Université de Montréal

Méta-enseignement : Génération active d'exemples par apprentissage par renforcement

par

Stéphanie Larocque

Département d'informatique et recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Intelligence artificielle

mai 2020

Université de Montréal

Faculté des études supérieures et postdoctorales

Ce mémoire intitulé

Méta-enseignement : Génération active d'exemples par apprentissage par renforcement

présenté par

Stéphanie Larocque

a été évalué par un jury composé des personnes suivantes :

Pierre-Luc Bacon
(président-rapporteur)

Yoshua Bengio
(directeur de recherche)

Emma Frejinger
(co-directrice)

Christopher Pal
(membre du jury)

Mémoire accepté le :
29 juin 2020

Sommaire

Le problème d'intérêt est un problème d'optimisation discrète dont on tente d'approximer les solutions des instances particulières à l'aide de réseaux de neurones. Un obstacle à résoudre ce problème par apprentissage automatique réside dans le coût d'étiquettage élevé (et variable) des différentes instances, rendant coûteuse et difficile la génération d'un ensemble de données étiquettées. On propose une architecture d'apprentissage actif, qu'on nomme architecture de méta-enseignement, dans le but de pallier à ce problème. On montre comment on combine plusieurs modèles afin de résoudre ce problème d'apprentissage actif, formulé comme un problème de méta-apprentissage, en utilisant un agent d'apprentissage par renforcement pour la génération active d'exemples. Ainsi, on utilise des concepts de plusieurs domaines de l'apprentissage automatique dont des notions d'apprentissage supervisé, d'apprentissage actif, d'apprentissage par renforcement, ainsi que des réseaux récurrents. Dans ce travail exploratoire, on évalue notre méthodologie sur un problème simple, soit celui de classifier des mains de poker en 10 classes pré-établies. On teste notre architecture sur ce problème jouet dans le but de simplifier l'analyse. Malheureusement, l'avantage d'utiliser l'architecture de génération active n'est pas significatif. On expose ensuite plusieurs pistes de réflexion sur certaines observations à approfondir dans de futurs travaux, comme la définition de la fonction de récompense. Dans de futurs projets, il serait également intéressant d'utiliser un problème plus similaire au problème d'optimisation initial qui comporterait, entre autres, des coûts d'étiquettage variables.

Mots-clés : Intelligence artificielle, apprentissage automatique, apprentissage actif, apprentissage par renforcement, méta-apprentissage, recherche opérationnelle

Summary

The motivating application behind this architecture is a discrete optimisation problem whose solution we aim to predict using neural networks. A main challenge of solving this problem by machine learning lies in the high (and variable) labelling cost associated to the various instances, which leads to an expensive and difficult dataset generation. We propose an active learning architecture, called meta-teaching, to address this problem. We show how we combine several models to solve the active learning problem, formulated as a metalearning problem, by using a reinforcement learning agent to actively generate new instances. Therefore, we use concepts from various areas of machine learning, including supervised learning, active learning, reinforcement learning and recurrent networks. In this exploratory work, we evaluate our method on a simpler problem, which is to classify poker hands in 10 predefined classes. We test our architecture on this toy dataset in order to simplify the analysis. Unfortunately, we do not achieve a significant advantage using our active generation architecture on this dataset. We outline avenues for further reflections, including the definition of the reward function. In future projects, using a more similar problem to our problem of interest having, among others, a variable labelling cost, would be interesting.

Keywords : Artificial intelligence, machine learning, active learning, reinforcement learning, metalearning, operational research

Table des matières

Liste des tableaux

Liste des figures

Chapitre 1

Introduction

L'apprentissage automatique fournit des modèles et des algorithmes d'apprentissage pouvant résoudre des problèmes très variés, tant au niveau du language naturel que de la reconnaissance d'images, ou même pour la prévision de la demande. Ce mémoire se positionne surtout à l'intersection de l'apprentissage automatique et de la recherche opérationnelle. Plus précisément, plusieurs techniques d'apprentissage automatique sont combinées pour les appliquer à un problème d'optimisation discrète ou, du moins, à tout problème pouvant générer directement des instances et ayant accès à un oracle pour leurs étiquettes respectives. L'application réelle qui a motivé ce projet est un problème d'optimisation dans le domaine ferroviaire où on a accès à un oracle mais où l'utilisation de cet oracle est impossible au moment de prédiction compte tenu, entre autres, du temps de calcul trop lourd pour obtenir certaines étiquettes. On essaie donc d'approximer cet oracle à l'aide de réseaux de neurones, où un exemple x est une instance particulière du problème d'optimisation, et où la vraie solution y peut être obtenue à l'aide de l'oracle. Cependant, un des problèmes majeurs à utiliser les algorithmes d'apprentissage automatique pour résoudre des problèmes d'optimisation est la difficulté d'obtenir des paires d'instances et solutions. Comme obtenir les étiquettes peut être coûteux, on tente donc d'utiliser le budget d'étiquettage plus judicieusement en générant des instances pertinentes tout au long de l'apprentissage, plutôt que de générer, initialement, un ensemble de données aléatoires. Une instance doit être utile au processus d'apprentissage du prédicteur pour être considérée pertinente et cette notion de pertinence doit inclure le coût d'étiquettage si le coût d'étiquettage est variable entre les instances. On se retrouve donc dans un paradigme d'apprentissage actif, qui réunit les méthodes sélectionnent séquentiellement, durant l'entrainement, certaines nouvelles instances à étiquetter, dans le but d'aider à

l'apprentissage du prédicteur. Ainsi, notre objectif est non seulement d'approximer l'oracle auquel on a accès pendant l'entrainement, mais surtout de tirer profit de la qualité des données qu'on peut choisir de générer (et étiquetter) pour aider l'apprentissage supervisé. Une des idées maitresses de ce projet est de formuler ce problème d'apprentissage actif comme un problème de méta-apprentissage, et de le résoudre à l'aide d'un agent d'apprentissage par renforcement. Cet agent choisira séquentiellement quelles instances générer et étiquetter, toujours dans le but d'aider l'apprentissage du prédicteur. On se retrouve dans un contexte particulier de résolution d'un problème de recherche opérationnelle (section 1.1) où on utilise des notions de base d'apprentissage automatique comme l'apprentissage supervisé (section 1.2), mais également de l'apprentissage actif (section 1.3), de l'apprentissage par renforcement (section 1.4) et des réseaux récurrents (section 1.5). Une vue d'ensemble est présentée à la section 1.6, alors que tous les détails concernant l'architecture active sont présentés dans l'article.

1.1. Recherche opérationnelle

La motivation de ce projet provient d'un problème en recherche opérationnelle (RO) où on a accès à un oracle, mais où certaines restrictions, comme le temps, nous empêchent de pouvoir l'utiliser au moment de prédiction. On veut donc utiliser des réseaux de neurones pour approximer cet oracle. Cependant, comme obtenir les solutions de certaines instances est très coûteux, le budget d'étiquettage des données, qu'il soit exprimé en terme du nombre d'instances ou du temps alloué, est au coeur de ce projet. Notre objectif est donc d'approximer l'oracle tout en limitant le coût d'étiquettage nécessaire à la création de l'ensemble d'entrainement, comme en apprentissage actif. Ainsi, on note par x une instance particulière de ce problème d'optimisation et par y la solution fournie par l'oracle. Il faudra donc combiner efficacement plusieurs modèles afin de réussir à approximer la vraie solution, dans le cas où on n'aurait justement pas accès au solveur mathématique, tout en limitant le budget total d'étiquettage. L'approximation des outils d'optimisation se fait à l'aide de réseaux de neurones, alors que la génération de l'ensemble de données passe par un agent de RL (dans un contexte d'apprentissage actif formulé comme du méta-apprentissage) qui génère intelligemment les prochaines instances à ajouter à l'ensemble d'entrainement, plutôt qu'aléatoirement.

On donne les concepts de base de recherche opérationnelle dans la section 1.1.1. Le problème d'intérêt, soit le problème de chargement planifié, est expliqué à la section 1.1.2, alors que les résultats antérieurs d'application d'apprentissage automatique à ce problème sont donnés à la section 1.1.3.

1.1.1. Définition

Contrairement aux problèmes résolus par apprentissage automatique, qui sont souvent faciles à comprendre mais difficiles à modéliser mathématiquement, les problèmes résolus par recherche opérationnelle (RO) peuvent être modélisés mathématiquement, mais requièrent souvent trop de temps ou d'énergie pour être résolus par des humains [Larsen et al., 2018]. Les problèmes d'optimisation de transport et les problèmes d'horaire font partie des problèmes classiques de recherche opérationnelle. Il s'agit de problèmes de prises de décision souvent résolus par des méthodes analytiques et mathématiques. La modélisation des problèmes de RO passe par trois composantes-clés [Bastin, 2010]:

(1) Les *variables*, dont la valeur doit être déterminée au cours du processus d'optimisation;
(2) La *fonction objectif*, qui correspond à une expression mathématique devant être maximisée (ou minimisée), selon la valeur des variables impliquées;
(3) Les *contraintes*, qui limitent les solutions faisables en imposant des restrictions sur la valeur des variables.

Selon la forme que prend la fonction objectif et les contraintes sur le type des variables, on obtient différentes classes de problèmes de RO. Lorsque la fonction objectif et les contraintes sont toutes des fonctions linéaires, on dit qu'il s'agit d'un problème *linéaire*. Lorsqu'une contrainte d'intégralité est mise sur toutes les variables, on nomme ceci de la *programmation en nombre entiers*, alors qu'une contrainte d'intégralité sur seulement une partie des variables rend le problème un problème de *programmation mixte (entière)*. Le problème d'intérêt initial, qui est expliqué plus loin, est un problème d'optimisation en nombres entiers, où les variables ne peuvent prendre que des valeurs entières. Selon le type de problème, différentes techniques peuvent être utilisées. La méthode du simplexe est largement utilisée dans le cas des problèmes linéaires [Hillier and Lieberman, 2015]. Pour la programmation mixte

entière, d'autres techniques sont utilisées, comme la relaxation des contraintes, le *Branch-and-Bound* ou, dans certains cas, la programmation dynamique [Bastin, 2010]. Les solveurs mathématiques (comme CPLEX) sont également très utilisés et prennent en compte les variables, les contraintes et le type de problème afin de fournir une solution optimale. Celle-ci pouvant être difficile ou coûteuse à calculer, l'utilisation d'heuristiques (algorithmes donnant une solution pouvant être sous-optimale mais généralement peu coûteuse à obtenir) peut être pertinente pour donner une solution de base ou bien une idée de la valeur de la fonction objectif.

Comme le présent mémoire ne se penche pas sur les techniques de recherche opérationnelle, mais plutôt sur des techniques d'apprentissage automatique appliquées à la RO, on n'élabore pas plus sur les techniques courantes de RO. On utilise d'ailleurs le solveur mathématique, CPLEX dans notre cas, comme un oracle donnant la solution demandée, au besoin, sans souci de comment il y arrive.

1.1.2. Problème de chargement planifié (*Load Planning Problem*)

L'application réelle qui a motivé ce projet est un problème dans le domaine ferroviaire nommé problème de chargement planifié pour les plateformes à double niveau, ou *LPP* (*load planning problem for double-stack intermodal trains* [Mantovani et al., 2018]). Il s'agit d'un problème combinatoire visant à assigner des conteneurs sur des plateformes en minimisant la quantité de conteneurs laissés en gare, en optimisant la disposition et l'utilisation des plateformes tout en respectant les contraintes sur les modes de chargements (*loading patterns*) concernant les poids maximaux et les types de marchandises et conteneurs permis sur les plateformes. Les caractéristiques des conteneurs incluent le poids, la longueur, la hauteur, le type de marchandise, alors que les plateformes sont décrites avec leur capacité maximale de poids, leur nombre de plateformes et leurs modes de chargement disponibles. On veut donc déplacer les conteneurs sur des plateformes d'une gare à l'autre de façon optimale.

Ce problème a été décrit comme un problème de programmation linéaire en nombres entiers (ILP) par Mantovani et al. [2018] et peut donc être résolu à l'aide d'un solveur ILP. Étant donné un ensemble de conteneurs et de plateformes (avec leurs propriétés respectives), un chargement optimal fourni par le solveur contient l'affectation de chacun des conteneurs choisis à un espace particulier sur une plateforme particulière. Cette solution détaillée peut

être tronquée pour ne conserver que le nombre de chaque type de conteneurs et de plateformes utilisés dans le chargement optimal. Ces versions réduites, également appelées solutions tactiques, fournissent des informations utiles dans différents problèmes de décisions en amont du problème LPP. Prédire ces solutions réduites est considéré nettement plus facile que prédire les solutions globales, car une grande partie de l'aspect combinatoire est retiré. Les solveurs pouvant résoudre ce type de problème exigent des valeurs d'entrée complètes (avec toutes les caractéristiques de chaque objet). Toutefois, on veut résoudre ce problème dans le contexte de prise de décision pour les réservations d'espace de conteneur sur les wagons lorsque (1) certaines propriétés (comme le poids des conteneurs) ne sont pas disponibles, car la réservation est faite longtemps avant le chargement du conteneur, et lorsque (2) seul un temps de calcul limité est accordé (seulement quelques fractions de secondes). Dans ce contexte, les solveurs ne peuvent pas être utilisés. Ces restrictions surviennent lors de l'optimisation du réseau ferroviaire en entier (plutôt qu'une seule gare à la fois), où seul importe le nombre de conteneurs et de plateformes qui transigent entre les différentes gares, et pas leur placement sur les plateformes. C'est pourquoi on ne s'intéresse qu'à prédire les solutions réduites. Le temps de calcul devient une contrainte, car on doit trouver la solution pour plusieurs gares en même temps lorsqu'on optimise le flot des plateformes sur le réseau entier.

Plus précisément, soit T_C, le nombre de différents types de conteneurs et soit T_P, le nombre de différents types de plateformes. Une solution y donnée par le solveur est une solution combinatoire où chaque conteneur déplacé est assigné à un endroit particulier d'une plateforme particulière. Considérant qu'on a limité le nombre de conteneurs différents à $T_C = 2$, et le nombre de plateformes différentes à T_P=10, la solution réduite y' est un vecteur de nombres entiers positifs de taille $T = 12$ indiquant le nombre de conteneurs ou plateformes de chaque type utilisés dans la solution combinatoire optimale. Puisque les poids des conteneurs (entre autres) ne sont pas disponibles au moment de la réservation, il y a deux types d'entrée différents. Lorsque les poids sont inconnus, puisque la flotte est restreinte à 12 types distincts d'objets, l'entrée est également de taille 12 (taille fixe), avec comme valeur le nombre d'objets de chaque type. D'un autre côté, lorsque les poids des conteneurs sont disponibles et connus, l'entrée complète ne peut pas être un vecteur de taille 12, car chacun des conteneurs a maintenant son propre ensemble de propriétés distinctes. On utilise alors

une liste contenant tous les objets utilisés, chacun étant unique et contenant ses propres caractéristiques (taille, poids, etc.). Cette entrée n'est pas de taille fixe.

Le but de ce projet est de fournir un algorithme d'apprentissage automatique visant à approximer les outils d'optimisation (solveurs mathématiques) dans ce contexte particulier. Pour créer l'ensemble de données d'entrainement, on génère les vecteurs de taille fixe (12), auxquels on assigne aléatoirement les caractéristiques manquantes. Comme le solveur mathématique peut résoudre le problème lorsque toutes les informations nécessaires sont fournies, les entrées complètes lui seront fournies afin d'obtenir la solution combinatoire détaillée. Dans l'ensemble de données supervisées, on utilise la version réduite de cette solution détaillée comme étiquette à notre entrée. Cependant, étiquetter ainsi un grand nombre de données peut devenir trop coûteux, surtout dans l'application qui nous intéresse, où le coût d'étiquettage varie beaucoup entre les instances. Ce coût varie selon la difficulté des instances (i.e., leur taille), mais peut également varier pour une même taille donnée. Ainsi, on aimerait pouvoir mieux utiliser le budget d'étiquettage en prenant en compte la pertinence des instances par rapport à leur coût d'étiquettage. En plus de s'intéresser à la tâche à résoudre, on s'intéresse donc principalement à réduire le coût d'étiquettage de l'ensemble de données. C'est cette idée qui guide le reste du mémoire. La méthode utilisée par Larsen et al. [2018] pour effectuer la tâche de prédiction par apprentissage automatique (sans toutefois tenir compte des coûts d'étiquettage) est présentée brièvement dans la prochaine section.

1.1.3. Résultats antérieurs

La première étape est d'avoir un prédicteur pouvant donner la solution réduite du problème d'optimisation discrète. Larsen et al. [2018] utilisent un MLP (voir section 1.2) afin de prédire les solutions réduites optimales. La sortie souhaitée est un vecteur de taille fixe, chaque composante correspondant au nombre de conteneurs ou plateformes de chaque type utilisés dans la solution optimale. Deux formulations différentes sont étudiées : l'une exprimant ce problème comme une tâche de classification et l'autre en tâche de régression. Dans les deux cas, des réseaux de neurones sont utilisés pour prédire les solutions tactiques fournies par le solveur (traitées comme les étiquettes). Larsen et al. [2018] comparent ces réseaux de neurones avec des heuristiques et avec une borne inférieure obtenue en résolvant un problème de programmation stochastique, et les résultats avec ces réseaux sont largement

meilleurs, en plus d'être près de la borne. Le réseau de régression a une performance légèrement supérieure à celui pour la classification. Ces réseaux comportent environ 9 couches cachées de 600 neurones. Les prédictions données par ces deux réseaux sont adaptées afin de satisfaire aux contraintes d'éligibilité (par exemple, on ne peut avoir qu'un nombre entier de conteneurs, et pas supérieur au nombre de conteneurs initial). De notre côté, on met l'accent sur la génération active d'exemples, en plus de la qualité de prédiction des réseaux de neurones. On définit dans les prochaines sections les notions nécessaires pour l'architecture développée (types de réseaux de neurones, types d'apprentissage, etc.).

1.2. Notions de base sur l'apprentissage automatique

On définit dans cette section les différents types d'apprentissage, les réseaux de neurones, les fonctions de coût ainsi que la descente de gradient et diverses techniques de régularisation.

1.2.1. Types d'apprentissages

On sépare habituellement les différentes classes d'apprentissage en catégories distinctes qui diffèrent tant au niveau des modèles et des algorithmes utilisés qu'au niveau des types de données et de l'objectif final. Ces catégories incluent l'apprentissage supervisé, l'apprentissage non-supervisé, l'apprentissage semi-supervisé et l'apprentissage par renforcement. Dans ce mémoire, on utilise principalement l'apprentissage supervisé et l'apprentissage par renforcement.

L'*apprentissage supervisé* utilise un ensemble de données étiquetté, défini comme

$$\mathcal{D} = \left\{(x^{(1)}, y^{(1)}),\, (x^{(2)}, y^{(2)}), \ldots,\, (x^{(n)}, y^{(n)})\right\},$$

où chaque exemple $x^{(i)}$ est associé à une étiquette $y^{(i)}$. Le but de l'apprentissage supervisé est non seulement d'apprendre à correctement prédire l'étiquette associée à chaque exemple fourni mais, surtout, de pouvoir bien généraliser à d'autres exemples inconnus, mais provenant de la même distribution. Il y a plusieurs tâches possibles. En *classification*, chaque instance $x^{(i)}$ est associée à une étiquette $y^{(i)} \in \{1, 2, \ldots, m\}$ correspondant à l'une des m classes possibles. L'algorithme n'a droit qu'aux $x^{(i)}$ pour prédire la classe correspondante; ainsi, toute l'information disponible et nécessaire doit s'y retrouver. Alors que les instances $x^{(i)}$ peuvent représenter des images, des mots ou des propriétés quelconques, les étiquettes, quant à elles, sont encodées dans un vecteur *one-hot*: si $x^{(i)}$ appartient à la k-ème classe,

alors $y^{(i)}$ est un vecteur de zéros de taille m, sauf à la k-ème position, qui accueille un 1. En *régression*, chaque instance $x^{(i)}$ est associée à une valeur $y^{(i)} \in \mathcal{R}^m$. De la même façon qu'en classification, toutes les propriétés disponibles pour l'apprentissage se retrouvent dans les $x^{(i)}$. Les étiquettes $y^{(i)}$ sont maintenant des vecteurs réels de taille m. Notons que le problème de classification peut également être vu comme un cas particulier de la régression, comme en régression logistique.

