_v2(output)
            acc_val=get_accuracy(lab,pred)
            val_acc+=acc_val
            i+=1
        net.train()

        print("Epoch: {}/{}...".format(e+1, epochs),
              "Step: {}...".format(counter),
              "Loss: {:.6f}...".format(loss.item()),
```

```
                "Val Loss: {:.6f}".format(np.mean(val_losses)))
          print("Val_acc : ",val_acc/i)
          val_acc_list.append(val_acc/i)
          val_acc=0.
    loss_list.append(loss)


          ## modifier label indivi
print("Val_acc mean: ",np.mean(val_acc_list))
df = pd.DataFrame({'epoch':[i for i in range(len(loss_list))],'loss':loss_list})
plt.scatter('epoch', 'loss', data=df)
plt.xlabel('epoch ')
plt.ylabel('loss ')
plt.show()
print("_______________ Test _______________")
    # Get test data loss and accuracy

test_losses = [] # track loss
num_correct = 0.
acc_mean=0.
# init hidden state
h = net.init_hidden(batch_size)

net.eval()
# iterate over test data
i=0
for inputs, labels in test_loader:


    # Creating new variables for the hidden state, otherwise
    # we'd backprop through the entire training history
    h = tuple([each.data for each in h])


    # get predicted outputs
    inputs = inputs.type(torch.LongTensor)
    output, h = net(inputs, h)
    lab=get_label_tensor(labels).to(dtype=torch.long).squeeze()
    out=output.squeeze()
    # calculate loss
    test_loss = criterion(out,lab)
    test_losses.append(test_loss.item())

    # convert output probabilities to predicted class (0 or 1)
    #pred = torch.round(output.squeeze())  # rounds to the nearest integer



    #pred = torch.round(output)
    pred=predicted_v2(out)
    acc=get_accuracy(lab,pred)
    # compare predictions to true label
    #correct_tensor = pred.eq(get_label_tensor(labels).float().squeeze().view_as(pred))
    #correct =  np.squeeze(correct_tensor.cpu().numpy())
```

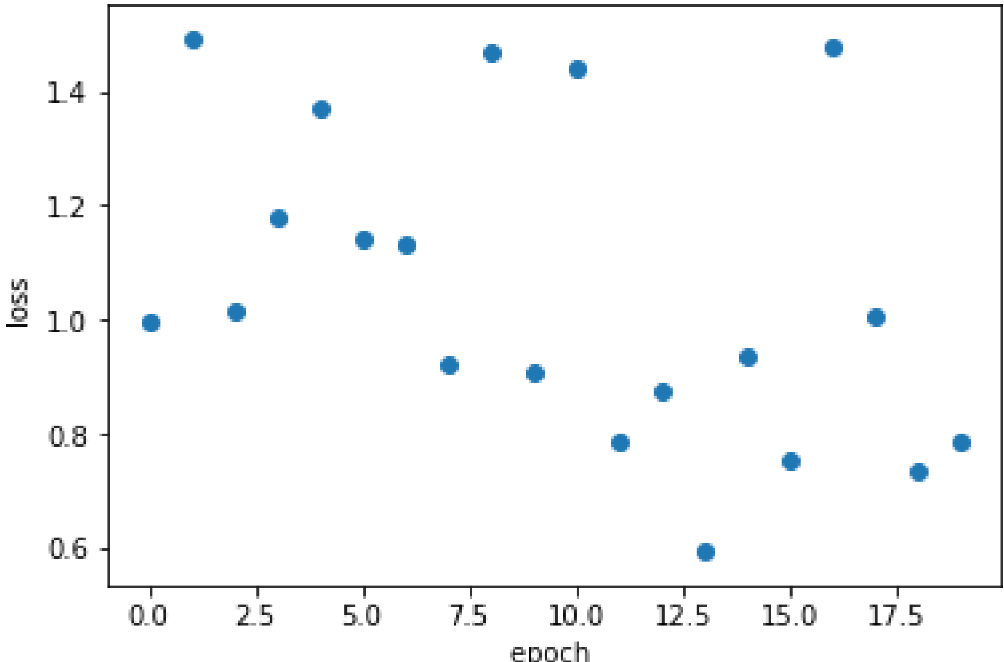
```
            #num_correct += np.sum(correct)
            acc_mean+=acc
            i+=1

        # -- stats! -- ##
        # avg test loss
        print("Test loss: {:.3f}".format(np.mean(test_losses)))
        #print(acc_mean/i,i,len(test_loader.dataset))
        # accuracy over all test data
        #test_acc =100.* num_correct/len(test_loader.dataset)
        print("Test accuracy: {:.3f}".format(acc_mean/i))
        res={"embedding_dim":embedding_dim,"hidden_dim":hidden_dim,"n_layers":n_layers,"lr":lr,"
        resultats.append(res)
        return res
```

```
In [0]: embedding_dim,hidden_dim,n_layers,lr,epochs=256,400,2,0.01,20
```

```
In [0]: evaluate(embedding_dim,hidden_dim,n_layers,lr,epochs)
```

```
  5%|         | 1/20 [00:45<14:15, 45.05s/it]

Epoch: 2/20... Step: 100... Loss: 2.089775... Val Loss: 1.231695
Val_acc :  35.0


 10%|         | 2/20 [01:29<13:30, 45.01s/it]

Epoch: 3/20... Step: 200... Loss: 1.266434... Val Loss: 1.147982
Val_acc :  50.0


 15%|        | 3/20 [02:15<12:46, 45.11s/it]

Epoch: 4/20... Step: 300... Loss: 1.136224... Val Loss: 1.120471
Val_acc :  50.0


 25%|       | 5/20 [03:46<11:19, 45.30s/it]

Epoch: 5/20... Step: 400... Loss: 1.371731... Val Loss: 1.150614
Val_acc :  50.0


 30%|       | 6/20 [04:32<10:36, 45.47s/it]

Epoch: 7/20... Step: 500... Loss: 1.080958... Val Loss: 1.130589
Val_acc :  49.0


 35%|      | 7/20 [05:18<09:54, 45.76s/it]

Epoch: 8/20... Step: 600... Loss: 1.396412... Val Loss: 1.149001
Val_acc :  45.0


 40%|      | 8/20 [06:05<09:13, 46.11s/it]

Epoch: 9/20... Step: 700... Loss: 1.906389... Val Loss: 1.150054
Val_acc :  44.0
```

```
 50%|     | 10/20 [07:41<07:52, 47.24s/it]

Epoch: 10/20... Step: 800... Loss: 0.906824... Val Loss: 1.154532
Val_acc :  39.0


 55%|    | 11/20 [08:31<07:11, 47.90s/it]

Epoch: 12/20... Step: 900... Loss: 1.232759... Val Loss: 1.321471
Val_acc :  34.0


 60%|    | 12/20 [09:21<06:29, 48.74s/it]

Epoch: 13/20... Step: 1000... Loss: 1.510694... Val Loss: 1.344626
Val_acc :  49.0


 65%|   | 13/20 [11:01<07:27, 63.95s/it]

Epoch: 14/20... Step: 1100... Loss: 0.962369... Val Loss: 1.353897
Val_acc :  33.0


 75%|  | 15/20 [13:35<05:52, 70.46s/it]

Epoch: 15/20... Step: 1200... Loss: 0.936237... Val Loss: 1.447068
Val_acc :  43.0


 80%|  | 16/20 [14:47<04:43, 70.89s/it]