En *apprentissage non-supervisé*, l'ensemble de données est uniquement constitué de données non étiquettées

$$D = \left\{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\right\}.$$

Le but n'est plus de prédire l'étiquette associée à chaque exemple, mais plutôt de comprendre l'ensemble de données. Par exemple, en *clustering*, on cherche à séparer les données en un nombre prédéfini de classes, ou *clusters*. La différence avec la classification réside dans le fait que seul le nombre de classes est donné, sans toutefois étiquetter aucun exemple. Un autre exemple de tâche non-supervisée consiste à apprendre la distribution des données ou même à générer de nouvelles données (comme dans le cas des modèles génératifs). Comme l'apprentissage non-supervisé n'est pas utilisé dans ce mémoire, on ne donne pas davantage de détails.

L'*apprentissage semi-supervisé* utilise autant des données étiquettées que des données non-étiquettées. Comme l'étiquettage des données peut être coûteux, l'apprentissage semi-supervisé tente d'utiliser toute l'information disponible, autant dans les données étiquettées que dans celles non étiquettées. On peut considérer l'apprentissage actif comme un cas particulier de l'apprentissage semi-supervisé. L'apprentissage actif permet d'augmenter la taille de l'ensemble d'entrainement en sélectionnant séquentiellement des instances à étiquetter. Ces instances sont souvent choisies à l'aide de critères comme la pertinence de l'ajout et la quantité d'information qu'elles pourraient apporter. Comme une grande partie de ce mémoire repose sur l'apprentissage actif, on présente ces techniques plus en détails à la section 1.3.

L'*apprentissage par renforcement* est un peu différent des précédents paradigmes d'apprentissages, car il s'agit d'un agent interagissant avec un environnement dans le but de trouver une stratégie pouvant récolter le plus de récompenses possibles. Ainsi, au lieu de prendre les décisions selon un ensemble de données fourni au départ, l'agent a accès et peut interagir avec un environnement afin de déterminer sa stratégie optimale. Il peut interagir

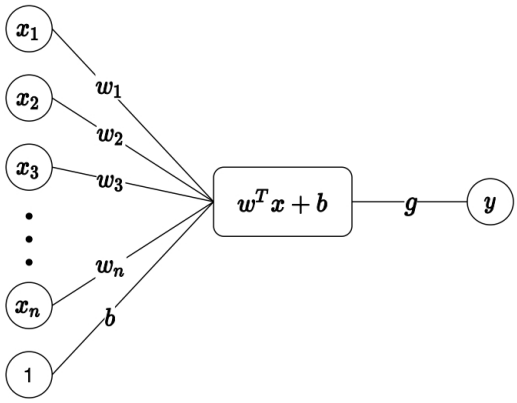


Fig. 1.1. Représentation visuelle d'un neurone calculant $y = g(w^T x + b)$

selon une politique différente de la politique à améliorer (off-policy) ou bien directement améliorer la politique utilisée (on-policy). La description de cet environnement inclut la description de l'état dans lequel l'agent se trouve, la description des actions possibles ainsi que le calcul des récompenses. Comme une grande partie du projet repose également sur l'apprentissage par renforcement, on en discute plus en détails dans la section 1.4.

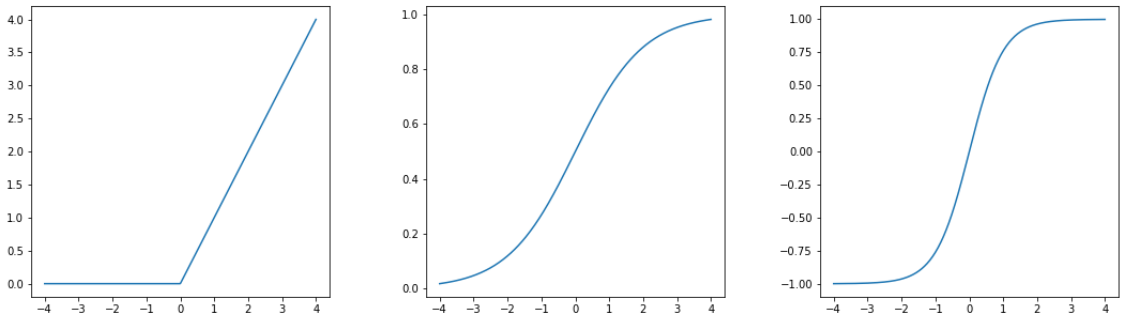
1.2.2. Réseaux de neurones

Une des façons de résoudre ces différentes tâches est d'utiliser des réseaux de neurones. On nomme l'unité de calcul de base un neurone. Comme représenté à la figure 1.1, il s'agit d'une unité de calcul ayant comme vecteur d'entrée $x \in R^n$ et calculant

$$y = g(w^T x + b), \tag{1.2.1}$$

où $w \in R^n$ et où b est un scalaire. Les poids w et b sont appris lors de l'entrainement, par exemple à l'aide des techniques de descente de gradient (section 1.2.4), afin de satisfaire le mieux possible l'objectif. On appelle la fonction g une fonction d'activation, et celle-ci permet d'éviter de se retrouver sous la contrainte de ne pouvoir calculer qu'une fonction linéaire. On présente certaines des fonctions d'activation les plus connues comme le *rectified linear unit* (ReLU), la sigmoid et la tangente hyperbolique (tanh) à la figure 1.2.

Un perceptron est constitué d'une couche de m neurones ayant chacun leur ensemble de paramètres. Un réseau de neurones est constitué d'une séquence de perceptrons dont l'entrée du suivant est la sortie du précédent. On appelle les réseaux de neurones profonds des MLP (pour *multilayer percreptron*) car on empile les couches de perceptron une par dessus



(a) $\text{ReLU}(x) = \max\{0,x\}$ (b) $\text{Sigmoid}(x) = \frac{1}{1+e^{-x}}$ (c) $\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Fig. 1.2. Fonctions d'activation courantes

l'autre, avec leurs fonctions d'activation. On les note par f_θ, où θ représente l'ensemble des paramètres à apprendre. Une représentation visuelle, inspirée de Dertat [2017], est présentée dans la figure 1.3. La première couche est appelée la couche d'entrée et la dernière couche, la couche de sortie. Toutes les couches intermédiaires sont nommées couches cachées. Les réseaux de neurones profond (*deep neural networks*) sont constitués de plusieurs couches cachées. À part la dernière couche (qui ne comporte parfois pas de fonction d'activation, ou une fonction d'activation particulière), toutes les autres couches se terminent par une fonction d'activation. L'utilisation de telles fonctions d'activation est cruciale, car cela permet d'éviter de ne calculer qu'une longue multiplication matricielle. L'architecture des réseaux de neurones est définie par plusieurs hyperparamètres comme le nombre de couches et la taille des couches.

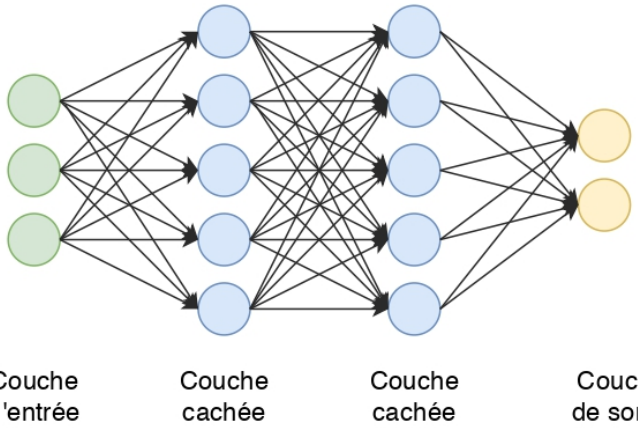


Fig. 1.3. Réseau de neurones profond, où chaque bulle représente un neurone.

1.2.3. Fonction de coût

Dans cette présente section, on se concentre sur les fonctions de coût pour l'apprentissage supervisé. Dans ce cas, la fonction de coût est une mesure de performance sur l'ensemble de données étiquettées $\mathcal{D}$. Elle prend habituellement la forme d'une moyenne de l'erreur sur chacun des exemples d'entrainement

$$J_{\mathcal{D}}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\big(y^{(i)}, \hat{y}^{(i)}\big), \tag{1.2.2}$$

où f_θ représente un réseau de neurones ayant comme paramètres θ et où $\mathcal{L}$ est une mesure de distance entre l'étiquette $y^{(i)}$ et l'étiquette prédite $\hat{y}^{(i)} = f_\theta(x^{(i)})$. Le but de l'apprentissage est d'obtenir la valeur des paramètres θ^* minimisant cette fonction afin d'obtenir la meilleure prédiction:

$$\theta^* = \arg\min_{\theta} J_{\mathcal{D}}(\theta). \tag{1.2.3}$$

Différentes fonctions de coût sont utilisées en apprentissage supervisé, selon la tâche concernée. En *classification*, puisque le but est de correctement prédire la classe à laquelle appartient chaque exemple, on souhaite avoir une fonction de coût permettant d'augmenter la probabilité d'appartenir à la bonne classe et ce, pour chacun des exemples fournis. Par exemple, en classification binaire où la sortie du modèle, notée $\hat{y}^{(i)} = f_\theta\left(x^{(i)}\right)$, représente en fait la probabilité d'appartenir à la classe 0, soit $\hat{y}^{(i)} = P\big(y^{(i)} = 0\big)$. On utilise l'entropie croisée afin de maximiser la probabilité d'appartenir à la bonne classe. Pour chaque couple $\big(x^{(i)}, y^{(i)}\big)$ de l'ensemble des données, on veut donc minimiser

$$\mathcal{L}\left(y^{(i)}, \hat{y}^{(i)}\right) = -\Big(y^{(i)} \log\left(\hat{y}^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - \hat{y}^{(i)}\right)\Big). \tag{1.2.4}$$

Cette fonction de coût se généralise également au cas de classification multiclasse ($m > 2$). Dans ce cas, la fonction *softmax*, définie par

$$\sigma(z)_j = \frac{\exp(z_j)}{\sum_{\ell=1}^{m} \exp(z_\ell)}, \tag{1.2.5}$$

est utilisée à la sortie du réseau pour obtenir la probabilité d'appartenir à chacune des m classes respectives (où $z \in R^m$ est la sortie pré-softmax du réseau). La fonction d'entropie croisée pour la classification multiclasse se définit ainsi :

$$\mathcal{L}\left(y^{(i)}, \hat{y}^{(i)}\right) = -\sum_{j=1}^{m} \Big(y_j^{(i)} \ \log\Big(\hat{y}_j^{(i)}\Big)\Big), \tag{1.2.6}$$

où l'étiquette $y^{(i)}$ est encodée en vecteur *one-hot* alors que $\hat{y}_j^{(i)}$ représente la probabilité prédite d'appartenir à la classe j. Ainsi, en sommant cette fonction sur toutes les données d'entrainement disponibles, on maximise la probabilité de chaque exemple d'appartenir à la bonne classe. En *régression*, où $\hat{y}^{(i)} = f_\theta\left(x^{(i)}\right)$ correspond au vecteur de nombres réels prédit, la fonction d'erreur quadratique moyenne est habituellement employée. Pour chaque couple $(x^{(i)}, y^{(i)})$, on veut minimiser l'erreur de prédiction:

$$\mathcal{L}\big(y^{(i)}, \hat{y}^{(i)}\big) = \big|\big|y^{(i)} - \hat{y}^{(i)}\big|\big|_2^2. \tag{1.2.7}$$

Dans certains cas, l'utilisation de l'erreur absolue (ou norme $\mathcal{L}_1$) peut être pertinente:

$$\mathcal{L}\big(y^{(i)}, \hat{y}^{(i)}\big) = \big|y^{(i)} - \hat{y}^{(i)}\big|. \tag{1.2.8}$$

1.2.4. Descente de gradient (stochastique)

Dans le cas d'un entrainement supervisé, on veut donc minimiser une mesure de performance de la forme

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(y^{(i)}, f_\theta(x^{(i)})\right), \tag{1.2.9}$$

où f_θ est le réseau de neurones et où $\mathcal{L}$ est la fonction de coût appropriée de la tâche demandée. On peut remarquer que cet objectif représente une approximation de l'espérance des erreurs sur la distribution des instances. Pour minimiser une telle fonction, la technique la plus commune est d'utiliser la descente de gradient sur les paramètres θ du réseau. Cette technique utilise la rétropropagation du gradient pour propager le gradient de l'erreur à partir de la fonction de performance vers les paramètres (donc, de la dernière couche à la première). La rétropropagation du gradient utilise le principe de dérivation en chaîne, mais effectué de façon efficace. La règle de mise à jour des paramètres s'écrit sous sa forme générale ainsi:

$$\theta \longleftarrow \theta - \eta\nabla_\theta J(\theta), \tag{1.2.10}$$

avec

$$\nabla_\theta J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta\mathcal{L}\left(y^{(i)}, f_\theta(x^{(i)})\right), \tag{1.2.11}$$

où η correspond à la taille du pas de gradient. Pour pouvoir calculer le gradient et effectuer la descente de gradient, il faut s'assurer que toutes les fonctions utilisées (i.e., la fonction de coût et les fonctions d'activation) soient différentiables. C'est d'ailleurs pour cette raison

qu'on minimise la fonction d'entropie croisée pour la classification, plutôt que de directement minimiser l'erreur de prédiction, car cette dernière est une fonction 0-1 non différentiable.

On veut donc minimiser, pour l'ensemble de données d'entrainement $\mathcal{D}$, la valeur de la mesure de performance. Or, pour les ensembles de données de grande taille, calculer ce gradient sur toutes les données en même temps n'est pas réaliste. On sacrifie donc un peu de variance au profit de la vitesse de calcul en utilisant une approximation stochastique du gradient. La descente de gradient stochastique performe, à chaque pas de temps, la descente de gradient uniquement sur un sous-ensemble des données (*minibatch*) au lieu de l'ensemble d'entrainement complet, comme indiqué à l'algorithme 1. Cette approximation stochastique est de même espérance, donc n'est pas biaisée. Lorsqu'on a effectué le calcul de gardient pour chacune des données disponibles, on appelle ceci une époque. L'entrainement se poursuit ainsi sur un nombre d'époques maximum (ou selon un critère d'arrêt).

Algorithm 1 Descente de gradient stochastique pour une tâche de classification

Require: Ensemble d'entrainement $\mathcal{D}_{\text{train}} = \left\{(x^{(i)}, y^{(i)})\right\}_{i=1}^{n}$

Require: Réseau de neurones f_θ, avec paramètres θ

Require: Taille de la minibatch m

Require: Taille du pas de gradient η

for un certain nombre d'époques **do**

for $j = 1 \ldots$ nombre de minibatchs **do**

Sélectionner un sous-ensemble de taille m, soit une minibatch $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$

Pour chaque instance, calculer la prédiction $\hat{y}^{(i)} = f_\theta(x^{(i)})$

Calculer la valeur de la fonction de performance $J(\theta) \leftarrow \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(y^{(i)}, \hat{y}^{(i)}\right)$

$\nabla_\theta J(\theta) \leftarrow$ Calculer le gradient de la mesure de performance sur cette minibatch

Mettre à jour les paramètres $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$

end for

end for

1.2.5. Généralisation et régularisation

L'objectif final de l'apprentissage n'est pas uniquement d'obtenir la plus petite erreur sur l'ensemble d'entrainement, mais surtout de pouvoir bien généraliser à de nouvelles données

jamais vues, mais provenant de la même distribution. L'objectif final est donc d'obtenir la plus petite erreur de généralisation. À l'inverse, si le but final de l'algorithme était de ne faire aucune erreur sur l'ensemble d'entrainement, apprendre par coeur toutes les données serait suffisant. Pour éviter un tel surapprentissage (apprentissage par coeur), des techniques de régularisation existent et sont présentées dans cette section.

Premièrement, pour pouvoir approximer l'erreur de généralisation, on sépare l'ensemble de données en trois parties : l'ensemble d'entrainement $\mathcal{D}_{\text{train}}$, l'ensemble de validation $\mathcal{D}_{\text{valid}}$ et l'ensemble de test $\mathcal{D}_{\text{test}}$. L'ensemble d'entrainement est constitué des données sur lequel le modèle f_θ s'entraine. On trouve donc les paramètres θ en effectuant la descente de gradient sur $\mathcal{D}_{\text{train}}$. L'ensemble de validation permet de choisir les hyperparamètres. On appelle hyperparamètres les valeurs qui déterminent l'architecture du réseau et de l'entrainement, par exemple le nombre de couches, la taille des couches cachées, le nombre d'époques, la taille du pas d'apprentissage (*learning rate*), etc. La performance d'un réseau est directement reliée à la valeur de ces hyperparamètres, car ils définissent la capacité du modèle à apprendre. La capacité d'un modèle correspond, globalement, à la complexité des fonctions qu'il peut approximer. Les hyperparamètres sont sélectionnés en utilisant la performance sur $\mathcal{D}_{\text{valid}}$ comme critère. On utilise également l'erreur sur l'ensemble de validation comme estimé de l'erreur de généralisation. L'ensemble de test est uniquement utilisé à la fin de l'entrainement pour donner une estimation de la vraie erreur de généralisation sur des données inconnues. Cet ensemble de test ne doit ni servir à trouver les paramètres ni définir les hyperparamètres car, sinon, on biaise l'estimation de la vraie erreur de généralisation.

On appelle *régularisation* l'ensemble des techniques servant à réduire l'erreur de généralisation (qu'on estime par l'erreur sur l'ensemble de validation), même si accompli au détriment de l'erreur d'entrainement [Goodfellow et al., 2016]. Lorsque le réseau apprend les données d'entrainement par coeur, au lieu de bien généraliser, on appelle ceci du surapprentissage, et on peut le détecter en conservant les erreurs sur les ensembles d'entrainement et de validation. Lorsque l'erreur de validation ne diminue plus (ou même augmente) alors que l'erreur d'entrainement continue de diminuer, cela veut généralement dire que le surapprentissage a commencé. La première technique parant au surapprentissage se nomme l'arrêt anticipé (*early stopping*). Cette technique vise à arrêter l'entrainement lorsque l'erreur sur l'ensemble de validation ne diminue plus depuis un certain temps (lorsque la patience est

écoulée), même si le nombre total d'époques permis n'est pas atteint. D'autres techniques, comme l'ajout de termes de pénalisation $\mathcal{L}_1$ ou $\mathcal{L}_2$ ajoutent quant à eux une pénalisation sur la norme des paramètres appris dans la fonction de coût. Dans le cas de la norme $\mathcal{L}_2$, la fonction de performance devient donc

$$J'(\theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\big(x^{(i)},\, f_\theta(x^{(i)})\big) + ||\theta||_2^2. \tag{1.2.12}$$

Cela réduit la capacité du réseau de neurones et peut donc réduire le surapprentissage, car les paramètres sont encouragés à rester près de 0. Sinon, avoir des paramètres démesurés peut occasioner de grandes variations dans la prédiction même avec un petit changement d'entrée, ce qui n'est pas souhaité. Une autre technique de régularisation consiste à partager des paramètres entre différentes parties de certains modèles. Par exemple, si deux réseaux prennent les mêmes valeurs en entrée, mais ont des buts différents, on peut partager les premières couches des réseaux afin d'obtenir une représention globale des données d'entrée, avant de poursuivre sur des couches séparées (pour chacun de leurs objectifs).

Le *bagging*, initialement introduit par Breiman [1996], est une autre technique de régularisation. Il s'agit d'une méthode d'ensemble entrainant M modèles sur M ensembles d'entrainement différents, obtenus par bootstrap (c'est-à-dire, obtenus par échantillonnage avec remise de l'ensemble d'entrainement initial). Après l'entrainement, on obtient la prédiction finale en faisant voter ces M modèles indépendants. Cependant, entrainer plusieurs modèles ainsi est coûteux. Le *dropout* [Srivastava, 2013; Srivastava et al., 2014] a été développé comme une méthode de bagging non coûteuse permettant d'avoir un nombre exponentiel de modèles. Celle-ci consiste à supprimer temporairement, pendant l'entrainement, une certaine proportion des neurones du réseau, entre autres dans le but de réduire leur co-adaptation. À chaque minibatch, on applique donc un masque sur les paramètres et on entraine seulement une partie des neurones. On effectue donc un type de bagging, mais avec partage de paramètres entre les modèles. Comme on obtient des réseaux différents avec les masques aléatoires, on se retrouve donc avec un ensemble de modèles prédictifs. Au moment du test, tous les neurones sont présents et les poids sont remis à l'échelle, selon la proportion des neurones qui étaient supprimés.

Utiliser d'avantage de données est également une bonne façon d'obtenir une meilleure généralisation. Par contre, comme on ne peut généralement pas obtenir de nouvelles données

gratuitement, certaines techniques permettent d'augmenter la taille de l'ensemble d'entrainement en transformant légèrement les données déjà présentes dans l'ensemble, selon certains invariants. Par exemple, en classification d'images, effectuer une petite rotation ou un léger recadrement ne devrait pas changer la prédiction associée. On considère donc qu'augmenter la taille de l'ensemble d'entrainement de cette façon est une technique de régularisation, car cela permet au modèle de comprendre comment mieux généraliser. Il existe plusieurs autres techniques de régularisation, comme l'utilisation de données adversariales, l'ajout de bruit dans les données, etc.

1.3. Apprentissage actif et méta-apprentissage

Par opposition à l'apprentissage passif, où le modèle d'apprentissage f_θ ne peut pas influencer sur l'ensemble de données fourni, les algorithmes d'apprentissage actif influencent l'ensemble de données accessible en y rajoutant, selon certains critères, des *nouveaux* exemples. Tenter de mieux utiliser le budget d'instances est au coeur des méthodes d'apprentissage actif, tout comme l'accès à un oracle pouvant étiquetter ces nouvelles instances. Comme le processus d'étiquettage peut être coûteux (autant en temps qu'en difficulté), on peut définir le budget soit sur le coût d'étiquettage total ou bien sur le nombre total d'instances étiquettées. Inclure le coût d'étiquettage est important lorsque les coûts sont variables car, sinon, l'omettre peut empêcher l'algorithme d'apprentissage actif de surpasser l'apprentissage passif par échantillonnage aléatoire [Settles, Craven and Friedland, 2008]. En apprentissage actif, des nouvelles instances sont séquentiellement sélectionnées afin de les étiquetter et les rajouter à l'ensemble de données d'entrainement. La motivation derrière ces techniques se base sur le fait que, si on peut sélectionner les instances à ajouter, plutôt que de les échantillonner aléatoirement, alors le modèle d'apprentissage f_θ aura besoin de moins d'instances pour obtenir une même performance [Settles, 2009].

On présente l'apprentissage actif, car l'un des buts de notre projet sera justement de limiter le budget d'étiquettage total dans des problèmes où l'étiquettage est coûteux, sans toutefois réduire la performance sur la tâche. On présente dans la section 1.3.1 les différents types d'apprentissage actif et dans la section 1.3.2 les principaux critères utilisés. Ces critères estiment la pertinence d'étiquetter et d'ajouter les instances à l'ensemble de données. Le lien avec le méta-apprentissage est présenté dans la section 1.3.3.

Algorithm 2 Approche *pool-based*

for $t = 1 \ldots$ budget d'instances **do**
  for Chaque instance $x \in \mathcal{D}_U$ **do**
    Attribuer une valeur $C(x)$, selon le critère de sélection
  end for
  Sélectionner $x_t = \arg\max_{x \in \mathcal{D}_U} C(x)$
  Obtenir l'étiquette associée y_t avec l'oracle
  Ajouter la paire étiquettée (x_t, y_t) à $\mathcal{D}_L$
  Poursuivre l'entrainement de f_θ sur $\mathcal{D}_L$
end for

Algorithm 3 Approche *stream-based*

while Budget n'est pas atteint **do**
  Obtenir une instance non étiquettée x par le flux
  Attribuer une valeur $C(x)$, selon le critère de sélection
  if $C(x)$ dépasse un certain seuil **then**
    Demander l'étiquette y à l'oracle
    Ajouter la paire (x,y) à $\mathcal{D}_L$
    Poursuivre l'entrainement de f_θ sur $\mathcal{D}_L$
  end if
end while

Fig. 1.4. Approches sélectives de l'apprentissage actif. Les deux approches requièrent (1) un ensemble de données étiquettées $\mathcal{D}_L$, (2) un oracle, (3) un modèle d'apprentissage f_θ ainsi (4) qu'un critère de sélection C. L'approche *pool-based* requière également un ensemble de données non-étiquettées, qu'on note $\mathcal{D}_U$.

1.3.1. Types d'apprentissage actif

On peut séparer l'apprentissage actif en trois grandes catégories [Settles, 2009] : (1) l'approche *pool-based*, (2) l'approche *stream-based* (par flux) et (3) l'approche par synthèse de requêtes. Dans l'approche *pool-based*, il y a deux ensembles : l'ensemble de données étiquettées $\mathcal{D}_L$ et l'ensemble de données non-étiquettées $\mathcal{D}_U$. Après avoir noté l'ensemble des exemples candidats $\mathcal{D}_U$ selon un critère de pertinence choisi, on sélectionne une instance prometteuse. Cette instance est étiquettée puis ajoutée à l'ensemble d'entrainement, comme

indiqué[1] dans l'algorithme 2. L'approche *stream-based*, quant à elle, englobe les techniques où les instances sont itérativement fournies à l'aide d'un flux. L'algorithme d'apprentissage actif doit donc décider, à chaque pas de temps, s'il demande l'étiquette à l'oracle ou bien s'il écarte l'instance fournie. Il choisit l'action à prendre selon un critère de pertinence, comme indiqué dans l'algorithme 3.

Ces deux dernières approches peuvent être considérées comme des approches sélectives alors que l'approche par *synthèse de requêtes* est considérée comme une approche constructive, puisque les instances y sont directement générées au lieu d'être sélectionnées d'un ensemble ou fournies par un flux. En effet, comme indiqué dans l'algorithme 4, un générateur d'instances génère directement l'instance à étiquetter, par exemple en échantillonnant d'une distribution. Les requêtes peuvent également être d'autres formes, comme des requêtes de concept [Angluin, 1988]. Cependant, la plus grande limitation de la synthèse de requêtes est qu'on doit avoir accès à un générateur d'instances. Si les instances sont trop complexes pour être générées directement, alors cette technique peut difficilement être utilisée. Comme nous avons accès à un tel générateur, nous utilisons cette dernière technique dans notre projet.

Algorithm 4 Apprentissage actif par synthèse de requêtes (approche constructive)

Require: Ensemble de données étiquettées $\mathcal{D}_L$
Require: Générateur d'instances
Require: Oracle
Require: Modèle d'apprentissage f_θ
  for $t = 1, \ldots$ budget **do**
    Générer une nouvelle instance x_t
    Demander l'étiquette y_t à l'oracle
    Ajouter la paire (x_t,y_t) à $\mathcal{D}_L$
    Poursuivre l'entrainement de f_θ sur $\mathcal{D}_L$
    Poursuivre l'entrainement du générateur d'instances, si applicable
  end for

[1]On assume à des fins de simplification que le budget est exprimé en termes du nombre d'instances.

1.3.2. Critères de sélection

Dans le cas des approches sélectives, on utilise des critères de sélection afin de déterminer quelles instances étiquetter. On explique ces différents critères dans le cas *pool-based* pour une tâche de classification. Pour chaque $t = 1, \ldots, T$, où T est notre budget d'instances total, on attribue donc une valeur aux instances $x \in \mathcal{D}_U$ selon le critère choisi. On détermine ensuite quelle instance sera étiquettée, et on la note $x_t \in \mathcal{D}_U$. L'échantillonnage par incertitude (*uncertainty sampling* [Lewis and Gale, 1994]) se réfère aux méthodes où on sélectionne l'instance selon le degré d'incertitude dans la prédiction de son étiquette. On peut directement choisir l'instance dont la prédiction est la plus incertaine ainsi :

$$x_t = \arg\max_x \Big(1 - P_\theta(\hat{y}|x)\Big), \tag{1.3.1}$$

où $\hat{y} = \arg\max_y P_\theta(y|x)$, mais cela peut cependant prioriser le choix de données aberrantes [Roy and McCallum, 2001]. On peut également choisir l'instance où l'incertitude réside entre les deux classes plus problables ainsi

$$x_t = \arg\min_x \Big(P_\theta(\hat{y}_1|x) - P_\theta(\hat{y}_2|x)\Big), \tag{1.3.2}$$

où $\hat{y}_1$ et $\hat{y}_2$ sont, respectivement, les deux classes les plus probables. Cela permet de choisir selon l'incertitude entre deux classes, plutôt que de baser ce choix uniquement sur la classe la plus probable. Si on veut choisir selon le degré d'appartenance à toutes les classes, on peut également utiliser l'entropie [Shannon, 1948] pour déterminer quelle instance choisir, avec comme critère

$$x_t = \arg\max_x \Big(-\sum_i P_\theta(y_i|x) \log P_\theta(y_i|x)\Big), \tag{1.3.3}$$

mais ce calcul peut devenir coûteux. Une autre technique utilise une mesure de désaccord entre différentes prédictions. Nommée requête par vote (*query-by-committee* [Seung et al., 1992; Freund et al., 1992]), cette technique crée, entraine et maintient plusieurs modèles $f_{\theta_1}, f_{\theta_2}, \ldots f_{\theta_M}$ fournissant leurs prédictions respectives. Comme ces différents modèles représentent plusieurs hypothèses quant à la vraie étiquette, on sélectionne l'instance dont la prédiction occasionne le plus grand désaccord, afin de l'étiquetter. Pouvoir définir un tel ensemble de modèles (parfois par *bagging, boostrap*, réinitialisation des paramètres initiaux, partition de l'espace des données, etc.) ainsi qu'avoir une mesure de désaccord sont requis pour pouvoir utiliser cette technique.

Le changement de modèle prévu (*expected model change* [Settles, Craven and Ray, 2008; Cai et al., 2013]) consiste à étiquetter l'instance dont l'ajout occasionnerait le plus grand changement aux paramètres du modèle. Lorsque le modèle est entrainé par descente de gradient, sélectionner l'instance qui maximiserait la taille du gradient, si ajoutée au dataset, serait optimal. Or, comme on ne connait pas sa véritable étiquette, on calcule la taille espérée du gradient sur tous les choix d'étiquettes possibles. Cependant, cette technique peut devenir coûteuse. La réduction de l'erreur espérée (*expected error reduction*), introduite par Roy and McCallum [2001], consiste à sélectionner l'instance qui maximiserait la diminution de l'erreur, si ajoutée à l'ensemble de données. D'autres techniques existent également (réduction de la variance prévue, modèles de densités pondérées, etc.). Dans ce projet, comme on utilise l'approche constructive, on utilise plutôt un critère indirect pour entrainer le générateur d'instances à fournir des instances pertinentes. Plus de détails sont données dans l'article.

1.3.3. Apprentissage actif comme un problème de méta-apprentissage

En psychologie, le méta-apprentissage consiste à être conscient et contrôler son propre processus d'apprentissage [Biggs, 2011]. Pour être considéré comme du méta-apprentissage en apprentissage automatique, un processus d'apprentissage doit respecter deux principales exigences : (1) il doit y avoir un sous-système d'apprentissage et (2) l'algorithme de méta-apprentissage doit utiliser des méta-données sur ce sous-système pour accomplir sa tâche [Thrun and Pratt, 1998; Lemke et al., 2015; Caruana, 1995; Hochreiter, Younger and Conwell, 2001]. Un algorithme de méta-apprentissage se sépare donc en deux parties: un premier modèle f_θ tente d'accomplir une tâche spécifiée, alors qu'un second modèle tente de comprendre les méta-données reliées à l'apprentissage de ce dernier. L'algorithme de méta-apprentissage tente donc d'apprendre à un deuxième niveau, d'où le terme *méta* [Santoro et al., 2016a].

L'apprentissage actif peut être perçu comme du méta-apprentissage. En effet, l'algorithme de sélection de données tente d'aider le processus d'apprentissage d'un modèle de base f_θ à l'aide de nouvelles instances. On utilise effectivement des méta-données afin d'améliorer la performance de f_θ, car la performance du modèle est utilisée dans les critères de sélection. Dans ce projet, nous formulons le problème d'apprentissage actif sous un problème de méta-apprentissage. De plus, nous utilisons un agent d'apprentissage par renforcement (agent de RL, voir section 1.4) pour directement générer les nouvelles instances à étiquetter. Formuler

l'apprentissage actif sous un contexte de méta-apprentissage en utilisant un agent de RL a été étudié [Bachman et al., 2017], mais principalement dans le cas des méthodes sélectives *pool-based* [Baram et al., 2004; Hsu and Lin, 2015] et *stream-based* [Santoro et al., 2016b; Woodward and Finn, 2017]. Davantage de détails sont donnés dans la revue de littérature propre à l'article, mais la principale différence avec notre projet réside dans le fait que nous générons directement les instances (approche constructive plutôt qu'approche sélective).

1.4. Apprentissage par renforcement

On présente l'apprentissage par renforcement (RL), car on utilise un agent de RL comme générateur d'instances dans un contexte d'apprentissage actif. L'apprentissage par renforcement regroupe les méthodes permettant à un agent de percevoir l'environnement dans lequel il est plongé et de prendre des actions affectant l'état où il se trouve, dans le but de maximiser ses récompenses au long terme. Le compromis entre exploitation et exploration est l'un des grands défis reliés à l'apprentissage par renforcement. L'exploitation consiste à sélectionner les actions qui rapportent beaucoup de récompenses, alors que l'exploration consiste à choisir des actions possiblement sous-optimales afin de pouvoir trouver une meilleure façon d'agir au long terme. Comme on utilise un générateur d'instances modélisé par un agent de RL, on explique dans cette section les notions et algorithmes les plus importants. Toute la section 1.4 est grandement inspirée de l'excellent ouvrage de référence *Reinforcement Learning: An Introduction*, par Sutton and Barto [2018], autant pour la notation, les explications que l'organisation des idées.

1.4.1. Notions de base et définitions

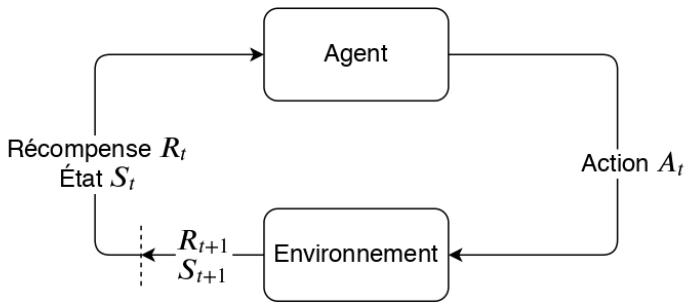


Fig. 1.5. Structure de l'apprentissage par renforcement

Les concepts liés à l'environnement, à l'agent, aux états, aux actions ainsi qu'aux récompenses sont définis dans la présente section, tout comme la définition de trajectoire, de politique et de fonction de valeur. L'*environnement* correspond à tout ce qui n'est pas contrôlé par l'agent et peut être déterministe ou non-déterministe. La figure 1.5, légèrement adaptée de Sutton and Barto [2018], démontre visuellement la différence entre environnement, action, récompense, agent et état. L'*agent* interagit avec l'environnement en posant des actions, ce qui influence l'état où il se trouve, dans le but de maximiser les récompenses obtenues au long terme. On note l'*état* où l'agent se trouve au temps t par S_t. Cet agent doit choisir une *action* $A_t \in \mathcal{A}(S_t)$ parmi les actions disponibles dans l'état S_t. L'espace des actions peut autant être discret que continu. Au prochain pas de temps, l'agent reçoit également une *récompense* numérale R_{t+1}, qui indique à quel point son choix d'action était bon pour l'objectif spécifié. Comme ceci est l'unique apport extérieur pour aider l'apprentissage de l'agent, il est crucial que la fonction de récompense soit bien définie. Une *trajectoire* est une suite de couples

$$(S_t, A_t, R_{t+1}, S_{t+1}), \;\; t = 1, 2, \ldots$$

représentant l'évolution de l'état et de l'agent au travers du temps. On distingue deux types d'environnements influençant sur le type de trajectoire: les environnements à horizon de temps fini et ceux à horizon de temps infini. Dans le cas fini (épisodique), l'agent commence dans un état initial et la trajectoire se termine après un horizon de temps fini T; c'est ce qu'on appelle un épisode. L'agent apprend au travers des multiples épisodes. Dans le cas infini, l'épisode ne termine jamais. Dans tous les cas, l'*objectif* de l'agent est de maximiser, à partir de l'état S_t d'une trajectoire, le cumul des récompenses obtenu, défini par

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t} R_T. \tag{1.4.1}$$

On note par γ le facteur d'escompte (*discount factor*), qui indique à quel point l'agent accorde de l'importance entre les récompenses présentes et futures. Pour unifier les deux cas, on permet l'abus de notation $T = \infty$ pour inclure le cas d'horizon de temps infini. La politique (*policy*) de l'agent, notée π, correspond à la stratégie employée pour obtenir le plus grand cumul de récompenses. Cette politique peut être déterministe, avec $A_t = \pi(S_t)$, ou stochastique, avec $A_t \sim \pi(S_t)$, selon l'environnement et la complexité du modèle. Alors que l'agent peut influencer la stratégie utilisée, la fonction de transition d'état $P(S_{t+1}|S_t,A_t)$ ainsi

que la fonction de récompense $r(S_t,A_t)$ sont propres à l'environnement. Dans ce qui suit, on considère que l'environnement peut être non-déterministe, tant au niveau de la transition vers l'état suivant que dans l'attribution de la récompense. Pour évaluer la pertinence de se retrouver dans un certain état, on définit la fonction de valeur, dans le cas d'une politique stochastique, par

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \sum_{a\in\mathcal{A}(s)} \pi(a|s) \sum_{s'} P(s'|s,a)\big[r(s,a) + \gamma v_\pi(s')\big], \tag{1.4.2}$$

ce qui correspond à la valeur espérée du cumul des récompenses que l'agent pourrait obtenir à partir de l'état s, s'il suit la politique π. L'agent tente donc de trouver une politique π qui l'amène dans les états prometteurs. On peut également utiliser la fonction de $q-$valeur, qui donne la valeur pour chaque couple état-action (s,a). Celle-ci est définie ainsi:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \sum_{s'} P(s'|s,a)\big[r(s,a) + \gamma v_\pi(s')\big] \tag{1.4.3}$$

et on remarque que $v_\pi(s) = \sum_{a\in A} \pi(a|s) q_\pi(s,a)$. De plus, la fonction d'avantage est définie ainsi

$$A_\pi(s,a) = q_\pi(s,a) - v_\pi(s), \tag{1.4.4}$$

et correspond à *l'avantage* de sélectionner l'action a lorsqu'on est dans l'état s par rapport aux autres actions possibles (qui sont prises en compte dans $v_\pi(s)$). On dit qu'une politique π est meilleure que la politique π' (noté $\pi \geq \pi'$) si, pour chaque état s, on a $v_\pi(s) \geq v_{\pi'}(s)$. On définit la politique optimale comme la politique π^* ayant comme fonction de valeur $v^*(s)$, soit la fonction de valeur optimale, définie par

$$v^*(s) = \max_\pi v_\pi(s). \tag{1.4.5}$$

Ainsi, l'équation d'optimalité de Bellman pour la fonction de valeur s'écrit comme

$$v^*(s) = \max_a \sum_{s'} P(s'|s,a)\big[r(s,a) + \gamma v^*(s')\big], \tag{1.4.6}$$

d'où découlent les algorithmes suivants. Pour la fonction de q-valeur, l'équation d'optimalité s'énonce ainsi

$$q^*(s,a) = \sum_{s'} P(s'|s,a)\big[r(s,a) + \gamma \max_{a'} q^*(s',a')\big]. \tag{1.4.7}$$