Epoch: 17/20... Step: 1300... Loss: 0.907322... Val Loss: 1.445982
Val_acc :  47.0


 85%| | 17/20 [15:58<03:32, 70.83s/it]

Epoch: 18/20... Step: 1400... Loss: 0.808017... Val Loss: 1.668245
Val_acc :  35.0


 90%| | 18/20 [17:15<02:25, 72.66s/it]

Epoch: 19/20... Step: 1500... Loss: 0.930450... Val Loss: 1.821746
Val_acc :  30.0


100%|| 20/20 [19:55<00:00, 76.96s/it]

Epoch: 20/20... Step: 1600... Loss: 0.785834... Val Loss: 1.759309
Val_acc :  43.0
Val_acc mean:  42.25
```




```
--------------- Test ---------------
Test loss: 2.137
Test accuracy: 35.000
```

```
Out[0]: {'Test accuracy': 35.0,
         'Test loss': 2.136674439907074,
         'Val_acc mean ': 42.25,
         'embedding_dim': 256,
         'epochs': 20,
         'hidden_dim': 400,
         'lr': 0.01,
         'n_layers': 2}
```

```
In [0]: def get_resultat():
          with open('resultat_lstm_simple.txt', 'w') as f:
              f.write("%s\n" % resultats)
```

```
In [0]: get_resultat()
```

```
In [0]: resultats
```

```
Out[0]: [{'Test accuracy': 36.0,
          'Test loss': 1.2955604434013366,
          'Val_acc mean ': 48.125,
          'embedding_dim': 7,
          'epochs': 10,
          'hidden_dim': 7,
          'lr': 0.001,
          'n_layers': 7},
```

```
{'Test accuracy': 36.0,
 'Test loss': 1.36827712059021,
 'Val_acc mean ': 40.875,
 'embedding_dim': 20,
 'epochs': 10,
 'hidden_dim': 27,
 'lr': 0.0001,
 'n_layers': 1},
{'Test accuracy': 42.0,
 'Test loss': 1.362808310985565,
 'Val_acc mean ': 22.5,
 'embedding_dim': 20,
 'epochs': 10,
 'hidden_dim': 20,
 'lr': 0.0001,
 'n_layers': 1},
{'Test accuracy': 36.0,
 'Test loss': 1.3388031721115112,
 'Val_acc mean ': 50.0,
 'embedding_dim': 20,
 'epochs': 10,
 'hidden_dim': 20,
 'lr': 0.0001,
 'n_layers': 2},
{'Test accuracy': 40.0,
 'Test loss': 1.768391615152359,
 'Val_acc mean ': 49.25,
 'embedding_dim': 256,
 'epochs': 10,
 'hidden_dim': 400,
 'lr': 0.01,
 'n_layers': 2},
{'Test accuracy': 35.0,
 'Test loss': 2.136674439907074,
 'Val_acc mean ': 42.25,
 'embedding_dim': 256,
 'epochs': 20,
 'hidden_dim': 400,
 'lr': 0.01,
 'n_layers': 2}]
```

B.3 Implémentation : LSTM Child Sum pour la cassification binaire

```
In [0]: #### import libraries

        import numpy as np
        import numpy
        import torch
        from tqdm import tqdm
        import random
        import ast
        import torch.optim as optim
```

```
from sklearn.metrics import confusion_matrix
from  statistics import mean
import seaborn as sns
import pandas as pd
```

In [0]:
```
#### Functions auxliaires


def calculate_evaluation_orders(adjacency_list, tree_size):
    '''Calcule le node_order et le edge_order à partir d'une arborescence adjacency_list
    et de la taille de l'arborescence tree_size. Le modèle TreeLSTM nécessite que
    node_order et edge_order soient transmis au modèle avec les entités de nœud et
    la liste adjacency. Nous pré-calculons ces ordres pour une optimisation de la vitesse.
    '''
    adjacency_list = numpy.array(adjacency_list)

    node_ids = numpy.arange(tree_size, dtype=int)

    node_order = numpy.zeros(tree_size, dtype=int)
    unevaluated_nodes = numpy.ones(tree_size, dtype=bool)

    parent_nodes = adjacency_list[:, 0]
    child_nodes = adjacency_list[:, 1]

    n = 0
    while unevaluated_nodes.any():
        # Find which child nodes have not been evaluated
        unevaluated_mask = unevaluated_nodes[child_nodes]

        # Find the parent nodes of unevaluated children
        unready_parents = parent_nodes[unevaluated_mask]

        # Mark nodes that have not yet been evaluated
        # and which are not in the list of parents with unevaluated child nodes
        nodes_to_evaluate = unevaluated_nodes & ~numpy.isin(node_ids, unready_parents)

        node_order[nodes_to_evaluate] = n
        unevaluated_nodes[nodes_to_evaluate] = False

        n += 1

    edge_order = node_order[parent_nodes]

    return node_order, edge_order



def unbatch_tree_tensor(tensor, tree_sizes):
    '''Convenience functo to unbatch a batched tree tensor into individual tensors given an ar
    sum(tree_sizes) must equal the size of tensor's zeroth dimension.
    '''
    return torch.split(tensor, tree_sizes, dim=0)
```

```
def batch_tree_input(batch):
    '''Combines a batch of tree dictionaries into a single batched dictionary for use by the T
    batch - list of dicts with keys ('features', 'node_order', 'edge_order', 'adjacency_list')
    returns a dict with keys ('features','Labels' , 'node_order', 'edge_order', 'adjacency_list',
    '''
    tree_sizes = [b['Features'].shape[0] for b in batch]

    batched_features = torch.cat([b['Features'] for b in batch])
    batched_labels = torch.cat([b['Labels'] for b in batch])

    batched_node_order = torch.cat([b['node_order'] for b in batch])
    batched_edge_order = torch.cat([b['edge_order'] for b in batch])

    batched_adjacency_list = []
    offset = 0
    for n, b in zip(tree_sizes, batch):
        batched_adjacency_list.append(b['adjacency_list'] + offset)
        offset += n
    batched_adjacency_list = torch.cat(batched_adjacency_list)

    return {
        'Features': batched_features,
        'Labels' : batched_labels ,
        'node_order': batched_node_order,
        'edge_order': batched_edge_order,
        'adjacency_list': batched_adjacency_list,
        'tree_sizes': tree_sizes
    }

### list fo batch
def batch(trees):
  batch=[]
  for tree in trees :
    batch.append(convert_tree_to_tensors(tree))
  return batch

#### batch trees
def batch_tree(trees,batch_size):
  B=[]
  b=batch_size
  i=0
  while (i+1)*b <=len(trees):
    t=trees[i*b:(i+1)*b]
    B.append(batch_tree_input(batch(t)))
    i+=1
  return B



def _label_node_index(node, n=0):
    node['index'] = n
    for child in node['children']:
        n += 1
        _label_node_index(child, n)
```

```
def _gather_node_attributes(node, key):
    features = [node[key]]
    for child in node['children']:
        features.extend(_gather_node_attributes(child, key))
    return features


def _gather_adjacency_list(node):
    adjacency_list = []
    for child in node['children']:
        adjacency_list.append([node['index'], child['index']])
        adjacency_list.extend(_gather_adjacency_list(child))

    return adjacency_list


def convert_tree_to_tensors(tree, device=torch.device('cpu')):
    # Label each node with its walk order to match nodes to feature tensor indexes
    # This modifies the original tree as a side effect
    _label_node_index(tree)

    features = _gather_node_attributes(tree, 'Features')
    labels = _gather_node_attributes(tree, 'Labels')
    adjacency_list = _gather_adjacency_list(tree)

    node_order, edge_order = calculate_evaluation_orders(adjacency_list, len(features))

    return {
        'Features': torch.tensor(features, device=device, dtype=torch.float32),
        'Labels': torch.tensor(labels, device=device, dtype=torch.float32),
        'node_order': torch.tensor(node_order, device=device, dtype=torch.int64),
        'adjacency_list': torch.tensor(adjacency_list, device=device, dtype=torch.int64),
        'edge_order': torch.tensor(edge_order, device=device, dtype=torch.int64),
    }

###################""""
```

```
In [0]: #### Load/Split data :

        def string_to_dict(s):
            return ast.literal_eval(s)

        def id_tree(t,trees):
          if t in trees :
            return trees.index(t)

        #### Split data :
        def diff(first, second):
                return [item for item in first if item not in second]
        def split(trees):
          size=len(trees)
          size_train=int(size*0.6)
```

```
    size_test=int(size*0.2)
    size_dev=size-size_train-size_test

    train_data=random.choices(trees,k=size_train)
    test_data=random.choices(diff(trees,train_data),k=size_test)
    dev_data=random.choices(diff(trees,train_data+test_data),k=size_dev)
    return train_data,test_data,dev_data


  def get_split_data(trees):
    start = time.time()
    train_data,test_data,dev_data=split(trees)
    print("train_data processing ... ")
    with open('train_data.txt', 'w') as f:
      for i in tqdm(range(len(train_data))):
        f.write("%s\n" % train_data[i])
    print("train_data Done!")
    print("test_data processing ... ")
    with open('test_data.txt', 'w') as f:
      for i in tqdm(range(len(test_data))):
        f.write("%s\n" % test_data[i])
    print("test_data Done!")
    print("dev_data processing ... ")
    with open('dev_data.txt', 'w') as f:
      for i in tqdm(range(len(dev_data))):
        f.write("%s\n" % dev_data[i])
    print("dev_data Done! ")
    end = time.time()
    print("temps de SPLIT (s):", end - start)

   #### data loader
  def data_loader():
    temp = open("trees_bin.txt",'r').readlines()
    trees=[string_to_dict(l.strip('\n'+'\t')) for l in temp]
    train = open("train_data.txt",'r').readlines()
    train_data=[string_to_dict(l.strip('\n'+'\t')) for l in train]
    test= open("test_data.txt",'r').readlines()
    test_data=[string_to_dict(l.strip('\n'+'\t')) for l in test]
    dev=open("dev_data.txt",'r').readlines()
    dev_data=[string_to_dict(l.strip('\n'+'\t')) for l in dev]
    return trees,train_data,test_data,dev_data

  trees,train_data,test_data,dev_data=data_loader()
```

In [0]:
```
#### Structure LSTM CHILD SU

class TreeLSTM(torch.nn.Module):
    '''PyTorch TreeLSTM model that implements efficient batching.
    '''
    def __init__(self, in_features, out_features):
        '''TreeLSTM class initializer
        Takes in int sizes of in_features and out_features and sets up model Linear network
        '''
        super().__init__()
```

```
        self.in_features = in_features
        self.out_features = out_features

        # bias terms are only on the W layers for efficiency
        self.W_iou = torch.nn.Linear(self.in_features, 3 * self.out_features)
        self.U_iou = torch.nn.Linear(self.out_features, 3 * self.out_features, bias=False)

        # f terms are maintained seperate from the iou terms because they involve sums over
        # while the iou terms do not
        self.W_f = torch.nn.Linear(self.in_features, self.out_features)
        self.U_f = torch.nn.Linear(self.out_features, self.out_features, bias=False)
          # embedding



    def forward(self, features, node_order, adjacency_list, edge_order):
        '''Run TreeLSTM model on a tree data structure with node features
        Takes Tensors encoding node features, a tree node adjacency_list, and the order in w
        the tree processing should proceed in node_order and edge_order.
        '''

        # Total number of nodes in every tree in the batch
        batch_size = node_order.shape[0]

        # Retrive device the model is currently loaded on to generate h, c, and h_sum result
        device = next(self.parameters()).device

        # h and c states for every node in the batch
        h = torch.zeros(batch_size, self.out_features, device=device)
        c = torch.zeros(batch_size, self.out_features, device=device)

        # h_sum storage buffer
        h_sum = torch.zeros(batch_size, self.out_features, device=device)
        soft=torch.nn.Softmax(dim=1)
        logsoft=torch.nn.LogSoftmax(dim=1)
        fc = torch.nn.Linear( self.out_features,2)
        # populate the h and c states respecting computation order
        for n in range(node_order.max() + 1):
            self._run_lstm(n, h, c, h_sum, features, node_order, adjacency_list, edge_order)

        return torch.sigmoid(h), c

    def _run_lstm(self, iteration, h, c, h_sum, features, node_order, adjacency_list, edge_o
        '''Helper function to evaluate all tree nodes currently able to be evaluated.
        '''
        # N is the number of nodes in the tree
        # n is the number of nodes to be evaluated on in the current iteration
        # E is the number of edges in the tree
        # e is the number of edges to be evaluated on in the current iteration
        # F is the number of features in each node
        # M is the number of hidden neurons in the network

        # node_order is a tensor of size N x 1
        # edge_order is a tensor of size E x 1
```

```
# features is a tensor of size N x F
# adjacency_list is a tensor of size E x 2

# node_mask is a tensor of size N x 1
node_mask = node_order == iteration
# edge_mask is a tensor of size E x 1
edge_mask = edge_order == iteration

# x is a tensor of size n x F
x = features[node_mask, :]
# At iteration 0 none of the nodes should have children
# Otherwise, select the child nodes needed for current iteration
# and sum over their hidden states
if iteration > 0:
    # adjacency_list is a tensor of size e x 2
    adjacency_list = adjacency_list[edge_mask, :]

    # parent_indexes and child_indexes are tensors of size e x 1
    # parent_indexes and child_indexes contain the integer indexes needed to index i
    # the feature and hidden state arrays to retrieve the data for those parent/chil
    parent_indexes = adjacency_list[:, 0]
    child_indexes = adjacency_list[:, 1]

    # child_h and child_c are tensors of size e x 1
    child_h = h[child_indexes, :]
    child_c = c[child_indexes, :]

    # Add child hidden states to parent offset locations
    h_sum[parent_indexes, :] += h[child_indexes, :]

# i, o and u are tensors of size n x M
iou = self.W_iou(x) + self.U_iou(h_sum[node_mask, :])
i, o, u = torch.split(iou, iou.size(1) // 3, dim=1)
i = torch.sigmoid(i)
o = torch.sigmoid(o)
u = torch.tanh(u)

c[node_mask, :] = i * u

# At iteration 0 none of the nodes should have children
# Otherwise, calculate the forget states for each parent node and child node
# and sum over the child memory cell states
if iteration > 0:
    # f is a tensor of size e x M
    f = self.W_f(features[parent_indexes, :]) + self.U_f(child_h)
    f = torch.sigmoid(f)