1.4.2. Apprentissage par différence temporelle (TD Learning)

On sépare souvent les politiques apprises en deux catégories : il y a (1) les politiques se basant directement sur la fonction de valeur (*value-based methodes*) et (2) les politiques apprises directement (*policy-gradient methods*). Dans le premier cas, la politique choisie est une politique $\epsilon-$gloutonne définie directement à partir de la fonction de valeur. Le but de l'apprentissage est donc d'obtenir la fonction de valeur la plus fiable possible pour qu'ensuite l'agent puisse agir de façon optimale. Par exemple, si l'agent a accès à la vraie fonction de $q-$valeur, il peut choisir son action par :

$$\pi(s) = \begin{cases} \arg\max_a q_\pi(s,a) & \text{avec probabilité } 1-\epsilon \\ \text{action aléatoire} & \text{avec probabilité } \epsilon \end{cases}$$

Cette politique est clairement séparée entre exploitation, où l'agent effectue l'action la plus prometteuse, et exploration, où l'agent effectue une action aléatoire. On peut trouver cette fonction de valeur en utilisant, par exemple, des méthodes de programmation dynamique ou de Monte Carlo. Les méthodes par programmation dynamique, comme *Policy Iteration, Value Iteration* ou encore *Policy Improvement*, utilisent du *bootstrapping* et présument une connaissance parfaite de l'environnement. En particulier, ces algorithmes utilisent la fonction de transition $P(s'|s,a)$ et la fonction de récompense $r(s,a)$ pour définir itérativement la valeur de chaque état, jusqu'à convergence. La fonction de transition donne la probabilité de se retrouver dans l'état s' lorsque l'agent effectue l'action a dans l'état s. De plus, on dit que les méthodes de programmation dynamique font du *bootstrapping*, car ils utilisent leur propre approximation de la fonction de valeur pour estimer cette même fonction. Plus précisément, si on note le cumul des récompenses (dans le cas épisodique) par $\sum_{k=0}^{T-t} \gamma^k R_{t+k+1}$, on appelle approximation par bootstrap sur $n = 1$ pas de temps l'approximation suivante:

$$G_{t:t+1} = R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}). \tag{1.4.8}$$

Notons qu'il y a deux raisons pour lesquelles il s'agit d'une approximation du vrai cumul de récompenses: (1) on regarde uniquement un certain nombre fixe de pas de temps dans le futur, plutôt que jusqu'à la fin de l'épisode et (2) on utilise l'approximation courante de la fonction de valeur pour l'état suivant S_{t+1}, bien qu'au moment où on sera dans cet état, la fonction de valeur et/ou la politique auront été mises à jour. On définit donc le *bootstrapping*

comme la méthode qui utilise la valeur (approximée) des états futurs pour le calcul de l'état présent. Ainsi, dans les méthodes par programmation dynamique, la règle de mise à jour de la fonction de valeur

$$\hat{v}_\pi(s) \longleftarrow \sum_{s'} P(s'|s,\pi(s))\big[r(s,a) + \gamma \hat{v}_\pi(s')\big] \tag{1.4.9}$$

utilise l'approximation *bootstrap* de la valeur des états futurs $\hat{v}_\pi(s')$ ainsi qu'une connaissance parfaite de l'envionnement à travers la fonction de transition. Cependant, il n'est pas toujours possible d'avoir une connaissance parfaite de l'environnement dans lequel l'agent évolue.

Les méthodes par Monte Carlo (MC), au contraire, ne présument pas avoir une telle connaissance de l'environnement. En fait, ces méthodes apprennent les fonctions de valeurs uniquement avec l'interaction avec l'environnement (que ce soit simulé ou pas). Elles effectuent des épisodes complets et, à la fin de chaque épisode, propagent les récompenses obtenues aux valeurs des états visités durant l'épisode. Pour un certain état S_t, sa valeur est mise à jour en faisant une moyenne sur toutes les expressions de la forme $R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t} R_T$, qui correspondent au total des récompenses obtenues à partir de cet état, pour chaque épisode où S_t a été visité.

L'apprentissage par différence temporelle (*TD-learning*, algorithme 5) est un hybride entre les algorithmes Monte Carlo et la programmation dynamique. Tout comme les algorithmes MC, la fonction de valeur est apprise directement avec l'interaction avec l'environnement, sans présumer une connaissance parfaite de la fonction de transition et, tout comme en programmation dynamique, ils utilisent l'approximation bootstrap de la fonction de valeur pour mettre à jour cette même fonction de valeur. Un des grands avantages de l'algorithme TD est qu'il est en-ligne et incrémental, on ne doit donc pas nécessairement attendre à la fin de l'épisode avant de mettre à jour la fonction de valeur. À chaque état S_t visité, on y assigne la valeur

$$\begin{aligned} \hat{v}_\pi(S_t) &\longleftarrow (1-\eta)\hat{v}_\pi(S_t) + \eta\big(R_{t+1} + \gamma \hat{v}_\pi(S_{t+1})\big) \\ &\longleftarrow \hat{v}_\pi(S_t) + \eta \underbrace{\big[R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}) - \hat{v}_\pi(S_t)\big]}_{\delta_t}. \end{aligned} \tag{1.4.10}$$

On nomme $\delta_t = \big[R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}) - \hat{v}_\pi(S_t)\big]$ l'erreur de différence temporelle (*TD-error*). Cette erreur de différence temporelle est au coeur dans plusieurs algorithmes et correspond en fait à la fonction d'avantage présentée plus haut.

Algorithm 5 Algorithme TD

Require: Environnement et politique initiale $\pi(s)$

Require: Longueur de l'épisode T et taille du pas d'apprentissage η

Initialiser $\hat{v}_\pi(s)$ pour chaque état s

for $i = 1 \dots$ nb épisodes **do**

$S_0 \leftarrow$ État initial

for $t = 0 \dots T$ **do**

$A_t \leftarrow \pi(S_t)$

$R_{t+1}, S_{t+1} \leftarrow$ obtenus de l'environnement selon S_t, A_t

$\hat{v}_\pi(S_t) \leftarrow \hat{v}_\pi(S_t) + \eta\Big[R_{t+1} + \gamma\hat{v}_\pi(S_{t+1}) - \hat{v}_\pi(S_t)\Big]$

$S_t \leftarrow S_{t+1}$

end for

end for

1.4.3. Méthode du semi-gradient

Dans les méthodes précédentes, on se trouve dans le cas tabulaire où on garde une liste contenant la valeur de chaque état. Cependant, ces méthodes ne peuvent s'appliquer que pour de petits problèmes avec un nombre limité d'états. Dans les autres cas, on utilise plutôt une fonction de valeur paramétrisée, par exemple à l'aide d'un réseau de neurones, pour obtenir la valeur de chaque état. On met à jour les paramètres servant à approximer cette fonction de valeur plutôt que de mettre à jour les valeurs directement. Comme on veut pouvoir utiliser la descente de gradient, la paramétrisation de la fonction de valeur doit être différentiable (comme dans un réseau de neurones classique). Nous voulons apprendre une fonction de valeur $\hat{v}_\pi(s, \omega)$, paramétrisée par ω, pour ensuite avoir une politique basée sur cette fonction de valeur. Notre objectif final est de nous rapprocher de la fonction de valeur cible $v^*_\pi(s)$, qui existe mais est inconnue. Supposons pour un instant que nous ayons accès à cette fonction de valeur cible $v^*_\pi(s)$. Dans ce cas, on pourrait utiliser le paradigme d'apprentissage supervisé et tenter d'en apprendre la meilleure approximation $\hat{v}_\pi(s, \omega)$ en minimisant une mesure de distance, pondérée par la distribution des états, ainsi:

$$\sum_{s \in S} \mu_\pi(s)\big(v^*_\pi(s) - \hat{v}_\pi(s, \omega)\big)^2, \tag{1.4.11}$$

où ω sont les paramètres de la fonction de valeur approximée $\hat{v}_\pi$ et où $\mu_\pi(s)$ est la distribution des états visités lorsque la politique π est utilisée. Notons ici que les deux fonctions de valeur ($v^*_\pi(s)$ et $\hat{v}_\pi(s,\omega)$) sont reliées à la stratégie utilisée, car la distribution des récompenses (et donc le cumul des récompenses, qu'on tente de maximiser) est directement reliée à la politique π. Par contre, seule la fonction de valeur approximée $\hat{v}_\pi(s,\omega)$ dépend des paramètres ω. En utilisant la descente de gradient stochastique, qui ne considère que l'erreur de l'approximation sur l'état courant S_t plutôt que sur toute la distribution des états (approximation non biaisée), la règle de mise à jour des paramètres au temps t deviendrait, à une constante près,

$$\begin{aligned}\omega &\longleftarrow \omega - \eta\nabla_\omega\big(v^*_\pi(S_t) - \hat{v}_\pi(S_t,\omega)\big)^2 \\ &\longleftarrow \omega + \eta\big[v^*_\pi(S_t) - \hat{v}_\pi(S_t,\omega)\big]\nabla_\omega\hat{v}_\pi(S_t,\omega).\end{aligned} \tag{1.4.12}$$

Seul problème avec cette méthode: la fonction de valeur cible v^*_π n'est en fait pas disponible. Au temps t, l'algorithme n'a seulement accès qu'à la récompense immédiate R_{t+1} et à la fonction de valeur approximée $\hat{v}_\pi$. On a donc besoin de remplacer la valeur cible $v^*_\pi(S_t)$ par une valeur approximée. Sachant que, pour une stratégie particulière, la valeur d'un état représente la somme espérée des récompenses, on pourrait remplacer la valeur cible par la somme des récompenses à partir de cet état jusqu'à la fin l'épisode en cours, soit par $G_t = \sum_{k=0}^{T-t-1}\gamma^k R_{t+k+1}$, ce qui est une réalisation du cumul de récompenses espéré, pour une trajectoire. Par contre, utiliser cette somme n'est pas très pratique dans le cas des problèmes à horizon temporel infini ou en-ligne. C'est pourquoi on peut utiliser $G_{t:t+1}$ comme approximation bootstrap de G_t, où

$$G_{t:t+1} = R_{t+1} + \gamma\hat{v}_\pi(S_{t+1},\omega). \tag{1.4.13}$$

Notons qu'il s'agit ici d'une cible approximée par bootstrap sur un pas de temps, mais on peut généraliser cette approximation sur n pas de temps. Avec cette approximation, la règle de mise à jour des paramètres deviendrait donc

$$\omega \longleftarrow \omega + \eta\Bigg(\underbrace{R_{t+1} + \gamma\hat{v}_\pi(S_{t+1},\omega) - \hat{v}_\pi(S_t,\omega)}_{\delta_t}\Bigg)\nabla_\omega\hat{v}_\pi(S_t,\omega). \tag{1.4.14}$$

On retrouve encore une fois l'erreur de différence temporelle δ_t. Cette méthode est appelée la méthode du semi-gradient car l'effet de ω sur la cible approximée $G_{t:t+1}$ est totalement ignoré : le gradient $\nabla_\omega\hat{v}_\pi(S_{t+1},\omega)$ est écarté bien qu'il puisse ne pas être nul. On pourrait

utiliser la méthode du semi-gradient pour obtenir une fonction de valeur, que ce soit dans une optique de méthode basée sur la fonction de valeur, ou bien dans une méthode hybrique entre les techniques *value-based* et *policy-based*, comme présenté dans les prochaines sections.

1.4.4. Gradient de la politique (*Policy Gradient*)

On présente dans les prochaines sections une méthode basée directement sur la politique, plutôt que sur la fonction de valeur, comme c'est cette méthode que nous utilisons dans notre travail. Les méthodes *policy-based* offrent certains avantages par rapport aux méthodes basées sur la fonction de valeur [Sutton et al., 2000]. Premièrement, alors que les méthodes *value-based* se concentrent à avoir la meilleure politique déterministe (ou $\epsilon-$gloutonne), les méthodes basées directement sur la politique tentent de trouver la politique stochastique la plus optimale. Comme, dans certaines situations, la politique optimale peut nécessiter d'être stochastique, utiliser les méthodes *policy-based*, qui ont une façon naturelle de trouver de telles politiques, est plus adapté. Deuxièmement, comme la probabilité de sélectionner chacune des actions va varier doucement en fonction des paramètres de la politique, cela permettra un apprentissage plus facile. Au contraire, les méthodes basées sur les fonctions de valeur peuvent souffrir de changements drastiques (causés par le argmax) dans leur politique, même si la fonction de valeur ne varie pas beaucoup.

On introduit ici la stratégie par gradient de la politique (*Policy gradient*). Contrairement à la méthode des différences temporelles (TD), où la politique dépend directement de la fonction de valeur, la stratégie par gradient de la politique est calculée directement à partir de l'état S_t, sans nécessairement passer par le calcul de la fonction de valeur. On paramétrise la politique à l'aide de paramètres θ, sous la contrainte que la fonction doive être différentiable. L'utilisation de réseaux de neurones, qui sont différentiables, est donc indiqué. Ces paramètres sont mis à jour en fonction du gradient d'une certaine mesure de performance $J(\theta)$ comme suit :

$$\theta \longleftarrow \theta + \eta\widehat{\nabla J(\theta)} \tag{1.4.15}$$

Dans le cas épisodique, la mesure de performance est définie comme $J(\theta) \doteq v_\pi(S_0)$, où S_0 est l'état initial. Or, cette valeur est inconnue, puisque v_π (la vraie fonction de valeur) est inconnue. Heureusement, le Théorème du gradient de la politique (*Policy Gradient Theorem*

[Sutton et al., 2000]) indique que

$$\nabla J(\theta) \propto \sum_s \mu_\pi(s) \sum_a q_\pi(s,a) \nabla_\theta \pi(a|s,\theta), \tag{1.4.16}$$

où $\mu_\pi(s)$ est défini comme la distribution sur les états lorsqu'on suit la politique active π. Ainsi, au lieu d'utiliser directement la mesure de performance $J(\theta)$ (ce qui est impossible, car si on avait la vraie fonction de valeur, on n'aurait pas besoin de faire ces algorithmes d'apprentissage), on utilise la double somme (1.4.16), qui introduit une façon d'obtenir un estimé non-biaisé du gradient. Ce théorème est crucial pour la prochaine méthode expliquée.

1.4.5. Algorithme REINFORCE

L'algorithme REINFORCE est basé sur le théorème du gradient de la politique et sur la règle de mise à jour des paramètres (1.4.15). Le théorème du gradient de la politique ne fournit qu'une expression (souvent intraitable à calculer) pour remplacer le gradient de la performance. Tout comme avec la descente de gradient stochastique (où on utilise une approximation non biaisée du vrai gradient en le calculant uniquement sur un sous-ensemble des données), on cherche en fait à calculer un estimé stochastique non biaisé de $\nabla J(\theta)$. Heureusement, on peut obtenir un estimé ayant la même espérance ainsi:

$$\begin{aligned}
\nabla J(\theta) &\propto \sum_s \mu_\pi(s) \sum_a q_\pi(s,a) \nabla_\theta \pi(a|s,\theta) && \text{théorème du gradient de la politique} \\
&= \mathbb{E}_{S_t \sim \pi} \left[\sum_a q_\pi(S_t,a) \nabla_\theta \pi(a|S_t,\theta) \right] && \text{par définition de l'espérance} \\
&= \mathbb{E}_{S_t \sim \pi} \left[\sum_a \pi(a|S_t,\theta) q_\pi(S_t,a) \frac{\nabla_\theta \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)} \right] && \text{multiplication par 1} \\
&= \mathbb{E}_{A_t,S_t \sim \pi} \left[q_\pi(S_t,A_t) \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \right] && \text{par définition de l'espérance} \\
&= \mathbb{E}_{A_t,S_t \sim \pi} \Big[G_t \nabla_\theta \ln \pi(A_t|S_t,\theta) \Big]. && \text{car } \mathbb{E}[G_t|A_t,S_t] \doteq q_\pi(S_t,A_t)
\end{aligned}$$

Ainsi, la double somme a été remplacée par une valeur espérée sur des paires d'états-actions, celles-ci pouvant être échantillonnées. En utilisant ces paires d'états-actions, on peut retrouver la règle de mise à jour non biaisée de l'algorithme REINFORCE [Williams, 1992]:

$$\theta \longleftarrow \theta + \eta \; G_t \; \nabla_\theta \ln \pi(A_t|S_t,\theta), \tag{1.4.17}$$

où G_t correspond au cumul des récompenses. Lorsqu'on termine l'épisode, on calcule le cumul des récompenses G_t pour chaque pas de temps t et on met à jour les paramètres. Cependant, utiliser directement cette règle de mise à jour occassionne une grande variance dans la mise à jour des paramètres résultant de l'échantillonnage des paires d'états-actions et du signal de récompenses. Une des méthodes pour diminuer l'effet de la variance consiste à utiliser une fonction de référence (*baseline*). Cette technique se base sur le fait qu'ajouter une valeur indépendante de l'espace des actions dans la deuxième somme du théorème du gradient de la politique ne change pas l'espérance, et donc que l'approximation stochastique reste valide (non biaisée). En effet, pour toute fonction $b(s)$ indépendante de l'espace des actions, on a que :

$$\sum_a b(s)\nabla_\theta \pi(a|s,\theta) = b(s)\sum_a \nabla_\theta \pi(a|s,\theta) = b(s)\nabla_\theta \sum_a \pi(a|s,\theta) = b(s)\nabla_\theta 1 = 0. \quad (1.4.18)$$

On peut donc modifier la règle de mise à jour sans affecter la valeur espérée. Notons que la fonction de référence peut être n'importe quelle fonction (même une fonction aléatoire!), tant qu'elle est indépendante de l'espace des actions. Combinée avec cette égalité (1.4.18), le théorème de la politique du gradient peut être modifié ainsi :

$$\begin{aligned}\nabla J(\theta) &\propto \sum_s \mu_\pi(s) \sum_a q_\pi(s,a)\nabla_\theta \pi(a|s,\theta) \\ &= \sum_s \mu_\pi(s) \sum_a \big(q_\pi(s,a) - b(s)\big)\nabla_\theta \pi(a|s,\theta), \end{aligned} \quad (1.4.19)$$

car la valeur espérée reste la même. De son côté, la mise à jour des paramètres pour l'algorithme REINFORCE avec fonction de référence [Williams, 1992], devient

$$\theta \longleftarrow \theta + \eta\Big(G_t - b(S_t)\Big)\nabla_\theta \ln \pi(A_t|S_t,\theta). \quad (1.4.20)$$

Un choix naturel pour la fonction de référence $b(s)$ est la fonction de valeur approximée $\hat{v}_\pi(S_t,\omega)$. En soustrayant la valeur de la fonction de référence à G_t, on peut espérer en réduire la variance, car on utilise la différence entre la valeur moyenne estimée de cet état et le cumul des récompenses obtenu, et pas uniquement le cumul des récompenses. Il y aura donc deux fonctions à apprendre: $b(s)$ et $\pi(a|s,\theta)$. On note que $\pi(a|s,\theta)$ utilise la fonction de valeur comme fonction de référence, mais pas pour directement choisir l'action à prendre.