    # fc is a tensor of size e x M
    fc = f * child_c

    # Add the calculated f values to the parent's memory cell state
    c[parent_indexes, :] += fc

h[node_mask, :] = o * torch.tanh(c[node_mask])
```

In [0]:
```
def emmb(f,emmb_dim):
  fc=torch.nn.Linear(1,emmb_dim)
  emmb_features=[]
  for i in range(len(f)):
    emmb_features.append(fc(f[i]).tolist())
  return torch.tensor(emmb_features)
```

In [0]:
```
######################
##   Training
######################
emmb_dim=200
net=TreeLSTM(emmb_dim,2)     ##

#loss_function = torch.nn.BCELoss()
loss_function = torch.nn.CrossEntropyLoss()

lr=0.001
optimizer = optim.Adam(net.parameters(), lr=lr)
#optimizer = optim.SGD(net.parameters(), lr=lr)

batch_size=10
nbr_epoch=12

epoch_list=[epoch for epoch in range(1,nbr_epoch+1)]
loss_list=[]
acc_class=[]
predicted_list=[]
net.train()
data_load=batch_tree(trees,batch_size)

for epoch in tqdm(range(1,nbr_epoch+1)):
  min_loss=0

  for data in data_load:

        # zero accumulated gradients
        optimizer.zero_grad()

        h, c = net(
            emmb(data['Features'],emmb_dim),
            data['node_order'],
            data['adjacency_list'],
            data['edge_order']
        )
        labels = data['Labels']
        acc_class.append(get_accuracy_class(h,labels))
        predicted_list.append(predicted(h))
        loss = loss_function(h, labels.to(dtype=torch.long).squeeze_())
        min_loss+=loss.item()/int((len(data_load)))
        loss.backward()
        optimizer.step()
  loss_list.append(min_loss)
```

```
### h moyenne
```

```
100%|| 12/12 [10:59<00:00, 49.90s/it]
```

```
In [0]: print(loss_list)
```

```
[0.6904570972796774, 0.6871611453518596, 0.6855793512348198, 0.6871385624924804, 0.6836520318882782,
```

```
In [0]: def get_hidden(h):
          l=[]
          for i in range(len(h)):
            for j in range(len(h[i])):
               if h[i][j]>= 0.5:
                   l.append([j])
          return torch.FloatTensor(l)
```

```
In [0]: import matplotlib.pyplot as plt

        df = pd.DataFrame({'epoch':epoch_list,'loss':loss_list})

        plt.scatter('epoch', 'loss', data=df)
        plt.xlabel('epoch ')
        plt.ylabel('loss ')
        plt.show()
```



```
In [0]: len(loss_list)
```

```
Out[0]: 12
```

```
In [0]: # Get test data loss and accuracy

        test_losses = [] # track loss
        acc=[]
        net.eval()
        data_load=batch_tree(trees,batch_size)
        for data in data_load[:800]:
                  h, c= net(
                    emmb(data['Features'],emmb_dim),
                    data['node_order'],
                    data['adjacency_list'],
                    data['edge_order'])
                  labels = data['Labels']
                  test_loss = loss_function(h,labels.to(dtype=torch.long).squeeze_())
                  #test_loss = loss_function(h,labels.float().squeeze_())
                  test_losses.append(test_loss.item())
                  acc.append(get_accuracy(h,labels))
        # -- stats! -- ##
        # avg test loss
        print("Test loss: {:.3f}".format(np.mean(test_losses)))

        # accuracy over all test data
        print("Test accuracy:",np.mean(acc))
```

```
Test loss: 0.680
Test accuracy: 79.44203038871288
```

```
In [0]: ####### TREES MIN

        def frequency(l):
            cl=[0.,1.,2.,3.,4.]
            count=[0,0,0,0,0]
            for i in range(len(l)) :
              if l[i]==0. :
                    count[0]+=1
              elif l[i]==1.:
                    count[1]+=1
              elif l[i]==2.:
                    count[2]+=1
              elif l[i]==3.:
                    count[3]+=1
              elif l[i]==4.:
                    count[4]+=1
            return np.argmax(np.array(count))
        def get_trees_min(trees):
            index_tree_class=[]
            tree=batch_tree(trees,1)
            for i in range(len(tree)):
                label_i=tree[i]['Labels']
                c_i=frequency(label_i)
                if c_i!=2 :
                    index_tree_class.append([i,c_i])
            return  index_tree_class
```

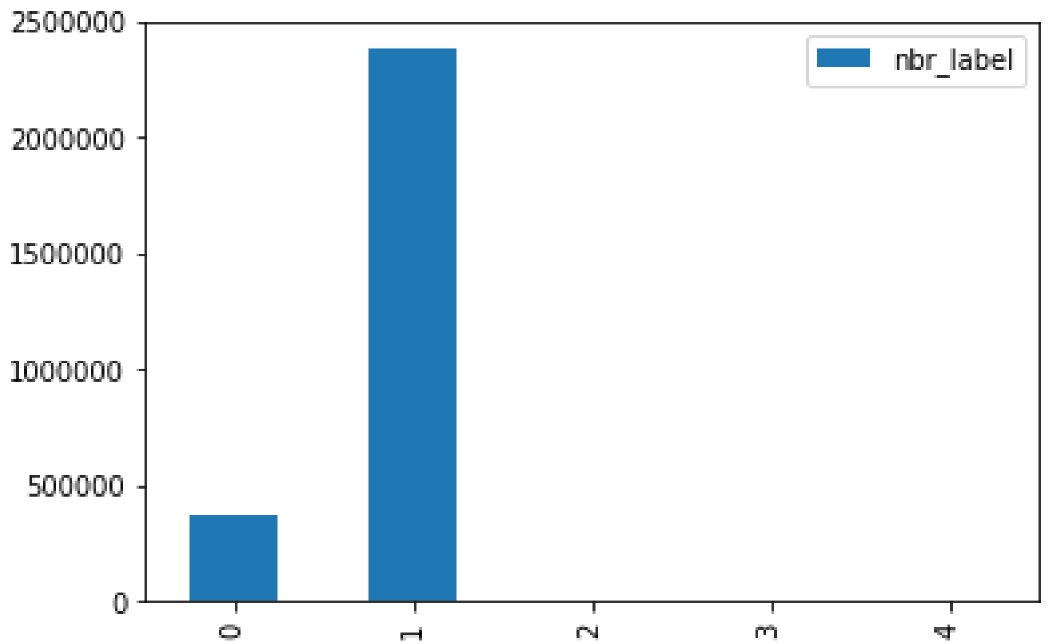
```
def get_tree(trees):
  trees_min=get_trees_min(trees)
  trees_min_list=[]
  for i,j in trees_min :
    trees_min_list.append(trees[i])
  return trees_min_list


def save_trees_min(N=220):
  with open('trees_min.txt', 'w') as f:
    for i in tqdm(range(1,N)):
      f.write("%s\n" % get_tree(trees))
```

```
In [0]: #### Evaluation
        ### Fonctions auxiliares
        def argmax(t):
          v,i=torch.max(t,0)
          return i.item()
        def predicted(h):
              t=[]
              for i in range(len(h)):
                t.append([argmax(h[i])])

              return torch.FloatTensor(t)
        def get_accuracy(h,labels):
              total = len(labels)
              correct = (predicted(h) == labels).float().sum()
              accuracy = 100.*correct / total

              return accuracy.item()
        def get_confusion_matrix(h,labels):
          H=predicted(h)
          H=H.tolist()
          labels=labels.tolist()
          c=confusion_matrix(labels, H,labels=[0,1,2,3,4])
          return c
        def get_accuracy_class(h,labels):
          c=get_confusion_matrix(h,labels)
          acc=[score_class(c,i) for i in range(0,5)]
          return acc

        def score_class(c,i):
          if c[i,i]==0:
            return 0
          else :
               acc=c[i,i]/c[:,i].sum()
               return acc*100
        def get_mean(ev):
          mean=np.array([0.,0.,0.,0.,0.])
          for i in range(len(ev)):
            mean+=np.array(ev[i])
          return mean/len(ev)
```

```python
## evaluation d'un arbre
def eval(data):
  data=convert_tree_to_tensors(data)
  model_eval=model_train.eval()
  h_ev, c_ev = model_eval(
            data['Features'],
            data['node_order'],
            data['adjacency_list'],
            data['edge_order']
        )
  labels = data['Labels']
  loss_ev = loss_function(h_ev, labels.to(dtype=torch.long).squeeze_())
  return {"org_labels":labels,"predicted_labels": predicted(h_ev) , "accuracy":get_accuracy(l

## evaluation sur un ensemble de données
def eval_data(data_set):
    acc_list=np.array([eval(data_set[i])["accuracy"] for i in range(len(data_set))])
    return np.mean(acc_list)



### get distribution of labels
def distribution(l):
    cl=[0.,1.,2.,3.,4.]
    count=[0,0,0,0,0]
    for j in range(len(l)):
        for i in range(len(l[j])) :
            if l[j][i]==0. :
               count[0]+=1
            elif l[j][i]==1.:
               count[1]+=1
            elif l[j][i]==2.:
                count[2]+=1
            elif l[j][i]==3.:
                count[3]+=1
            elif l[j][i]==4.:
                count[4]+=1
    df = pd.DataFrame({'nbr_label':count})
    df.plot.bar()
    return count
```

In [0]:
```python
distribution(predicted_list)
```

Out[0]: [375466, 2379314, 0, 0, 0]



B.4 Implémentation : LSTM child sum multiclass

```
In [0]: #### import libraries

        import numpy as np
        import numpy
        import torch
        from tqdm import tqdm
        import random
        import ast
        import torch.optim as optim
        from sklearn.metrics import confusion_matrix
        from  statistics import mean
        import seaborn as sns
        import matplotlib.pyplot as plt

        import pandas as pd
```

```
In [0]: ####  Functions Helper


        def calculate_evaluation_orders(adjacency_list, tree_size):
            '''Calculates the node_order and edge_order from a tree adjacency_list and the tree_size.
            The TreeLSTM model requires node_order and edge_order to be passed into the model along
            with the node features and adjacency_list.  We pre-calculate these orders as a speed
            optimization.
            '''
            adjacency_list = numpy.array(adjacency_list)

            node_ids = numpy.arange(tree_size, dtype=int)
```

```
    node_order = numpy.zeros(tree_size, dtype=int)
    unevaluated_nodes = numpy.ones(tree_size, dtype=bool)

    parent_nodes = adjacency_list[:, 0]
    child_nodes = adjacency_list[:, 1]

    n = 0
    while unevaluated_nodes.any():
        # Find which child nodes have not been evaluated
        unevaluated_mask = unevaluated_nodes[child_nodes]

        # Find the parent nodes of unevaluated children
        unready_parents = parent_nodes[unevaluated_mask]

        # Mark nodes that have not yet been evaluated
        # and which are not in the list of parents with unevaluated child nodes
        nodes_to_evaluate = unevaluated_nodes & ~numpy.isin(node_ids, unready_parents)

        node_order[nodes_to_evaluate] = n
        unevaluated_nodes[nodes_to_evaluate] = False

        n += 1

    edge_order = node_order[parent_nodes]

    return node_order, edge_order



def unbatch_tree_tensor(tensor, tree_sizes):
    '''Convenience functo to unbatch a batched tree tensor into individual tensors given an ar
    sum(tree_sizes) must equal the size of tensor's zeroth dimension.
    '''
    return torch.split(tensor, tree_sizes, dim=0)


def batch_tree_input(batch):
    '''Combines a batch of tree dictionaries into a single batched dictionary for use by the T
    batch - list of dicts with keys ('features', 'node_order', 'edge_order', 'adjacency_list')
    returns a dict with keys ('features','Labels' , 'node_order', 'edge_order', 'adjacency_list',
    '''
    tree_sizes = [b['Features'].shape[0] for b in batch]

    batched_features = torch.cat([b['Features'] for b in batch])
    batched_labels = torch.cat([b['Labels'] for b in batch])

    batched_node_order = torch.cat([b['node_order'] for b in batch])
    batched_edge_order = torch.cat([b['edge_order'] for b in batch])

    batched_adjacency_list = []
    offset = 0
    for n, b in zip(tree_sizes, batch):
        batched_adjacency_list.append(b['adjacency_list'] + offset)
        offset += n
```

```
    batched_adjacency_list = torch.cat(batched_adjacency_list)

    return {
        'Features': batched_features,
        'Labels' : batched_labels ,
        'node_order': batched_node_order,
        'edge_order': batched_edge_order,
        'adjacency_list': batched_adjacency_list,
        'tree_sizes': tree_sizes
    }

### list fo batch
def batch(trees):
  batch=[]
  for tree in trees :
    batch.append(convert_tree_to_tensors(tree))
  return batch

#### batch trees
def batch_tree(trees,batch_size):
  B=[]
  b=batch_size
  i=0
  while (i+1)*b <=len(trees):
    t=trees[i*b:(i+1)*b]
    B.append(batch_tree_input(batch(t)))
    i+=1
  return B



def _label_node_index(node, n=0):
    node['index'] = n
    for child in node['children']:
        n += 1
        _label_node_index(child, n)


def _gather_node_attributes(node, key):
    features = [node[key]]
    for child in node['children']:
        features.extend(_gather_node_attributes(child, key))
    return features


def _gather_adjacency_list(node):
    adjacency_list = []
    for child in node['children']:
        adjacency_list.append([node['index'], child['index']])
        adjacency_list.extend(_gather_adjacency_list(child))

    return adjacency_list
```

```
def convert_tree_to_tensors(tree, device=torch.device('cpu')):
    # Label each node with its walk order to match nodes to feature tensor indexes
    # This modifies the original tree as a side effect
    _label_node_index(tree)

    features = _gather_node_attributes(tree, 'Features')
    labels = _gather_node_attributes(tree, 'Labels')
    adjacency_list = _gather_adjacency_list(tree)

    node_order, edge_order = calculate_evaluation_orders(adjacency_list, len(features))

    return {
        'Features': torch.tensor(features, device=device, dtype=torch.float32),
        'Labels': torch.tensor(labels, device=device, dtype=torch.float32),
        'node_order': torch.tensor(node_order, device=device, dtype=torch.int64),
        'adjacency_list': torch.tensor(adjacency_list, device=device, dtype=torch.int64),
        'edge_order': torch.tensor(edge_order, device=device, dtype=torch.int64),
    }

###################""""
```

In [0]:
```
#### Load/Split data :

def string_to_dict(s):
    return ast.literal_eval(s)

def id_tree(t,trees):
  if t in trees :
    return trees.index(t)