1.4.6. Agent-critique (*Actor-Critic*)

L'objectif principal de l'algorithme d'acteur-critique est de modifier l'algorithme REINFORCE pour le rendre en-ligne et incrémental. Dans l'algorithme REINFORCE, il faut attendre la fin de l'épisode avant de pouvoir mettre à jour les paramètres. Rappelons que, dans l'algorithme REINFORCE, la fonction de référence $b(s)$ (qui est souvent choisie comme étant la fonction de valeur approximée) n'est utilisée que comme référence, et pas dans le calcul du cumul des récompenses G_t. Par contre, dans l'algorithme d'acteur-critique, la fonction de valeur est utilisée autant comme fonction de référence que pour l'approximation du cumul des récompenses sur un pas de temps, par bootstrap. Sachant que l'approximation du cumul des récompenses sur un pas de temps est défini par $G_{t:t+1} \doteq R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}, \omega)$, on peut remplacer le vrai cumul des récompenses G_t dans la règle de mise à jour de l'algorithme REINFORCE avec baseline par son approximation sur un pas de temps (le *1-step return*):

$$\begin{aligned}\theta &\longleftarrow \theta + \eta\Bigg(\underbrace{G_{t:t+1}}_{\text{cumul sur 1 pas de temps}} - \underbrace{\hat{v}_\pi(S_t, \omega)}_{\text{référence}}\Bigg)\nabla_\theta \ln \pi(A_t|S_t, \theta) \\ &\longleftarrow \theta + \eta\Bigg(\underbrace{R_{t+1} + \gamma\hat{v}_\pi(S_{t+1}, \omega)}_{\text{cumul sur 1 pas de temps}} - \underbrace{\hat{v}_\pi(S_t, \omega)}_{\text{référence}}\Bigg)\nabla_\theta \ln \pi(A_t|S_t, \theta). \qquad (1.4.21)\end{aligned}$$

On remarque qu'on retrouve encore, à travers cette règle de mise à jour, l'erreur de différences temporelles $\delta_t = R_{t+1} + \gamma\hat{v}_\pi(S_{t+1}, \omega) - \hat{v}_\pi(S_t, \omega)$. On pourrait également prendre d'autres approximations du cumul des récompenses (par exemple, sur n pas de temps, tout en utilisant la fonction de valeur par bootstrap). On appelle "critique" le réseau apprenant la fonction $\hat{v}_\pi$, alors que l'acteur correspond plutôt à l'agent et à sa politique π. Il est important de noter que la valeur $\hat{v}_\pi(S_{t+1}, \omega)$ est reliée au critique, tandis que $\hat{v}_\pi(S_t, \omega)$ est uniquement la valeur de la fonction de référence. Il s'agit d'un *hasard* judicieux que ces deux valeurs puissent être calculées à partir de la même fonction. On peut également réaliser que $R_{t+1} + \gamma\hat{v}_\pi(S_{t+1}, \omega)$ devient un estimateur de q_π, et donc que l'erreur de différences temporelles est en fait la fonction d'avantage (équation 1.4.4). Comme indiqué plus haut, on a donc deux fonctions à apprendre : la politique et la fonction de valeur, chacune ayant son propre ensemble de paramètres. La règle de mise à jour des paramètres θ de la politique π, obtenue avec l'algorithme REINFORCE avec baseline et bootstrap, est

$$\theta \longleftarrow \theta + \eta_\theta\ \delta_t \nabla_\theta \ln \pi(A_t|S_t, \theta), \qquad (1.4.22)$$

alors que la mise à jour des paramètres ω de la fonction de valeur $\hat{v}_\pi$, obtenue avec la méthode du semi-gradient (section 1.4.3), est

$$\omega \longleftarrow \omega + \eta_\omega \ \delta_t \nabla_\omega \hat{v}_\pi(s, \omega) \tag{1.4.23}$$

Une subtilité importante à noter ici est que bien que l'erreur-TD δ_t est la même dans chacune des règles de mise à jour, elle ne provient pas du tout des mêmes calculs et dérivations. Le δ_t de la mise à jour des paramètres de la politique provient de l'algorithme REINFORCE, avec référence puis approximation bootstrap, alors que le δ_t de la fonction de valeur provient de la méthode du semi-gradient. On peut donc réaliser que les méthodes d'acteur-critique sont en fait des méthodes hybrides entre les méthodes basées sur les fonctions de valeurs et celles basées sur le gradient de la politique. L'algorithme entier d'acteur-critique est présenté dans l'algorithme 6.

Algorithm 6 Algorithme Acteur-Critique AC

Require: Environnement, Learning rate η_θ et η_ω

Require: Fonctions de valeur $\hat{v}_\pi(s;\omega)$ et politique $\pi(a|s,\theta)$

Initialiser les paramètres θ et ω

for $i = 1 \ldots$ nb épisodes **do**

$S_0 \leftarrow$ État initial

for $t = 0 \ldots T$ **do**

$A_t \leftarrow$ échantilloné à partir de $\pi(a|S_t, \theta)$

$R_{t+1}, S_{t+1} \leftarrow$ obtenus de l'environnement selon S_t, A_t

$\delta_t \leftarrow R_{t+1} + \gamma \hat{v}_\pi(S_{t+1};\omega) - \hat{v}_\pi(S_t;\omega)$

$\theta \leftarrow \theta + \eta_\theta \ \delta_t \nabla_\theta \ln \pi(A_t|S_t, \theta)$

$\omega \leftarrow \omega + \eta_\omega \ \delta_t \nabla_\omega \hat{v}_\pi(S_t, \omega)$

$S_t \leftarrow S_{t+1}$

end for

end for

1.5. Réseaux de neurones récurrents

Comme on utilise, dans notre projet, un agent de RL agissant comme générateur d'instances dans un contexte d'apprentissage actif, il faut définir l'état perçu par l'agent de RL.

Cependant, comme il n'y a pas de définition d'état S_t triviale à utiliser pour notre application, nous avons utilisé un réseau récurrent (RNN) pour le définir. Notre choix s'est arrêté sur un réseau récurrent, initialement introduit par Rumelhart et al. [1986], puisque l'agent de RL apprend au travers du temps, et les réseaux récurrents ont été spécialement pensés pour des données séquentielles. De cette façon, des entrées reliées à la qualité de l'apprentissage du prédicteur sont successivement fournies au RNN, qui s'en servira pour faire un résumé de l'évolution de l'apprentissage et de l'effet des instances ajoutées. On aura donc une définition d'état S_t pertinente pour l'agent de RL. Mais avant, définissons mieux les réseaux récurrents.

1.5.1. Description

Un des buts principaux des réseaux récurrents est de pouvoir traiter des données séquentielles, parfois de tailles variables [Goodfellow et al., 2016]. Ils sont entre autres utilisés pour traiter le langage naturel dans le but de traduire, de comprendre, de résumer, de faire une analyse ou même de pouvoir discuter avec un humain [Young et al., 2017]. Dans un réseau de neurones classique (à propagation avant), on passe toujours l'information vers les prochaines couches du réseau. Au contraire, dans les réseaux récurrents, il y a du transfert d'information à travers les différents *copies* du RNN, ces copies étant obtenues en déroulant le réseau récurrent à travers le temps, comme illustré à la Figure 1.6. Les entrées x_{t-1}, x_t, x_{t+1} représentent les éléments séquentiels de l'entrée. Le réseau récurrent accumule une mémoire de ce qu'il a déjà vu et compris à l'aide de son état h_t. Idéalement, cette mémoire interne h_t a assez d'information pour permettre au RNN de voir l'évolution des données séquentielles et de pouvoir ainsi effectuer sa tâche au temps t. Le réseau combine ensuite l'état précédent de la mémoire (h_{t-1}) avec l'entrée courante (x_t) pour obtenir l'état courant h_t. Si le réseau récurrent retourne une valeur, il utilise justement cette mémoire interne h_t pour calculer la sortie y_t. Ensuite, la valeur de la mémoire h_t est réutilisée pour calculer la mémoire suivante, et ainsi de suite. Le réseau conserve donc l'information des précédentes données dans cet état interne, qu'on appelle également cellule.

On utilise le partage des paramètres entre les différentes copies du réseau de neurones. À chaque flèche de la représentation compacte de la figure 1.6 étant associée des paramètres, on utilise ces mêmes paramètres pour chaque *copie* du réseau déroulé. Cela permet entre autres aux séquences $x_1, \ldots x_T$ d'être de tailles variables puisque, peu importe la taille de l'entrée,

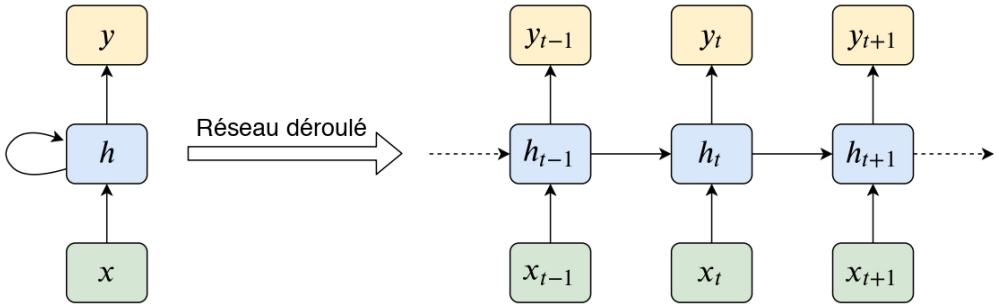


Fig. 1.6. Réseau récurrent. À gauche : représentation compacte, à droite : représentation déroulée. L'entrée du réseau est représentée par x, sa sortie par y, et son état caché (où la mémoire est conservée), par h.

les mêmes paramètres sont utilisés tout au long pour calculer la sortie (indépendamment de la taille de la séquence). Cela permet en même temps une forme de régularisation.

1.5.2. Rétropropagation du gradient

La rétropropagation est effectuée comme dans les réseaux de neurones conventionnels, sauf qu'il faut maintenant considérer en plus la rétropropagation à travers le temps (BBTT, pour *backpropagation through time* [Werbos, 1990]). Cela implique qu'il faut faire propager le gradient à travers tous les autres pas de temps précédents, donc au travers des différentes copies du réseau déroulé. Cela amène certains problèmes par rapport au gradient, soit la disparition ou l'explosion des gradients (*vanishing or exploding gradients*), et rend l'apprentissage ardu [Hochreiter, 1991; Hochreiter, Bengio, Frasconi and Schmidhuber, 2001; Bengio et al., 1994]. Dans le cas des réseaux récurrents, la dérivation en chaîne implique de multiplier les mêmes matrices plusieurs fois (t fois pour t pas de temps). Or, un nombre inférieur à 1 ira assez rapidement vers 0, alors qu'un nombre supérieur à 1 grandira très rapidement, lorsque multiplié avec lui-même plusieurs fois. De ce fait, l'apprentissage devient difficile à gérer. Dans le cas de la disparition des gradients, cela implique que les mises à jour des paramètres sont très petites, et donc que l'apprentissage est pratiquement absent. Dans le cas des gradients explosifs, les paramètres du modèle deviendront arbitrairement grands. Les interactions à long terme sont rapidement écartées au profit des interactions à court terme.

1.5.3. Réseaux récurrents à portes

Certaines architectures classiques, comme les LSTM (*Long Short Term Memory* [Hochreiter and Schmidhuber, 1997]) et les GRU (*Gated Recurrent Unit* [Cho et al., 2014]), ont été développées afin de parer au problème des gradients microscopiques ou explosifs. Ces architectures se basent sur un système de portes pouvant contrôler le flux d'information à travers les différentes parties du modèle. Celles-ci déterminent quelle partie de l'information est assez pertinente pour être conservée, et quelle partie n'est pas assez importante et doit être oubliée. Le but de ces architectures est de garder des tailles de gradients raisonnables (sans disparition ni explosion) en créant des chemins où le flux du gradient n'est pas coupé. On utilise uniquement le modèle LSTM, expliqué que très brièvement à l'aide d'un schéma, puisqu'il est utilisé tel quel.

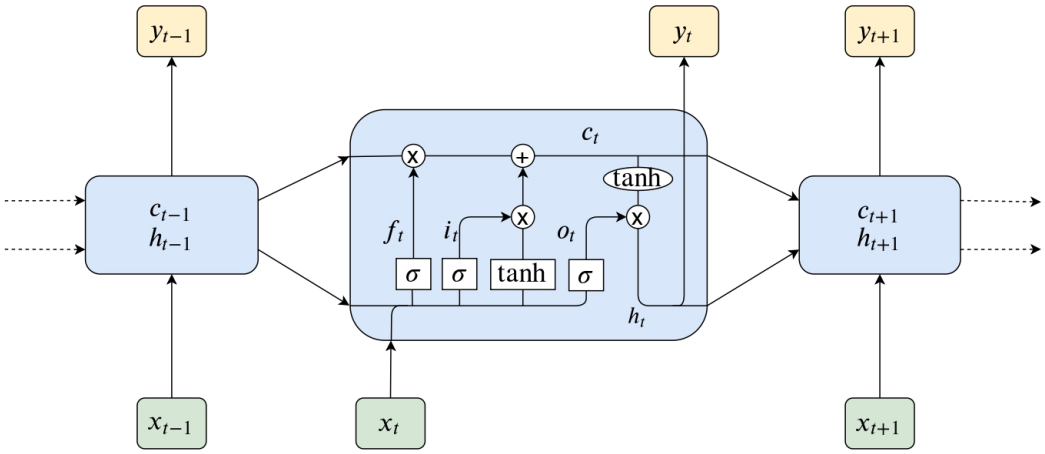


Fig. 1.7. Cellule de mémoire d'un réseau récurrent LSTM. À chaque pas de temps t, la cellule de mémoire prend en entrée le précédent état interne (c_{t-1}, h_{t-1}) ainsi que l'entrée séquentielle x_t et redonne en sortie l'état interne courant (c_t, h_t). Cet état est utilisé pour calculer la sortie du LSTM y_t ainsi que pour fournir l'information au prochain pas de temps $t+1$. Les détails des opérations sont expliquées dans les équations fournies, mais f_t représente la porte d'oubli, i_t la porte d'entrée et o_t la porte de sortie.

Le modèle LSTM utilise un systèmes de portes (avec poids) laissant passer certaines informations à différents niveaux. C'est la cellule de mémoire qui est modifiée pour permettre cette nouvelle architecture. La figure 1.7, adaptée de Olah [2015], montre les opérations effectuées à l'intérieur de la cellule de mémoire. On utilise un état interne c_t, qui conserve

un résumé global, et une cellule de sortie h_t, qui prépare la mémoire à être utilisée pour la sortie du réseau. Les équations suivantes montrent les opérations effectuées dans la cellule de mémoire, avec les paramètres w_i, w_f, w_o respectivement associés aux portes d'entrée i_t, d'oubli f_t et de sortie o_t.

$$
\begin{aligned}
i_t &= \sigma\big(w_i[h_{t-1}, x_t] + b_i\big) && \text{porte d'entrée } (\textit{input}) \\
f_t &= \sigma\big(w_f[h_{t-1}, x_t] + b_f\big) && \text{porte d'oubli } (\textit{forget}) \\
o_t &= \sigma\big(w_o[h_{t-1}, x_t] + b_o\big) && \text{porte de sortie } (\textit{output}) \\
\tilde{c}_t &= \tanh\big(w_c[h_{t-1}, x_t] + b_c\big) && \text{état de cellule candidat} \\
c_t &= f_t * c_{t-1} + i_t * \tilde{c}_t && \text{état de cellule} \\
h_t &= o_t * \tanh(c_t) && \text{sortie de la cellule de mémoire}
\end{aligned}
$$

Notons que les paramètres w illustrent les multiplications matricielles effectuées et sont appris par descente de gradient (comme dans la plupart des réseaux de neurones). La valeur calculée à la sortie de chacune des portes permet de balancer l'information entrante avec celle déjà accumulée dans la mémoire interne. Un réseau récurrent profond est un réseau récurrent avec plusieurs couches de *mémoire* superposées.

1.6. Vue d'ensemble

Comme le projet a été motivé par une application en recherche opérationnelle, mais qu'on le résout à l'aide de techniques d'apprentissage automatique tout en restreignant le budget d'étiquettage, on fait intervenir dans ce projet beaucoup de différents réseaux et types d'apprentissage. Pour résumer, nous avons accès à un *oracle* qui permet d'étiquetter les données, mais à un certain coût. Cet oracle n'est pas accessible au temps de prédiction. Pour l'approximer, on utilise un *prédicteur*, modélisé par un MLP, dont le but est d'imiter cet oracle (entrainement supervisé classique). Or, comme créer l'ensemble d'entrainement est coûteux, on veut limiter le budget d'étiquettage en sélectionnant intelligemment les instances à étiquetter. On se retrouve donc dans un contexte d'apprentissage actif, qu'on formule comme un problème de méta-apprentissage. Le *générateur d'instances* est modélisé par un *agent de RL* qui génère directement les instances à étiquetter. Finalement, on utilise un *encodeur d'état*, modélisé par un *réseau récurrent*, afin de fournir une description pertinente de l'état à l'agent

de RL. L'apprentissage se produit donc à deux niveaux: (1) l'apprentissage du prédicteur, qui tente directement de résoudre la tâche de prédiction, et (2) l'apprentissage de l'agent de RL, qui tente de trouver comment aider le prédicteur dans son processus d'apprentissage. Ces apprentissages doivent être effectués tout en limitant le budget d'étiquettage.

Comme les données ferroviaires sont très complexes, on teste cette architecture d'apprentissage actif avec un ensemble de données jouet. L'architecture complète, qu'on nomme architecture de méta-enseignement, et la description de l'ensemble de données jouet est présentée dans l'article.

Premier article.

Meta-Teaching : Active Example Generation by Reinforcement Learning

par

Stéphanie Larocque[1], Yoshua Bengio[1], and Emma Frejinger[2]

([1]) Mila, Université de Montréal

([2]) Department of Computer Science and Operations Research and CIRRELT, Université de Montréal

.

Les principales contributions de Stéphanie Larocque à cet article sont présentées.

- Élaboration du modèle;
- Implémentation de l'architecture;
- Analyse des résultats;
- Écriture de l'article.

Yoshua Bengio et Emma Frejinger ont contribué, au travers d'idées, à l'élaboration du modèle, à l'analyse ainsi qu'à l'organisation et révision de l'article.

Abstract.

Instead of having a fixed supervised dataset, active learning techniques focus on having a flexible dataset that is enhanced during training with new instances. The key assumption is that the instances budget can be reduced if those instances are chosen wisely instead of randomly sampled. In active learning, the access to an exact oracle is mandatory to label those instances, and this oracle is assumed to be expensive. This paper propose to formulate the active learning problem into a sequential metalearning problem that uses a reinforcement learning agent to generate which instances to add to the dataset, in problems where a way to generate instances is granted. Unlike pool-based methods, where the instance to label is selected from a pool of unlabelled instances, our architecture directly generates the instance to label from a distribution. The development of this architecture is designed for applications where the cost of labels varies over examples, but we test our architecture on a toy dataset with simplifying assumptions, including using a cheap exact oracle.