#### Split data :
def diff(first, second):
        return [item for item in first if item not in second]
def split(trees):
  size=len(trees)
  size_train=int(size*0.6)
  size_test=int(size*0.2)
  size_dev=size-size_train-size_test

  train_data=random.choices(trees,k=size_train)
  test_data=random.choices(diff(trees,train_data),k=size_test)
  dev_data=random.choices(diff(trees,train_data+test_data),k=size_dev)
  return train_data,test_data,dev_data


def get_split_data(trees):
  start = time.time()
  train_data,test_data,dev_data=split(trees)
  print("train_data processing ... ")
  with open('train_data.txt', 'w') as f:
    for i in tqdm(range(len(train_data))):
      f.write("%s\n" % train_data[i])
  print("train_data Done!")
  print("test_data processing ... ")
```

```
  with open('test_data.txt', 'w') as f:
    for i in tqdm(range(len(test_data))):
      f.write("%s\n" % test_data[i])
  print("test_data Done!")
  print("dev_data processing ... ")
  with open('dev_data.txt', 'w') as f:
    for i in tqdm(range(len(dev_data))):
      f.write("%s\n" % dev_data[i])
  print("dev_data Done! ")
  end = time.time()
  print("temps de SPLIT (s):", end - start)

 #### data loader
"""def data_loader():
  temp = open("trees.txt",'r').readlines()
  trees=[string_to_dict(l.strip('\n'+'\t')) for l in temp]
  train = open("train_data.txt",'r').readlines()
  train_data=[string_to_dict(l.strip('\n'+'\t')) for l in train]
  test= open("test_data.txt",'r').readlines()
  test_data=[string_to_dict(l.strip('\n'+'\t')) for l in test]
  dev=open("dev_data.txt",'r').readlines()
  dev_data=[string_to_dict(l.strip('\n'+'\t')) for l in dev]
  return trees,train_data,test_data,dev_data

trees,train_data,test_data,dev_data=data_loader()"""

def data_loader():

  train = open("data_lstm_coc.txt",'r').readlines()
  train_data=[string_to_dict(l.strip('\n'+'\t')) for l in train][:10]
  temp = open("vocab.txt", 'r').readlines()
  vocab = [l.strip('\n' + '\t') for l in temp]

  return train_data,vocab

data,vocab=data_loader()
```

In [0]:

In [0]:

```
#### Structure LSTM CHILD SU

class TreeLSTM(torch.nn.Module):
    '''PyTorch TreeLSTM model that implements efficient batching.
    '''
    def __init__(self, in_features, out_features):
        '''TreeLSTM class initializer
        Takes in int sizes of in_features and out_features and sets up model Linear network
        '''
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features

        # bias terms are only on the W layers for efficiency
        self.W_iou = torch.nn.Linear(self.in_features, 3 * self.out_features)
        self.U_iou = torch.nn.Linear(self.out_features, 3 * self.out_features, bias=False)
```

```
        # f terms are maintained seperate from the iou terms because they involve sums over
        # while the iou terms do not
        self.W_f = torch.nn.Linear(self.in_features, self.out_features)
        self.U_f = torch.nn.Linear(self.out_features, self.out_features, bias=False)
          # embedding



    def forward(self, features, node_order, adjacency_list, edge_order):
        '''Run TreeLSTM model on a tree data structure with node features
        Takes Tensors encoding node features, a tree node adjacency_list, and the order in w
        the tree processing should proceed in node_order and edge_order.
        '''

        # Total number of nodes in every tree in the batch
        batch_size = node_order.shape[0]

        # Retrive device the model is currently loaded on to generate h, c, and h_sum result
        device = next(self.parameters()).device

        # h and c states for every node in the batch
        h = torch.zeros(batch_size, self.out_features, device=device)
        c = torch.zeros(batch_size, self.out_features, device=device)

        # h_sum storage buffer
        h_sum = torch.zeros(batch_size, self.out_features, device=device)
        soft=torch.nn.Softmax(dim=1)
        logsoft=torch.nn.LogSoftmax(dim=1)
        fc = torch.nn.Linear( self.out_features,2)
        # populate the h and c states respecting computation order
        for n in range(node_order.max() + 1):
            self._run_lstm(n, h, c, h_sum, features, node_order, adjacency_list, edge_order)

        return torch.sigmoid(h), c

    def _run_lstm(self, iteration, h, c, h_sum, features, node_order, adjacency_list, edge_o
        '''Helper function to evaluate all tree nodes currently able to be evaluated.
        '''
        # N is the number of nodes in the tree
        # n is the number of nodes to be evaluated on in the current iteration
        # E is the number of edges in the tree
        # e is the number of edges to be evaluated on in the current iteration
        # F is the number of features in each node
        # M is the number of hidden neurons in the network

        # node_order is a tensor of size N x 1
        # edge_order is a tensor of size E x 1
        # features is a tensor of size N x F
        # adjacency_list is a tensor of size E x 2

        # node_mask is a tensor of size N x 1
        node_mask = node_order == iteration
        # edge_mask is a tensor of size E x 1
```

```
            edge_mask = edge_order == iteration

            # x is a tensor of size n x F
            x = features[node_mask, :]
            # At iteration 0 none of the nodes should have children
            # Otherwise, select the child nodes needed for current iteration
            # and sum over their hidden states
            if iteration > 0:
                # adjacency_list is a tensor of size e x 2
                adjacency_list = adjacency_list[edge_mask, :]

                # parent_indexes and child_indexes are tensors of size e x 1
                # parent_indexes and child_indexes contain the integer indexes needed to index i
                # the feature and hidden state arrays to retrieve the data for those parent/chil
                parent_indexes = adjacency_list[:, 0]
                child_indexes = adjacency_list[:, 1]

                # child_h and child_c are tensors of size e x 1
                child_h = h[child_indexes, :]
                child_c = c[child_indexes, :]

                # Add child hidden states to parent offset locations
                h_sum[parent_indexes, :] += h[child_indexes, :]

            # i, o and u are tensors of size n x M
            iou = self.W_iou(x) + self.U_iou(h_sum[node_mask, :])
            i, o, u = torch.split(iou, iou.size(1) // 3, dim=1)
            i = torch.sigmoid(i)
            o = torch.sigmoid(o)
            u = torch.tanh(u)

            c[node_mask, :] = i * u

            # At iteration 0 none of the nodes should have children
            # Otherwise, calculate the forget states for each parent node and child node
            # and sum over the child memory cell states
            if iteration > 0:
                # f is a tensor of size e x M
                f = self.W_f(features[parent_indexes, :]) + self.U_f(child_h)
                f = torch.sigmoid(f)

                # fc is a tensor of size e x M
                fc = f * child_c

                # Add the calculated f values to the parent's memory cell state
                c[parent_indexes, :] += fc

            h[node_mask, :] = o * torch.tanh(c[node_mask])
```

```
In [0]: # Foncion d'Embedding
        def emmb(f,emmb_dim):
          fc=torch.nn.Linear(1,emmb_dim)
          emmb_features=[]
          for i in range(len(f)):
```

```
      emmb_features.append(fc(f[i]).tolist())
    return torch.tensor(emmb_features)
```

```
In [0]: #### Evaluation
        ### Fonctions auxiliares
        def argmax(t):
          v,i=torch.max(t,0)
          return i.item()
        def predicted(h):
              t=[]
              for i in range(len(h)):
                t.append([argmax(h[i])])

              return torch.FloatTensor(t)
        def get_accuracy(h,labels):
              total = len(labels)
              correct = (predicted(h) == labels).float().sum()
              accuracy = 100.*correct / total

              return accuracy.item()
        def get_confusion_matrix(h,labels):
          H=predicted(h)
          H=H.tolist()
          labels=labels.tolist()
          c=confusion_matrix(labels, H,labels=[0,1,2,3,4])
          return c
        def get_accuracy_class(h,labels):
          c=get_confusion_matrix(h,labels)
          acc=[score_class(c,i) for i in range(0,5)]
          return acc

        def score_class(c,i):
          if c[i,i]==0:
            return 0
          else :
               acc=c[i,i]/c[:,i].sum()
               return acc*100
        def get_mean(ev):
          mean=np.array([0.,0.,0.,0.,0.])
          for i in range(len(ev)):
            mean+=np.array(ev[i])
          return mean/len(ev)


        ## evaluation d'un arbre
        def eval(data):
          data=convert_tree_to_tensors(data)
          model_eval=model_train.eval()
          h_ev, c_ev = model_eval(
                    data['Features'],
                    data['node_order'],
                    data['adjacency_list'],
                    data['edge_order']
```

```
        )
    labels = data['Labels']
    loss_ev = loss_function(h_ev, labels.to(dtype=torch.long).squeeze_())
    return {"org_labels":labels,"predicted_labels": predicted(h_ev) , "accuracy":get_accuracy(l

## evaluation sur un ensemble de données
def eval_data(data_set):
    acc_list=np.array([eval(data_set[i])["accuracy"] for i in range(len(data_set))])
    return np.mean(acc_list)



### get distribution of labels
def distribution(l):
    cl=[0.,1.,2.,3.,4.]
    count=[0,0,0,0,0]
    for j in range(len(l)):
        for i in range(len(l[j])) :
            if l[j][i]==0. :
               count[0]+=1
            elif l[j][i]==1.:
               count[1]+=1
            elif l[j][i]==2.:
                count[2]+=1
            elif l[j][i]==3.:
                count[3]+=1
            elif l[j][i]==4.:
                count[4]+=1
    df = pd.DataFrame({'nbr_label':count})
    df.plot.bar()
    return count
```

In [0]:

In [0]:

```
######################
##   Training
######################
emmb_dim=1
nbr_classes=len(vocab)
# 7: 21% ,
net=TreeLSTM(emmb_dim,nbr_classes)    ##

#loss_function = torch.nn.BCELoss()
loss_function = torch.nn.CrossEntropyLoss()

lr=0.001
optimizer = optim.Adam(net.parameters(), lr=lr)
#optimizer = optim.SGD(net.parameters(), lr=lr)

batch_size=4
nbr_epoch=12

epoch_list=[epoch for epoch in range(1,nbr_epoch+1)]
loss_list=[]
acc_class=[]
```

```
    predicted_list=[]
    net.train()
    data_load=batch_tree(data,batch_size)

    for epoch in tqdm(range(1,nbr_epoch+1)):
      min_loss=0

      for data in data_load[:2]:

            # zero accumulated gradients
            optimizer.zero_grad()

            h, c = net(
                #emmb(data['Features'],emmb_dim),
                data['Features'],
                data['node_order'],
                data['adjacency_list'],
                data['edge_order']
            )
            labels = data['Labels']
            acc_class.append(get_accuracy_class(h,labels))
            predicted_list.append(predicted(h))
            loss = loss_function(h, labels.to(dtype=torch.long).squeeze_())
            min_loss+=loss.item()/int((len(data_load)))
            loss.backward()
            optimizer.step()
      loss_list.append(min_loss)

    ### h moyenne
```

```
  0%|          | 0/12 [00:00<?, ?it/s]
```

In [0]:
```
resultats=[]
```

In [0]:
```
def evaluate(emmb_dim,batch_size,nbr_epoch,lr):
    ######################
    ##   Training
    ######################

    # 7: 21% ,
    net=TreeLSTM(emmb_dim,5)    ##

    #loss_function = torch.nn.BCELoss()
    loss_function = torch.nn.CrossEntropyLoss()


    optimizer = optim.Adam(net.parameters(), lr=lr)
    #optimizer = optim.SGD(net.parameters(), lr=lr)

    epoch_list=[epoch for epoch in range(1,nbr_epoch+1)]
    loss_list=[]
    acc_class=[]
    predicted_list=[]
    net.train()
    data_load=batch_tree(trees,batch_size)
```

```
for epoch in tqdm(range(1,nbr_epoch+1)):
  min_loss=0

  for data in data_load:

        # zero accumulated gradients
        optimizer.zero_grad()

        h, c = net(
            emmb(data['Features'],emmb_dim),
            data['node_order'],
            data['adjacency_list'],
            data['edge_order']
        )
        labels = data['Labels']
        acc_class.append(get_accuracy_class(h,labels))
        predicted_list.append(predicted(h))
        loss = loss_function(h, labels.to(dtype=torch.long).squeeze_())
        min_loss+=loss.item()/int((len(data_load)))
        loss.backward()
        optimizer.step()
  loss_list.append(min_loss)

df = pd.DataFrame({'epoch':epoch_list,'loss':loss_list})

plt.scatter('epoch', 'loss', data=df)
plt.xlabel('epoch ')
plt.ylabel('loss ')
plt.show()
print("_________________ Test _______________")
# Get test data loss and accuracy

test_losses = [] # track loss
acc=[]
net.eval()
data_load=batch_tree(trees,batch_size)
for data in data_load[:800]:
          h, c= net(
            emmb(data['Features'],emmb_dim),
            data['node_order'],
            data['adjacency_list'],
            data['edge_order'])
          labels = data['Labels']
          test_loss = loss_function(h,labels.to(dtype=torch.long).squeeze_())
          #test_loss = loss_function(h,labels.float().squeeze_())
          test_losses.append(test_loss.item())
          acc.append(get_accuracy(h,labels))
# -- stats! -- ##
# avg test loss
print("Test loss: {:.3f}".format(np.mean(test_losses)))

# accuracy over all test data
print("Test accuracy:",np.mean(acc))
```

```
        res={"embedding_dim":emmb_dim,"lr":lr,"batch_size":batch_size,"epochs":nbr_epoch,"Test l
        resultats.append(res)
        return resultats
```

In [0]:
```
emmb_dim,batch_size,nbr_epoch,lr=7,2,10,0.001
```

In [0]:
```
 evaluate(emmb_dim,batch_size,nbr_epoch,lr)
```

In [0]:
```
resultats
```

In [0]:
```
# Get test data loss and accuracy

test_losses = [] # track loss
acc=[]
net.eval()
data_load=batch_tree(trees,batch_size)
for data in data_load[:800]:
          h, c= net(
            emmb(data['Features'],emmb_dim),
            data['node_order'],
            data['adjacency_list'],
            data['edge_order'])
          labels = data['Labels']
          test_loss = loss_function(h,labels.to(dtype=torch.long).squeeze_())
          #test_loss = loss_function(h,labels.float().squeeze_())
          test_losses.append(test_loss.item())
          acc.append(get_accuracy(h,labels))
# -- stats! -- ##
# avg test loss
print("Test loss: {:.3f}".format(np.mean(test_losses)))