Keywords: Active Learning, Metalearning, Reinforcement Learning, Exploration Strategy, Optimization problems, Oracle

1. Introduction

In typical supervised learning setting, the dataset is fixed and both the instances and labels are known from the start. Instead of using such an established dataset, active learning techniques focus on having a flexible dataset from which the supervised learning model can learn. Our goal is to provide a way to actively increase (during training) the dataset size by carefully choosing new instances. These new instances need to be relevant in terms of the amount of information the predictor can get to learn and generalize better. The purpose of the active learning framework is to obtain a good predictor performance, but with a small budget. This budget can be expressed in terms of computational cost or number of instances, or both. The access to an oracle (e.g., a human annotator or a mathematical solver) that can label the new instances is required, and this oracle is assumed to be expensive. The expensive labelling cost is the main motivating reason of active learning. Active learning techniques essentially revolve around carefully using the allowed labelling budget to achieve the best possible predictive performance.

The main contribution of this paper to the literature on active learning [Settles, 2009] is to cast the active learning task in a sequential metalearning problem using reinforcement learning to provide new relevant instances. The reinforcement learning agent learns how to output a distribution from which we can sample instances helping the predictor's learning process. Those instances are then labelled by the oracle and added to the training dataset. In this perspective, we can see this architecture as a combination of active and metalearning. Since metalearning can be defined (as in psychology) by someone/something that is *being aware of and taking control of one's own learning* [Biggs, 2011], our reinforcement learning agent effectively falls under this paradigm : it is controling the predictor's learning by providing some specific datapoints to learn from. Furthermore, it is extracting meta knowledge, such as performance measures, to ensure an efficient teaching for the learner (predictor), all this meta data being ideally captured by the reward. To exploit such meta knowledge is crucial for a metalearning algorithm [Vilalta and Drissi, 2002]. Metalearning algorithms however require multiple metaproblems to learn correctly [Lemke et al., 2015]. In this context, we use related oracles, either being the exact expensive solver or approximate cheap heuristics, as metaproblems. We sequentially provide the cheap heuristics to train the RL

metalearner agent to provide relevant instances, according to both the predictor and respective oracle. In order to spend the labelling budget wisely, we wait until the RL agent is well trained (e.g., at the end of training) to use the expensive and exact oracle to finally create the active dataset for the original task.

We assume that we have access to a way of generating new examples (e.g., from a distribution), in addition to the availability of an exact oracle. The access to such an exact oracle, assumed to be expensive, is not always granted, but it is the case in various optimization problems [Hillier and Lieberman, 2015], where the motivation of this project came from. The labelling cost is one of the main reasons restricting the use of the oracle at the time of prediction. Therefore, we train a predictor to approximate it, bearing in mind that getting the dataset labelled is expensive and must be done carefully, for example using active learning. Another particularity of the motivating example is its highly variable labelling cost. Therefore, simply minimizing the total number of labelled instances is not enough to reduce the total labelling cost. In problems where the labelling cost greatly varies, the cost (e.g., time) should be part of the budget, not only the number of instances. Otherwise, active learning techniques that do not take the labelling cost into account, when varying, might not be able to outperform random labelling [Settles, Craven and Friedland, 2008]. An important simplification is that we only use the total number of instances as budget in the results, as we test on a simplified problem using a toy dataset having a cheap oracle (fixed cost). We state the other simplifications in Section 1.3.

1.1. Motivating Application

The motivation behind this architecture came with a discrete optimization problem in operational research (OR), the Load Planning Problem, which was formulated as an integer linear programming by Mantovani et al. [2018], and that Larsen et al. [2018] have tried to solve via machine learning. In this problem, we have access to a mathematical solver providing exact solutions, but computational restrictions prevents us from using it at the time of prediction, since the mathematical solver can be very expensive. It also has a highly variable labelling cost among instances. Thus, we want to train a machine learning model to approximate the solver, but creating the supervised dataset with randomly drawn instances is too expensive. We want to reduce the computational time associated with the

dataset generation by actively creating the dataset while training, but without decreasing the predictor's performance. There is a trade-off that arise regarding both the number of instances, the cost of labelling each individual instance and the cost of the whole active generation architecture.

1.2. Related work

The overall objective between active learning (AL) and our architecture is the same, which is to reach high performance on a supervised learning network only using a small dataset, with the assumption that labelling is either time-consuming or difficult to do (or both). Another assumption is that not every datapoint is relevant for the supervised task, e.g., the sample complexity is below the number of datapoints needed in a random dataset obtained with passive learning [Settles, 2009]. As with our architecture, the active learning techniques assume the access to a solver (sometimes a human labeller) in order to actively label the unlabelled datapoints. Active learning techniques are often separated into three different categories [Settles, 2009]: pool-based sampling, stream-based selective sampling and membership query synthesis. The difference between those techniques lies in *how* to select the instance to label. Pool-based methods are widely used and refer to methods iteratively choosing examples to label from an unlabelled pool. Stream-based sampling are methods using a stream that iteratively provides unlabelled examples, and where one must decide whether to request the label or discard the example. The third category, membership query synthesis, refers to methods that directly generate instances and ask for membership or other concept-related queries [Angluin, 1988]. For every method, evaluating the relevance of labelling new instances is crucial. Criterions such as as uncertainty sampling [Lewis and Gale, 1994], query-by-commitee [Seung et al., 1992], expected model change [Settles, Craven and Ray, 2008; Cai et al., 2013] or expected error reduction [Roy and McCallum, 2001; Zhu et al., 2003] are amongst the most widely used.

In our work, we model the active learning problem as a sequential metalearning problem that uses a reinforcement learning agent to properly generate relevant instances. It uses the reward, instead of the above criterions, as an approximate measure of relevance. Other people, such as Bachman et al. [2017], also cast the active learning problem into a metalearning problem that uses RL but, to our knowledge, not to directly generate the new instances.

Instead, it is used for pool-based or stream-based active learning. Our architecure also includes a recurrent neural network to encode the state. Other projects combine (some) of these areas. Baram et al. [2004] and Hsu and Lin [2015] combine active learning with RL, but in the pool-based setting. In the latter, instead of having an established AL strategy, the RL agent iteratively chooses, at each time step, from a given set of AL strategies, using an estimate of the test performance as reward. Using the criterion associated with this particular AL strategy, it selects an example from the unlabelled pool. It differs from our architecture, since it chooses amongst AL strategies (that, in turn, choose from an unlabelled pool) instead of directly generating new examples from the input space. Using a RL agent in a metalearning setting to solve a stream-based active learning problem has also been investigated for one-shot learning [Santoro et al., 2016b; Woodward and Finn, 2017]. In the latter, they also use the LSTM architecure, but for modeling their RL agent. However, in both cases, the RL agent iteratively decides whether to query for the label or discard the example that is provided in the stream, while our RL agent directly generates which instances to add to the dataset.

Our meta active learning architecture is designed for problems having both a way to generate instances and to label those using an exact oracle, as in various operational research problems. In their paper, Nakajima and Iima [2017] also combine RL and operational research, but the RL agent gives the solution of the OR problem by sequentially providing the value of each of the decision variables, not to behave as an example generator. It differs from our project, where the RL agent is used at a metalevel : instead of directly solving the task of interest, it provides examples for the predictor to correctly learn the task, thus the meta-active learning architecture.

1.3. Objectives, contribution and structure of the paper

Instead of generating the whole dataset prior to training, we propose a novel active generation model using reinforcement learning to generate relevant instances to add to the dataset, according to the current task and its associated predictor. This methodology intends to reduce the total number of datapoints required, as in active learning techniques, for achieving a good performance on a supervised dataset. Our main contribution is to formulate the active learning problem as a metalearning problem using an RL agent that directly

generates the new instances to add, instead of selecting from a pool or from a stream. The aim is to attest whether the active generation by RL outperforms, in terms of predictor's generalization error, computation time and labelling cost, the random generation method.

We use the *Poker Hand Dataset* from the UCI Machine Learning Repository [Dua and Graff, 2017] as a first step in this direction, since it constitutes of a simplified setting where the oracle is inexpensive and exact. While it does not entirely correspond to the setting our methodology was originally designed for, it allows us to perform a first numerical analysis and validate the methodology, but with a cheap oracle. Two more simplifications were made with this dataset, namely to consider the number of instances as budget instead of labelling cost and to use the exact oracle at each episode, with the aim of reducing the bias introduced by artificial heuristics. All this modifications arise since the exact oracle is not expensive.

The remainder of the paper is structured as follows: Section 2 provides the complete description of the architecure and training algorithms, Section 3 describes the particularities of the chosen dataset, while the results and discussion are presented in Sections 4 and 5.

2. Active Generation by RL

The final goal of this architecture is to obtain the best possible accuracy and to correctly generalize from an active labelled dataset $\mathcal{D} = \{x_i, y_i\}_{i=1}^{M}$, but using a fixed small budget. Achieving the same predictive performance while using fewer datapoints is the major focus of active learning [Settles, 2009], especially when labelling is expensive. We show in Figure 2.0.1 the most important parts of this architecture, namely the *predictor*, the *example generator*, the *labeller* and the *current dataset*. The predictor learns how to correctly label the datapoints, while a second model provides new relevant instances. The latter is modeled by a RL agent aiming to help the predictor by gaining metaknowledge about its learning process. It thus controls the predictor's learning process throughout training by providing new instances, and having this kind of interaction between two distinct learning processes can be viewed as metalearning [Santoro et al., 2016b]. The supervised learning predictor always takes (x_i, y_i) pairs from the current dataset as input, and minimizes the distance between the ground truth label y_i and its prediction $\hat{y}_i$, with the help of the loss function. The major difference with standard supervised learning is that the dataset is not fixed, but rather iteratively updated during training. In order to choose which instance to add to the

dataset, we need an example generator to provide, at time step t, a relevant example x_t. In this context, we hypothesize that an instance that is initially incorrectly classified, that leads to a gain in new information and that helps the generalization process is relevant. After generating this example, we query the oracle (either an exact oracle or an heuristic) for the associated label y_t and then we add the (x_t, y_t) pair to the dataset. We assume that we have access to both a way of generating new instances (e.g., from a distribution) and a way to label those (e.g,. using a solver), even if very expensive.

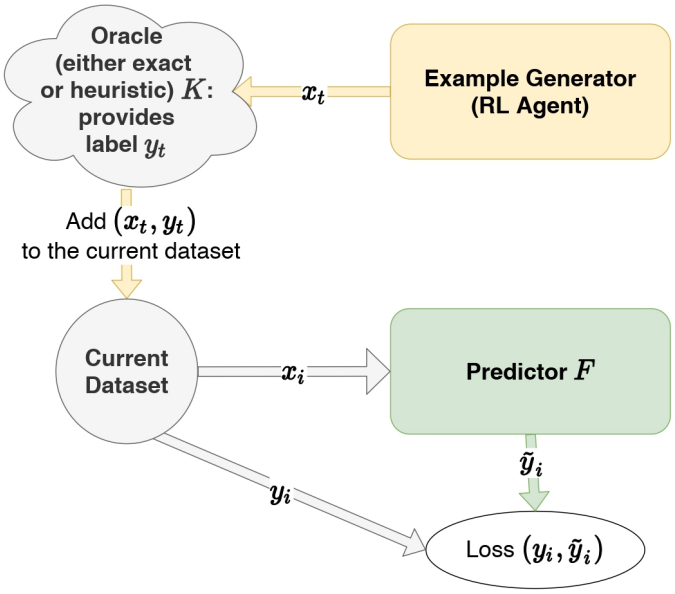


Fig. 2.0.1. Simplified Components Diagram. The subscript t is used to indicate the new instance generated at time step t, while the subscript i is used for every (x_i, y_i) pair in the current dataset D_{current} provided to the predictor.

2.1. Model Overview : RL agent as a metalearner

We propose to model the example generator using a reinforcement learning agent. Instead of having hand-crafted criterions to choose which instances to add to the dataset, as in typical active learning, the choice is left to the RL agent. It is the RL agent that iteratively chooses and provides new instances to the predictor. With that said, one can notice that the RL agent respects the basic requirements [Lemke et al., 2015] to be viewed as a metalearner, namely the presence of a learning subsystem in the active architecture

and the experience gain by exploiting meta knowledge. The learning subsystem is the predictor, while the meta knowledge consists of, for example, performance measures extracted throughout the episodes. This architecture effectively formulates the active learning problem as a metalearning problem, and uses a reinforcement learning agent due to the sequential decisions.

Learning on different metaproblems is crucial for a metalearning algorithm to learn how to generalize [Lemke et al., 2015]. Since we assume to have access to an exact oracle to label new instances, we use heuristics as metaproblems to label the generated instances. However, recall that this approach is designed for the case where the exact oracle is expensive (otherwise, the cost of labelling would not be a focus), but it is important, in the contrary, that the heuristics are non-expensive (otherwise, it would be wiser to directly use the computing power with the exact oracle). Therefore, the RL agent cannot train with this exact oracle for the entirety of its episodes, but rather needs to learn on related tasks having cheap labelling cost such as heuristics. At each episode, a new task is given to the predictor, and the RL metalearner agent needs to efficiently help the predictor to correctly map the examples to their associated labels by providing relevant new examples.

Whilst the predictor's goal is to minimize its prediction error, the RL agent's goal consists in correctly and rapidly adapt to the current task (which is for the predictor to approximate the appropriate oracle), to spot the weaknesses of the current predictor and to develop the intuition underlined by the general family of the problems. The word *oracle*, noted as K, serves as a unifying term to refer to the two following concepts: (1) the exact oracle K_0, assumed to be expensive, and (2) the heuristics K_j, assumed to be cheap. When refering to a specific oracle, we use the according term. The environment in which the RL operates include both the task description (the oracle) and the associate learner (the predictor). It is important that the RL learns how to switch from task to task, eventually changing to the exact oracle. Note that since the predictor is always updated as the training goes, the environment is not only modified in between episodes, but also inside a specific episode.

For a given episode, the RL immediate goal is to help the predictor's learning process by providing a distribution from which we can sample relevant examples, that are then labelled with the associated target function. The RL agent thus *teaches* the predictor by selecting examples that can help both the predictor's accuracy and generalization. While

the RL agent's action is to output such a distribution, the relevance of the instance sampled is accounted for by the amount of reward the agent gets. The RL state will be encoded using an auxiliary recurrent network. However, our main focus revolves around the active example generator, including both the RL agent definition (input, output, reward and state encoding) and the meta-training loop around it.

2.2. Architecture

Three neural networks are used in the active architecture (the predictor, the RL agent and the state encoder), as well as two types of oracles (the exact oracle and the heuristics). We use the term *task* when referring to the predictor's task, which is to approximate the appropriate oracle. An overview of the models specificities (inputs, outputs and goals) and motivation between the different targets are provided in this section, while the full interaction diagram can be found in Figure 2.2.1. The notation is summarized in Table 2.2.1.

The *predictor* (or learner) F is a fully-connected neural network that learns how to correctly classify the instances provided and, furthermore, to generalize from the available dataset. Since we consider a classification task, it learns its parameters φ using the cross entropy loss function between the ground truth label y_i and its prediction $\hat{y}_i = F(x_i; \varphi)$.

The *RL agent* (used as an instance generator) is modelled by an *actor-critic* agent taking a state S_t as input that describes both the predictor's current state and the task at hand. The actor-critic architecture consists of an fully-connected neural network with shared layers between the actor and the critic, and two heads. One of those heads gives the probability vector from which we sample the new instance x_t (actor) and the other head gives the value function (critic). We use the following notation respectively for the actor and the critic :

$$\pi(\,\cdot\,;\theta) \text{ and } \hat{v}(\,\cdot\,;\omega),$$

with trainable parameters θ and ω (some of these being shared). The actor's goal is to provide a distribution from which we can sample relevant new instances. The reward should capture the relevance of the new generated instance, which means being able to learn something useful from it. To train the actor to maximize the expected return, we use the REINFORCE with baseline algorithm [Williams, 1992], derived from the Policy-Gradient Theorem [Sutton et al., 2000], but with the bootstrap approximation of the return. The critic's goal is to output a scalar number representing the state's value, thus minimizing the distance between

Tab. 2.2.1. Notation

$F(\,\cdot\,;\varphi)$	Predictor, with parameters φ
$\pi(\,\cdot\,;\theta)$ and $\hat{v}(\,\cdot\,;\omega)$	Instance generator, modelled by an actor-critic RL agent, with respective trainable parameters θ (actor π) and ω (critic $\hat{v}$)
$E(\,\cdot\,;\mu)$	State encoder (LSTM) with parameters μ
K	Oracle (unifying notation for either K_0 or K_j)
K_0	Exact oracle
K_j	Heuristic used at episode j
$\mathcal{D}_{\text{train}}$, $\mathcal{D}_{\text{valid}}$, $\mathcal{D}_{\text{test}}$	Training, validation and test datasets
$\mathcal{D}_{\text{init}}, \mathcal{D}_{\text{current}}, \mathcal{D}_{\text{final}}$	Initial, current and final training datasets for a given episode
(x_i, y_i)	Example/label pairs from the dataset
$\hat{y}_i$	Label prediction from the current predictor, $\hat{y}_i = F(x_i;\varphi)$
e_i	Error on example x_i: $e_i = \text{Loss}(y_i, \hat{y}_i)$
(S_t, A_t, R_t, S_{t+1})	Typical reinforcement learning notation with current state/action/reward/next state
(x_t, y_t)	Example/label pairs generated at time step t
Acc_t^{valid}	Accuracy on the validation dataset at time step t
Acc_t^{inst}	Accuracy on the newly generated instances at time step t

its approximation $\hat{v}(S_t;\omega)$ and the corresponding bootstrap target $R_t + \gamma\hat{v}(S_{t+1};\omega)$. We use the semi-gradient update rule to train the critic [Sutton and Barto, 2018]. Since our final goal is to obtain a predictor with good generalization capacity, the RL agent must have a signal to reduce the generalization error. There is multiple ways to capture this, e.g., to use the validation accuracy

$$R_t = Acc_t^{\text{valid}}$$

as reward, with Acc_t^{valid} being the accuracy on the validation dataset at time step t. We also study the effect of adding other terms to the reward, such as the immediate accuracy on the newly generated instances. A subgoal of the agent being to generate instances that are not already being correctly classified, we want to minimize the accuracy on the new instances while maximizing the validation accuracy. We evaluate alternative rewards such as

$R'_t = Acc_t^{\text{valid}} - Acc_t^{\text{inst}}$, where Acc_t^{inst} is the instant accuracy of the new instances before any update. We can separate this alternative reward into a sparse and delayed reward (long-term reward Acc_t^{valid}) and a dense proxy (short-term reward Acc_t^{inst}).

The last component is the *state encoder*. We consider the evolution of training and network performance as sequential data. Therefore, using a recurrent neural network to model the state encoder is a natural choice, as it takes sequential data as input [Goodfellow et al., 2016]. Its goal is to keep and provide a relevant state description S_t from which the RL agent can distinguish important features of the task at hand and the predictor's associated weaknesses (e.g., the task-predictor information). More specifically, the state encoder is modelled by a Long Short Term Memory network (LSTM, [Hochreiter and Schmidhuber, 1997]). It sequentially takes as input the tuple

$$(x_i, y_i, \hat{y}_i, e_i) = \text{(example, ground truth label, prediction, error)}$$

representing both the quality of the predictor's prediction on the specific example x_i and the general task at hand. We use y_i as the ground-truth label, $\hat{y}_i$ as the prediction and e_i as the error value on this specific example. This tuple evolves throughout the training, as the predictor gradually improves at its task. The state encoder goal is to provide a relevant state description S_t (its hidden state), using the backpropagation of the critic's loss into its own network to update its learnable weights μ so that, for instance, the critic can output a proper state value that helps, in turn, the agent to behave correctly. We do not focus on this neural network, as it is mainly used as an helper to get a relevant state S_t.

The RL meta learner needs to see multiple metaproblems to learn correclty. Those metaproblems consist of the various oracles. The access to an *exact oracle* K_0 is central to this approach. It is considered ground truth and is never altered during training. We assume that this exact oracle has an associated labelling cost higher than what our budget allows. It can be restricting the dataset size in a way that prevents appropriate generalization. Thus, the exact oracle K_0 must be use sparingly, hence the motivation behind the use of heuristics. These heuristics K_j can be hand-crafted or approximations of the exact oracle, for example. If we take the variable labelling time into account in the budget, we can define those heuristics on the input space either by defining hand-crafted heuristics or by using the exact oracle itself, but restricted to a very small computation time, or by allowing the RL agent to generate instances only from cheap parts of the input space. In a varying labelling

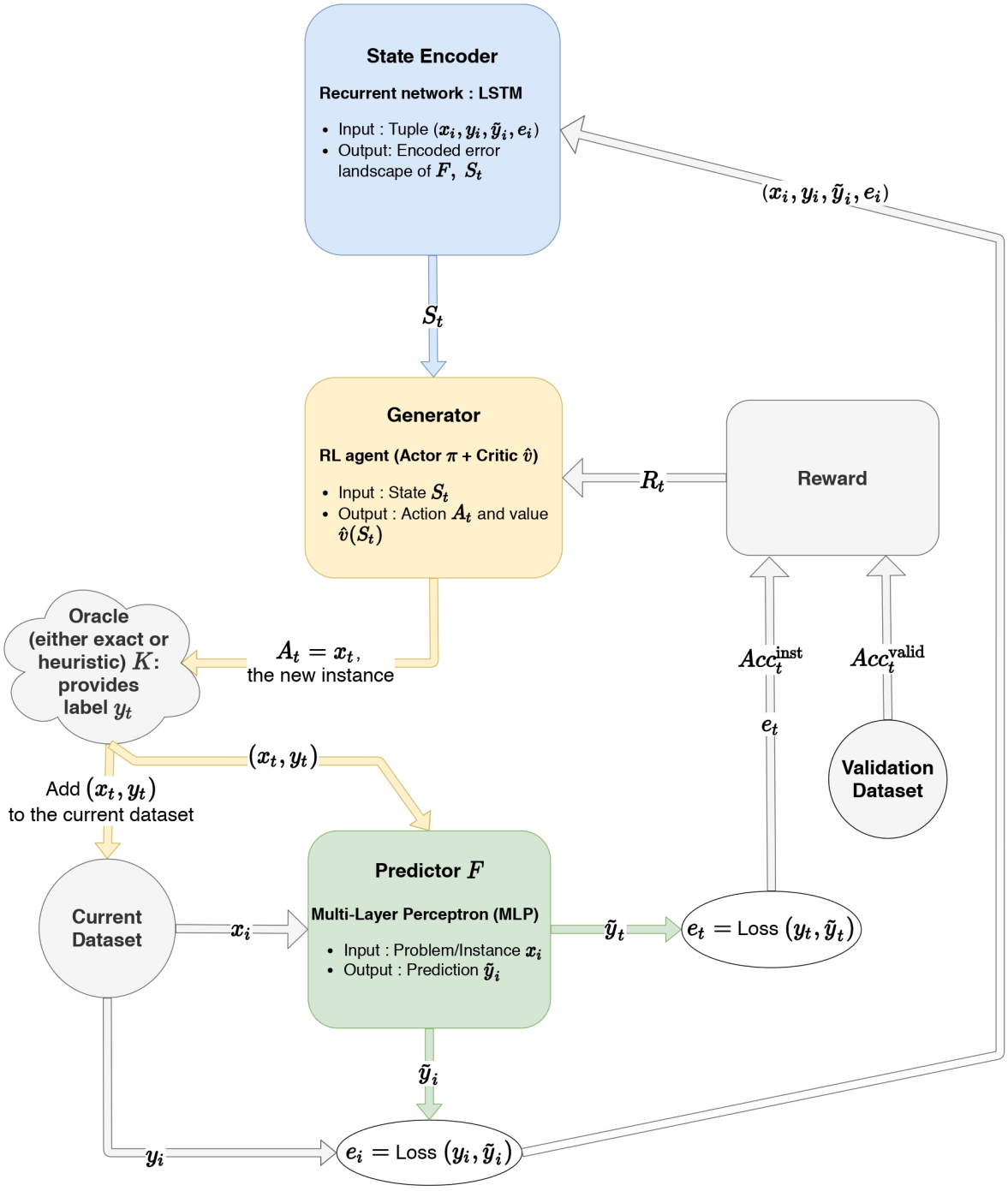


Fig. 2.2.1. Components Diagram. Note that there are two *timeframes* represented in the diagram. For instance, the RL agent outputs a new example x_t at every time step t (in an episode) and, in between those generations, a few training epochs take place to train the predictor. The subscripts t indicates that this action is performed at each time step t of an episode, while the subscipts i indicates this action is done at each time a new pair (x_i, y_i) is given to the predictor (so, multiple times for each time step).

cost setting where time is considered as budget, we can have more instances for a same budget if we separate the cheap and expensive instances.

Since the task changes during training, essentially at each episode, we need to have different datasets. At each episode, we have an small labelled dataset, created by randomly sampling instances x_i that are then labelled with the oracle K, such that $y_i = K(x_i)$. We divide this labelled dataset into the initial training dataset $\mathcal{D}_{\text{init}}$ and the validation dataset $\mathcal{D}_{\text{valid}}$. The training dataset is used for the supervised task of the predictor and is increased with new instances $\big(x_t, K(x_t)\big)$ during the training. The validation dataset is used in both the reward computataion as an estimate for test performance and in early stopping as a regularisation purpose. We do not take the validation dataset size into account in our budget as we take a big dataset in order to reduce the variance in the reward.

2.3. Training phases

For a given episode, we consider the oracle, either if it is exact or an heuristic, as the target of interest. This specific task is a new example for the metalearner, at each episode. As presented in Figure 2.3.1, each episode is split into three phases : (1) the pre-training phase, (2) the active generation phase and (3) the post-training phase. Having an initial dataset $\mathcal{D}_{\text{init}}$, the pre-training consists in training the predictor in a totally supervised setting. At the same time, we also update the RNN's hidden state in order to have an hidden state in line with the current task. The active example generation part is where the RL agent provides new examples x_t and where we sequentially update the dataset with relevant pairs (x_t, y_t), the latter being obtained with the target K. Before generating another new instance, the predictor is trained on the current dataset, thus enabling a way to attest the previously generated instance's relevance. The RL agent is also updated during this phase. The active generation part is limited by the predefined instances budget. The last part - the post-training - consists of additional *cheap* training for the predictor in a fully supervised training fashion (no instance generation or expensive labelling is happening during this phase). Otherwise there is limited amount of training for the last generated instances.

The only difference between episodes is which metaproblem is chosen : either the exact oracle K_0, or the heuristic K_j. See Figure 2.3.2 for visual explanation of the global training. We use these different tasks to train the RL on adapting to new tasks while keeping an

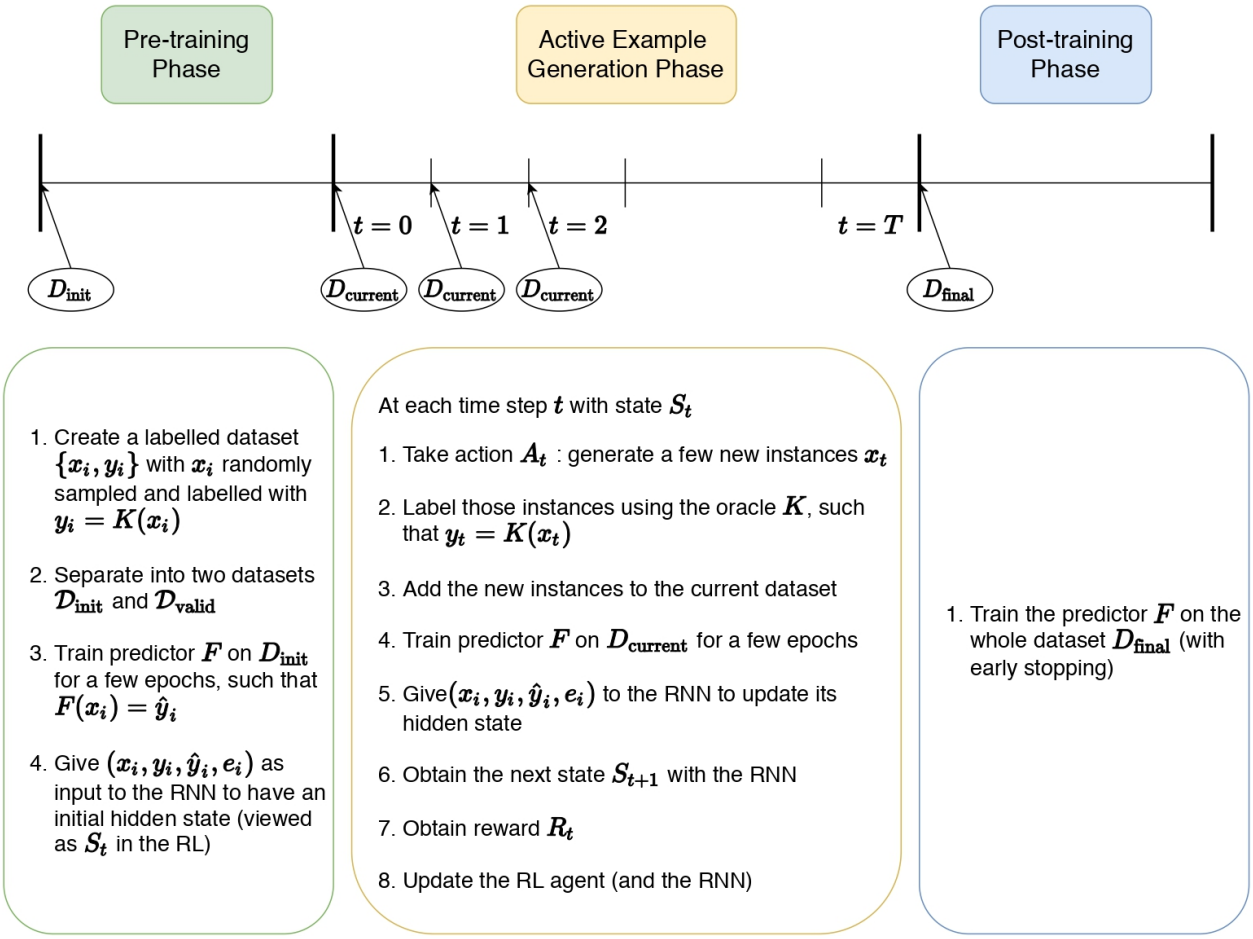


Fig. 2.3.1. Description of the three phases of an episode using oracle K (either exact or heuristic)

internal intuition of the general family of the different tasks. Furthermore, the state encoder also learns how to get accurate information from both the target and its associate predictor to provide a relevant state description to the RL metalearner. We use the heuristics on all episodes, except the last one. By doing this, we focus our labelling budget on the real oracle when the RL generator is technically well-trained and yields only relevant examples, instead of wasting it on an expensive oracle at the beginning of training. Since the heuristics change from episode to episode, the RL and state encoder are not surprised when we give the true oracle as target in the last episode. However, note that we make an important simplification on the role of the exact oracle in our results. Since we use a dataset with a cheap yet exact oracle, this exact oracle is used at every episode to limit the possible bias introduced by heuristics. Therefore, we don't consider heuristics in numerical results, but it is a main part

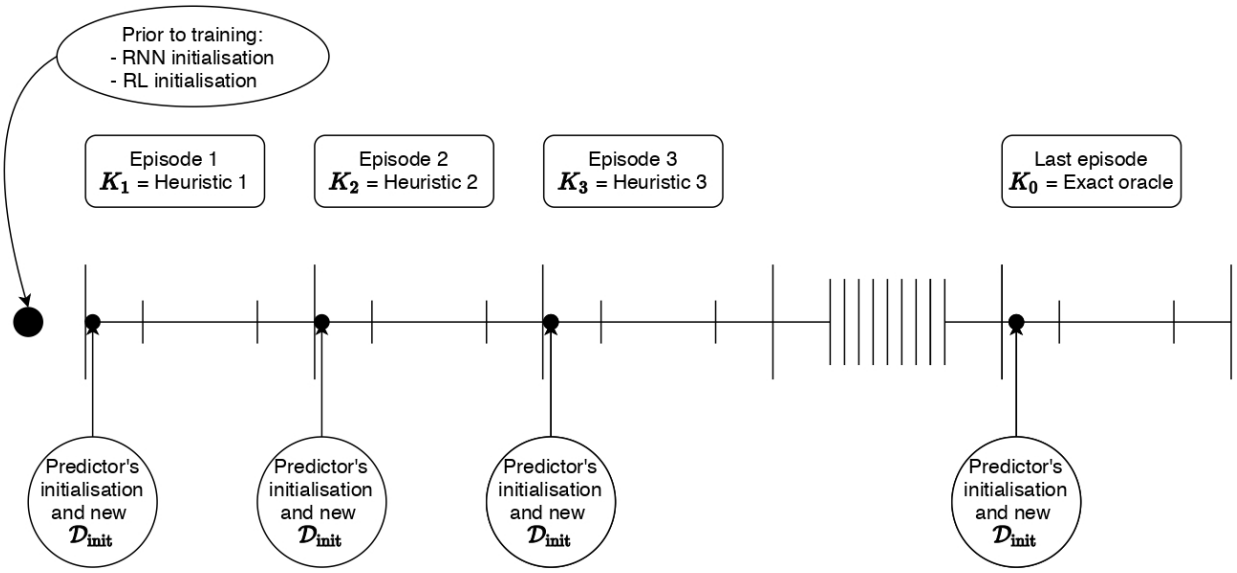


Fig. 2.3.2. Complete training Diagram. The oracle at episode j is noted as K_j, either being the exact solver or an heuristic. The actor-critic and the state encoder (RNN) are initialized only once, at the beginning of the training. However, since every new episode involves a new related task to understand, a new predictor is initialized at each episode. For the same reason, a new initial dataset must be created at each episode (and labelled using the associate oracle). The oracles (either exact or heuristics) are given at the beginning of the training and never altered.

of the architecture when the exact oracle is expensive (e.g., for every dataset except a toy dataset).

Note that, since there is multiple episodes (in which we train a predictor multiple epochs in each episode, in addition of the agent's learning and the oracle labelling cost associated with the high number of new examples to label), the architecture is heavy in terms of compuation time. We justify the computation time required and the number of datapoints used in the whole RL training loop by the fact that we try to learn way more than to fit (x,y) pairs, but also a way of actively exploring the input space during training, using reinforcement learning as our exploration strategy in a metalearning manner, which is the main contribution of this paper.

2.4. Training algorithms

In this section, we present the algorithms needed throughout training. In Algorithm 1, we describe the main training loop. However, to simplify the code, some details were put

Algorithm 1 Main training loop

Require: Oracles : exact solver K_0 and heuristics K_j (not trained in this loop)

Require: Total budget size M and initial dataset size m

Require: Number of time steps T in each episode, defined by the budget size

Require: Respective learning rates η and discounting factor γ

Initialize actor π with parameters θ

Initialize critic $\hat{v}$ with parameters ω

Initialize state encoder E with parameters μ

for $j = 0, 1, \ldots$ number of episodes allowed **do**

  {*Pre-training phase*}

  Define K (either exact or heuristic) as the episode's oracle

  Randomly generate instances x_i and label using $y_i = K(x_i)$

  Separate the labelled dataset into $\mathcal{D}_{\text{init}} = \{x_i, y_i\}_{i=1}^{m}$ and $\mathcal{D}_{\text{valid}}$

  Initialize predictor F with parameters φ

  Train predictor on the initial dataset $\mathcal{D}_{\text{init}}$ and observe initial state S_0 (pre-training phase with state encoder, Algorithm 2)

  {*Active generation phase*}

  for $t = 1, \ldots, T$ **do**

    Take action $A_t \sim \pi(S_t; \theta)$, which is to generate new instance x_t

    Label the new instance using $y_t = K(x_t)$ (might be expensive)

    Update current dataset $\mathcal{D}_{\text{current}} \leftarrow \mathcal{D}_{\text{current}} \cup \{(x_t, y_t)\}$

    Train predictor F on $\mathcal{D}_{\text{current}}$ and observe next state S_{t+1} (see algo 2)

    Observe reward R_t (see below)

    Compute TD error $\delta_t = R_t + \gamma \hat{v}(S_{t+1}; \omega) - \hat{v}(S_t; \omega)$

    Update actor's parameters : $\theta \leftarrow \theta + \eta_\theta \ \delta_t \ \nabla_\theta \ln \pi(A_t | S_t; \theta)$

    Update critic's parameters : $\omega \leftarrow \omega + \eta_\omega \ \delta_t \ \nabla_\omega \hat{v}(S_t; \omega)$

    Update RNN's parameters : $\mu \leftarrow \mu + \eta_\mu \ \delta_t \ \nabla_\mu \hat{v}(S_t(\mu); \omega)$

  end for

  {*Post-training phase*}

  Train predictor on the whole generated dataset $\mathcal{D}_{\text{final}}$ with early stopping using the validation dataset (typical supervised learning)

end for

Keep statistics such as accuracy and reward for further analysis

aside in this pseudocode, namely how to get the state S_t (Algorithm 2), how to get the action A_t (and new instance x_t) and, finally, how to get the reward R_t. Since our environment is not simple, obtaining the state S_t, action A_t or R_t is not straightforward either. First of all, in order to obtain the state S_t we need to train the predictor on the current dataset for a certain number of minibatches. For every (x_i,y_i) pair given to the predictor, we also provide $(x_i, y_i, \hat{y}_i, e_i)$ tuples to the state encoder to update its hidden state, where $\hat{y}_i = F(x_i; \varphi)$ is the predicted label and e_i is the corresponding loss. When needed, we can thus query the state encoder for the new state S_t. See Algorithm 2 for details[2].

Algorithm 2 Predictor training loop, including procedure to obtain next state S_{t+1}

Require: Predictor F with parameters φ and learning rate η_φ

Require: State encoder E (with parameters μ, not trained in this sub-algorithm)

Require: Episode time step t

1: **for** ℓ minibatchs **do**
2: $\quad\Delta \leftarrow 0$
3: $\quad$**for** (x_i, y_i) in minibatch **do**
4: $\quad\quad\hat{y}_i \leftarrow F(x_i; \varphi)$
5: $\quad\quad e_i \leftarrow$ CrossEntropy Loss $(y_i, \hat{y}_i)$
6: $\quad\quad$Accumulate predictor's gradient : $\Delta \leftarrow \Delta + \nabla_\varphi e_i$
7: $\quad\quad$ {*Update LSTM's hidden state, considered as the RL state*}
8: $\quad\quad S_{t+1} \leftarrow E\,(x_i, y_i, \hat{y}_i, e_i\ ; \mu)$, the hidden state of the state encoder E
9: $\quad$**end for**
10: $\quad$Update predictor's parameters : $\varphi \longleftarrow \varphi - \eta_\varphi \Delta$
11: **end for**
12: **return** Updated predictor F, state encoder E and next state S_{t+1}

To take action A_t (Line 13, Algorithm 1) means to obtain a new x_t instance. It implies to

(1) Pass through the actor and get the output distribution

(2) Sample one or more instances from this distribution

(3) Make sure the instances respect some restrictions, if applicable

[2]Note that, except the LSTM's hidden state update at Line 8 (of Algorithm 2), this is a typical supervised learning algorithm for the predictor. The pre-training phase includes the update of the LSTM's hidden state, while it is omitted in the post-training phase.

Afterwards, we label the new instance using the respective oracle K and add the (x_t,y_t) pair to the current dataset. The choice of reward R_t is not straightforward either. The reward must help the RL agent to learn how to provide relevant examples and help the learning process of the predictor. The RL agent should minimize the generalization error by maximizing the reward, approximated by the validation accuracy. But, to compute such a reward, one must use the validation dataset and compute its respective accuracy with the new updated predictor, and any additional term in the reward definition will result in more computation or more statistics to keep while training. Therefore, the obtention of the reward is not immediate. Furthermore, since the oracle and predictor both change during training, the environment is dynamic, thus the reward must pursue a moving target, which is a challenge.

The losses and parameter updates of the predictor, the reinforcement learning agent and the RNN are be explained below. Since the predictor has to achieve a multi-class classification task, it tries to minimize the cross-entropy loss as objective (Line 10, Algorithm 2). The actor update rule (Line 19, Algorithm 1) is the typical update rule for the REINFORCE algorithm with baseline using the bootstrap target. The expected return is approximated by the one-step TD target $R_t + \gamma\ \hat{v}(S_{t+1};\omega)$ and the baseline is the approximated value function $\hat{v}(S_t;\omega)$. Therefore, the update rule becomes

$$\theta \leftarrow \theta + \eta_\theta\ \delta_t\ \nabla_\theta \ln \pi(A_t|S_t;\theta),$$

with $\delta_t = R_t + \gamma\hat{v}(S_{t+1};\omega) - \hat{v}(S_t;\omega)$ being the temporal difference error and η_θ, the learning rate. The critic is updated using the mean squared error loss (MSE) between its prediction and the state real value. Since we cannot have the state *real* value (otherwise we would not need a critic), we consider the bootstrap target as the real state value and prevent the gradient to flow through it, as in semi-gradient techniques [Sutton and Barto, 2018]. We note the predicted state value as $\hat{v}(S_t;\omega)$ and the bootstrap target as $R_t + \gamma\ \hat{v}(S_{t+1};\omega)$. The update rule for the critic (Line 20, Algorithm 1) thus becomes

$$\omega \leftarrow \omega + \eta_\omega\ \delta_t\ \nabla_\omega \hat{v}(S_t;\omega),$$

with η_ω being the learning rate. The state encoder's loss (Line 21, Algorithm 1) is similar to the critic loss. Actually, the purpose of the LSTM state encoder is to provide a relevant state description from which the critic (and the actor) can distinguish important features of

both the current oracle and predictor. We hypothesize that a good LSTM helps the critic to be precise, and then the actor to be good as well. Therefore, the LSTM is trained with the critic's loss, but with the gradient flowing from the loss through the critic, and from the state description through the LSTM, meaning that the mean squared error loss function is propagated up to the LSTM's weights. Note that $\hat{v}(\ \cdot\ ;\omega)$ can also be seen as $\hat{v}(\ \cdot\ ;\omega,\mu)$, since $S_t = S_t(\mu)$. As seen in Line 21, we get the following update rule

$$\mu \leftarrow \mu + \eta_\mu\ \delta_t\ \nabla_\mu \hat{v}(S_t(\mu);\omega),$$

where $\nabla_\mu \hat{v}(S_t(\mu);\omega) = \nabla_\omega \hat{v}(S_t;\omega)\ \nabla_\mu S_t(\mu)$ and η_μ is the learning rate.

2.5. Spotcheck Loop : Motivation and Description

Between each episode, the environment in which the RL agent is changes : the oracle is different, thus changing the difficulty of the problem to solve, which, in turn, modifies the ability of the predictor to accomplish its task. It can result in either an easier or harder task to achieve, both for the RL agent and the predictor, and this dynamic environment poses a challenge for the reward. The moving target causes the reward to be difficult to define and analyse. Therefore, we need a way to attest of the RL agent's learning progress by comparing rewards between episodes, but bearing in mind that the environment changes in between episodes. To address this issue, we propose to use a *spotcheck loop*. Its only purpose is to attest the quality of the RL agent and its learning progress. This training loop consists of one particular episode, but outside of the main training loop. For a same task and predictor, the RL should become better at selecting examples to add to the current dataset in between episodes. To attest of its performance, we keep the same predictor and oracle and compare the RL agents and their exploration strategies obtained throughout the main training loop.

Therefore, in every spotcheck loop of a specific experiment, we use the same oracle and reinitialize the predictor to a fixed pretrained predictor (it is possible, since we use the same oracle for every spotcheck episode and thus the predictor is always trained on the same initial dataset). The RL and state encoder are the current ones from the main training loop. Since the initial environment is always identical (same oracle, same initial dataset, same pretrained predictor), we are able to compare the quality of the RL between spotcheck episodes, which means throughout the main training loop. The only difference between spotcheck episodes

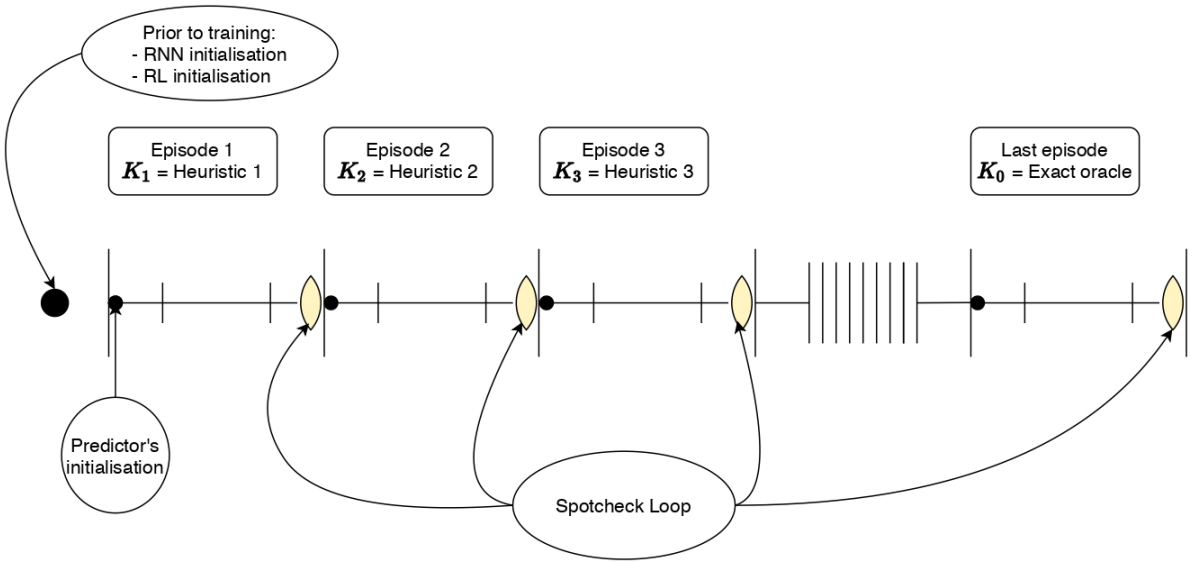


Fig. 2.5.1. Complete training with spotcheck loops

is the initial RL and state encoder parameters at the beginning of the episode. As seen in Figure 2.5.1, the spotcheck training loop happens at the end of every other episode. The pseudocode of a specific run of the spotcheck episode is provided in Algorithm 3. Using this spotcheck episode, we can compare the evolution of the RL agent's performance when always applied to the same oracle and predictor, both by analysing the reward and the validation accuracy.

Algorithm 3 Spotcheck Loop Episode

1: Reset predefined spotcheck oracle
2: Load predefined initial dataset associated with this oracle
3: Reset predictor's weight to predefined pretrained weights
4: Set RL and state encoder's weights to their parameters in the main training loop
5: Perform the active generation phase
6: Perform the post-training phase
7: Compute and keep statistics for further analysis
8: Reset RL and state encoder's weights to their parameters before this spotcheck loop
9: Resume main training loop

3. Poker Hand Dataset

The original operational research task that started the idea of this architecture is complex, both because of the expensive oracle (using an exact mathematical solver) and the variable labelling cost between instances. We choose the *Poker Hand Dataset* form the UCI Machine Learning Repository [Dua and Graff, 2017] as a first step towards our goal. Poker hands need to be classified among 10 classes. This dataset respects the two requirements stated before, namely that there exists a way of generating new hands (by sampling 5 cards) and an exact oracle able to label those instances. Note we can obtain the real poker hand's strength instantaneously and without any neural network, and this is why we consider this dataset as a toy dataset. However, we choose this dataset precisely because the real labels can be obtained easily and this is helpful to test the architecture. This is the first two simplifications of the model : to use a dataset that (1) has a exact cheap oracle and that (2) does not have variable labelling cost. Therefore, our budget can be expressed by simply using the total number of instances (instead of incorporating a labelling cost/time). The third modification is to use, for every episode, the real exact solver instead of artificial heuristics, thus avoiding potential errors introduced by heuristics. We describe the dataset in Section 3.1 and its imbalanced particularities in Section 3.2.

3.1. Dataset description

This dataset is a supervised dataset consisting of poker hands and their associated strength. Each hand consists of 5 cards, each one represented by its value (Ace to King) and its suit (spade, heart, diamond or club), both encoded in one-hot vectors. The class of a hand is a categorical value (0-9) representing the actual strength of this hand in poker (0 for nothing, 1 for one pair, etc., up to 8 for Straight Flush and 9 for Royal Flush). The goal is to correctly classify the hand's strength using the rank and suit attributes of each card. Note that we consider that the order of the cards is important, as it increases the number of different possible hands.

The exact oracle is straightforward to implement, as it is sufficient to check if the hand has the attributes of each class. We explain below how to generate a new hand. Since the example generator uses an actor-critic RL agent, we define the output of the actor as a probability vector (of size 52, for each possible card of a regular deck). We use this

probability vector in a Multinomial distribution from which we sample $n = 5$ cards without replacement[3]. We can then compute the probability of getting the cards C_1, C_2, C_3, C_4, C_5 with:

$$P(C_1 \cap C_2 \cap C_3 \cap C_4 \cap C_5) = P(C_1)P(C_2|C_1) \dots P(C_5|C_1 \cap C_2 \cap C_3 \cap C_4).$$

3.2. Imbalanced classes

One of the interesting properties of the *Poker Hand Dataset* is its highly imbalanced classes (a Royal Flush is indeed not as frequent as a pair). The proportion of each class in the initial training dataset are given in Table 3.2.1.

Tab. 3.2.1. Proportion of each class in $\mathcal{D}_{\text{train}}$

(0) Nothing	49.95%	(5) Flush	0.22%
(1) One pair	42.38%	(6) Full House	0.14%
(2) Two pairs	4.82%	(7) Four of a kind	0.02%
(3) Three of a kind	2.05%	(8) Straight flush	0.02%
(4) Straight	0.37%	(9) Royal flush	0.02%

The methods that address the imbalanced classification problem can be generally separated into two groups, or into a combination of both [Kotsiantis et al., 2005]. The first group of methods is based on techniques aiming to change the distribution by resampling the dataset, for example by undersampling the most frequent classes, or oversampling the less frequent ones. These techniques cannot be used with our architecture since the RL agent chooses the instance to add to the dataset by itself. By resampling the dataset, we would undo the effect of the RL agent. Therefore, we focus on the second group of methods, which consists of penalized models. It means that we use cost-sensitive classifiers with a varying weight penalty on the classes [Huang et al., 2016], the weight being inversely proportional to each class proportion in the training dataset. We thus study the effect of imbalanced weights correction in the cross-entropy loss. All these techniques are designed to have a greater hand

[3]Actually, it is coded using 5 Multinomial distributions from which we sample $n = 1$ card each. We use the same probability vector for every Multinomial distribution, except that a zero-probability is assigned to the alreadry drawn cards, as it is impossible to have two identical cards.

diversity, or to help with the learning process of some of the less frequent classes. Furthermore, since the accuracy score is often not the best metric for imbalanced classification tasks [Barandelaa and Rangela, 2002], we also provide measures such as precision, recall and f1-score. For a specific class i, those performance measures are defined as follows :

$$\text{precision} = \frac{\text{Number of instances correctly labelled } i}{\text{Number of instances predicted as class } i}$$

$$\text{recall} = \frac{\text{Number of instances correctly labelled } i}{\text{Number of instances of class } i \text{ in the dataset}}$$

$$\text{f1-score} = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

4. Results

We give the hyperparameters in Section 4.1, while we present the comparison with the benchmarks in Section 4.2.

4.1. Hyperparameters

In the following experiments, we use a predictor with 3 hidden layers of 50 neurons with 10% dropout, a learning rate of 3e-4, weight decay of 1e-5 and no correction on the weights (we observed that the learning process is worse using the imbalanced class weight correction). There are 500 hands in the initial dataset, and we add 2,500 other hands during the active generation phase. Note that we only use 3,000 instances because we want to test the architecture when the instances budget is very low, assuming that the budget is restricted by the expensive cost of labelling new instances. Obviously, labelling the poker hands is not costly, since this is a toy dataset, but we assume that the task of interest has an expensive solver, as stated in Section 1.

When using the active generation by RL, we train for 50 episodes and we use the reward

$$R_t = Acc_t^{\text{valid}},$$

with Acc_t^{valid} being the validation accuracy after optimizing with the current dataset at time step t (after adding the new instances generated at time t). The RL agent is an actor-critic that has one shared layer of size 64, followed by two heads (one for the actor and one for the critic) of 2 layers of size 64. The actor-critic architecture uses 5% dropout and a learning rate of 1e-4 with 1e-5 weight decay penalty. The state encoder is a LSTM with 2 hidden

layers of size 100, learning rate of 3e-4 and weight decay penalty of 1e-5. There is 200 epochs during the pre-training phase and a maximum of 2000 epochs for the post-training phase (with 200 epochs patience).

An important modification to our original architecture is that we use the real solver at each episode instead of heuristics. This allows us to focus the analysis on learning without potential bias introduced by the heuristics. Therefore, there is no spotcheck loops nor heuristics in the following results. Obviously, this technique could not work with a real dataset using an expensive solver. The *Poker Hand* solver has the advantage of being cheap. Recall that we use this dataset to get an intuition on what is happening during training.

4.2. Comparison between active RL generation and benchmarks

The active generation process with RL agent is intended to increase the accuracy given a fixed dataset size, compared to a random generation approach. To attest of its quality, we compare our active generation method with two different benchmarks : (1) the whole active generation architecture, but with a random agent instead of the RL agent and (2) the standard supervised learning procedure, with the dataset completely generated prior to training. We compare the accuracy score on the validation and test datasets. There is 5002 hands in the validation dataset[4] and 1,000,000 hands in the test dataset[5].

For every method, the accuracy score is obtained as follows. In the active generation with the RL agent, we average the accuracy score of the best predictor, for each of the last 5 episodes. We average these scores because of the volatility in the training process. For the random generation process, we use the same architecture, except that the instance generator is completely random. Therefore, we do 50 episodes, but with a random agent. The accuracy score of the best predictor of the last 5 episodes is averaged. Note that, since the agent is random, it would be sufficient to only run the experiment for 5 episodes and then

[4]Note that, when using an expensive oracle, the validation dataset should be small as the instances are labelled. However, we use a big validation dataset in order to avoid high variance in the reward in our toy experiment (and since the labelling is cheap).

[5]Note that *some* of the hands generated during the active generation phase might also be present in the test dataset, but this is negligible, considering the large size of the test dataset (1,000,000) comparatively to the small amount of generated instances (2,500). We could have removed the generated hands from the test dataset, but it was not done in this paper.

take the average on those. We made 50 episodes in order to have a qualitative comparison when showing figures. Lastly, we compare the active generation procedures (either with RL or random agent) with standard supervised learning, where all the dataset is generated prior to training. In that case, we average the performance on 5 runs, using the same hyperparameters.

	Valid accuracy		Test Accuracy	
Method	mean	std	mean	std
Active generation by RL	59.76	1.95	59.84	1.86
Random generation	60.87	2.36	60.60	2.76
Standard supervised learning	58.66	1.27	58.13	1.04

Fig. 4.2.1. Comparative results on the validation and test accuracy for the active generation by RL method, the random generation method (using a random agent instead of the RL agent) and the standard supervised learning method. Accuracy score are averaged on the 5 last episodes of each experience (or over 5 runs, for the supervised learning method).

The results of the validation and test accuracies are given in Figure 4.2.1. Unfortunately, our active generation architecture does not beat the random benchmark, where the instances are generated randomly instead of being picked by the RL agent. The active generation by RL, the random generation and the typical supervised learning methods respectively achieve 59.84%, 60.60% and 58.13% accuracy on the test dataset. These results are very similar, no matter the method used. The standard deviation is slightly smaller when using the RL agent than when using the random agent. Note that, for every method, the accuracy score on the training dataset was between 99% and 100%. The main takeaway of these results is that our active architecture at least does not perform worse than the random generator, and that there is a lot of room for improvement. Let's look in more depth and details at a typical experiment of 50 episodes.

In Figure 4.2.2 we show a complete classification report on the test dataset, using the last episode's predictor. Even though these classification reports do change a bit between episodes, this is representative of the general results obtained in most of the experiments. There is no significant difference on the predictor's behavior between the active RL and the

	precision	recall	f1-score	support
Nothing (0)	0.63	0.69	0.66	501209
1 Pair (1)	0.53	0.53	0.53	422498
2 Pair (2)	0.12	0.05	0.07	47622
3 of a Kind (3)	0.16	0.03	0.04	21121
Straight (4)	0.01	0.00	0.00	3885
Flush (5)	0.00	0.00	0.00	1996
Full House (6)	0.00	0.00	0.00	1424
4 of a Kind (7)	0.00	0.00	0.00	230
Straight Flush (8)	0.00	0.00	0.00	12
Royal Flush (9)	0.00	0.00	0.00	3
micro avg	0.57	0.57	0.57	1000000
macro avg	0.14	0.13	0.13	1000000
weighted avg	0.55	0.57	0.56	1000000

(a) Active RL generator

	precision	recall	f1-score	support
Nothing (0)	0.65	0.69	0.67	501209
1 Pair (1)	0.54	0.58	0.56	422498
2 Pair (2)	0.11	0.03	0.04	47622
3 of a Kind (3)	0.23	0.03	0.06	21121
Straight (4)	0.01	0.00	0.00	3885
Flush (5)	0.00	0.00	0.00	1996
Full House (6)	0.00	0.00	0.00	1424
4 of a Kind (7)	0.00	0.00	0.00	230
Straight Flush (8)	0.00	0.00	0.00	12
Royal Flush (9)	0.00	0.00	0.00	3
micro avg	0.59	0.59	0.59	1000000
macro avg	0.15	0.13	0.13	1000000
weighted avg	0.56	0.59	0.57	1000000

(b) Random generator

Fig. 4.2.2. Classification report for each class on the last episode with and without active generation by RL, on the test dataset.

random generator. In both cases, the predictor learns some of the underlying structure in the classes 0 and 1 (the most frequent ones), fails to label correctly most of the hands from classes 2 and 3, and completely fails the less frequent classes 4-9, even though it can correctly predict 99% to 100% of the hands in the training set.

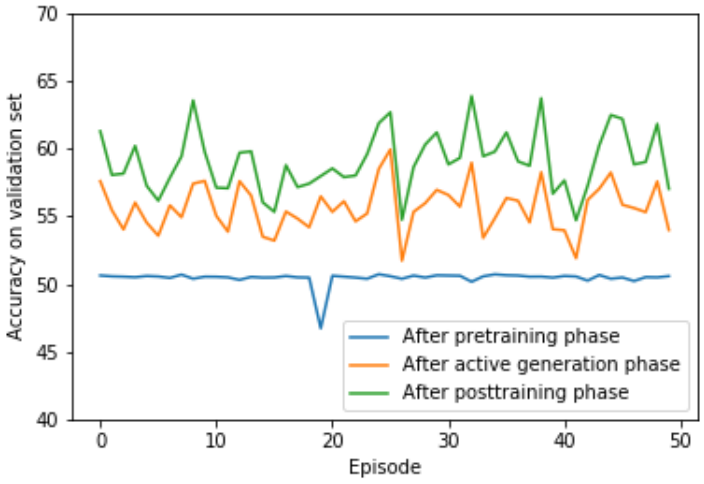