# accuracy over all test data
print("Test accuracy:",np.mean(acc))
```

In [0]:
```
####### TREES MIN

def frequency(l):
    cl=[0.,1.,2.,3.,4.]
    count=[0,0,0,0,0]
    for i in range(len(l)) :
      if l[i]==0. :
            count[0]+=1
      elif l[i]==1.:
            count[1]+=1
      elif l[i]==2.:
            count[2]+=1
      elif l[i]==3.:
            count[3]+=1
      elif l[i]==4.:
            count[4]+=1
    return np.argmax(np.array(count))
def get_trees_min(trees):
    index_tree_class=[]
    tree=batch_tree(trees,1)
    for i in range(len(tree)):
```

```
        label_i=tree[i]['Labels']
        c_i=frequency(label_i)
        if c_i!=2 :
            index_tree_class.append([i,c_i])
    return  index_tree_class

def get_tree(trees):
  trees_min=get_trees_min(trees)
  trees_min_list=[]
  for i,j in trees_min :
    trees_min_list.append(trees[i])
  return trees_min_list


def save_trees_min(N=220):
  with open('trees_min.txt', 'w') as f:
    for i in tqdm(range(1,N)):
      f.write("%s\n" % get_tree(trees))
```

```
In [0]: l=batch_tree_input(batch(trees))['Labels']
        l
```

```
In [0]: distribution(predicted_list)
```

B.5 Parsing Coco Data set

```
In [0]: from tqdm import tqdm
        import torch
        from nltk import *
        from nltk import Tree
        import spacy
        nlp = spacy.load('en')
        from spacy import displacy

        from string import digits ##text
        import re
        import pandas as pd
        import csv
        import torch.nn as nn
        import ast
```

```
In [0]: def get_data():
            ## Upload datasetSentences.txt
            temp = open("answers.txt", 'r').readlines()
            answers = [l.strip('\n' + '\t') for l in temp]
            ## Upload sentiment_labels.txt
            temp2 = open("questions.txt", 'r').readlines()
            questions = [l.strip('\n' + '\t') for l in temp2]
            ### Upload vocabulary to csv
            temp3 = open("vocab.txt", 'r').readlines()
            vocab = [l.strip('\n' + '\t') for l in temp3]
            return questions,answers,vocab
```

```
    questions,answers,vocab=get_data()
```

In [0]:
```
#### get Trees  :

def to_nltk_tree(node):
    if node.n_lefts + node.n_rights > 0:
        return Tree(node.orth_, [to_nltk_tree(child) for child in node.children])
    else:
        return node.orth_

def get_tree(doc):
  l=[]
  for sent in doc.sents :
    l.append(to_nltk_tree(sent.root))
  return l

def nlp_sent(i,text):
  return nlp(text[i])
def extract_tree(l):
  L=[]
  for t in l :
    if isinstance(t, Tree):
      L.append(t)
  return L
def get_tree_i(i,text):
    return extract_tree(get_tree(nlp_sent(i,text)))
```

In [0]:
```
get_tree_i(len(questions)-1,questions)
```

Out[0]:
```
[Tree('one', ['the', Tree('left', ['first']), '?'])]
```

In [0]:
```
### Print tree
def print_tree_i(i,text):
    [to_nltk_tree(sent.root).pretty_print() for sent in  nlp_sent(i,text).sents]
    displacy.render(nlp_sent(i,text), style='dep', jupyter=True, options={'distance': 90})

print_tree_i(10,questions)
```

```
     is
  ___|________
 |   |        on
 |   |        |
 |   |      table
 |   |    ____|_____
 it  ?  the        close
```

```
<IPython.core.display.HTML object>
```

In [0]:
```
def get_label(i,answers):
  if answers[i]=='Yes':
    return 1
  else :
    return 0
```

```
In [0]: ## trees representation
        def tree_to_dict(tree,answers,vocab,emmb_dim):
            return {'Features': emmbeding(tree.label(),vocab,emmb_dim),'Labels':[get_label_vocab(tree.
        def tree_repr(i,questions,answers,vocab,emmb_dim):
          for tree in get_tree_i(i,questions) :
              d = tree_to_dict(tree,answers,vocab,emmb_dim)
          return d

In [0]: def get_label_vocab(word,vocab):
            i=0
            while vocab[i]!=word and i<len(vocab)-1:
              i+=1
            return i+1

In [0]: def emmbeding(word,vocab,emmb_dim):
          n=len(vocab)
          embeds = nn.Embedding(n,emmb_dim)
          lookup_tensor = torch.tensor(get_label_vocab('is',vocab), dtype=torch.long)
          return embeds(lookup_tensor).tolist()

In [0]: len(questions)

Out[0]: 579633

In [0]: tree_repr(len(questions)-1,questions,answers,vocab,1)

Out[0]: {'Features': [0.07187435775995255],
         'Labels': [45],
         'children': [{'Features': [-1.399527907371521],
           'Labels': [5],
           'children': []},
          {'Features': [0.20755603909492493],
           'Labels': [20],
           'children': [{'Features': [0.08566649258136749],
             'Labels': [523],
             'children': []}]},
          {'Features': [0.27789604663848877], 'Labels': [2768], 'children': []}]}

In [0]:

In [0]: def save_data_lstm_multi():
          with open('data_lstm_multi.txt', 'w') as f:
              for i in range(len(questions)-1):
                    f.write("%s\n" % tree_repr(i,questions,answers,vocab,1))
        save_data_lstm_multi()

In [0]: def string_to_dict(s):
            return ast.literal_eval(s)

In [0]: tree_repr(1,questions,answers,vocab,1)

Out[0]: {'Features': [-0.597744975566864],
         'Labels': [1],
```

```
 'children': [{'Features': [2.4134552478790283],
   'Labels': [6],
   'children': [{'Features': [-0.6968395113945007],
     'Labels': [5],
     'children': []}]},
  {'Features': [1.0749070644378662],
   'Labels': [1197],
   'children': [{'Features': [0.686913251876831],
     'Labels': [3],
     'children': []}]},
  {'Features': [0.4419419467449188], 'Labels': [2768], 'children': []}]}
```

```
In [0]: def data_loader():

            train = open("data_lstm_multi.txt",'r').readlines()
            train_data=[string_to_dict(l.strip('\n'+'\t')) for l in train]

            return train_data

        data=data_loader()
```

```
In [0]: data[0]
```

```
Out[0]: {'Features': [-2.0613930225372314,
          0.5069316625595093,
          -1.4566175937652588,
          0.26030299067497253,
          -0.0955534353852272,
          0.3825051486492157,
          1.4525792598724365],
         'Labels': [1],
         'children': [{'Features': [0.5152605772018433,
            0.17078812420368195,
            0.25252625346183777,
            -0.468440443277359,
            -0.6161168813705444,
            -0.9002719521522522,
            -1.1329265832901],
           'Labels': [2],
           'children': []},
          {'Features': [0.5762326121330261,
            -1.288496971130371,
            1.1080185174942017,
            2.346923589706421,
            0.9453363418579102,
            0.48000234365463257,
            -1.8401901721954346],
           'Labels': [6],
           'children': [{'Features': [1.573652744293213,
              -0.6953001618385315,
              0.31231674551963806,
              0.37766388058662415,
              2.299513816833496,
              -2.253248691558838,
              -0.4915894865989685],
```

```
     'Labels': [3],
     'children': []}]},
   {'Features': [0.3830993175506592,
     -0.833378255367279,
     0.19006630778312683,
     1.355594515800476,
     -0.6927298307418823,
     1.9384177923202515,
     -1.1607344150543213],
    'Labels': [2768],
    'children': []}]}
```

In [0]: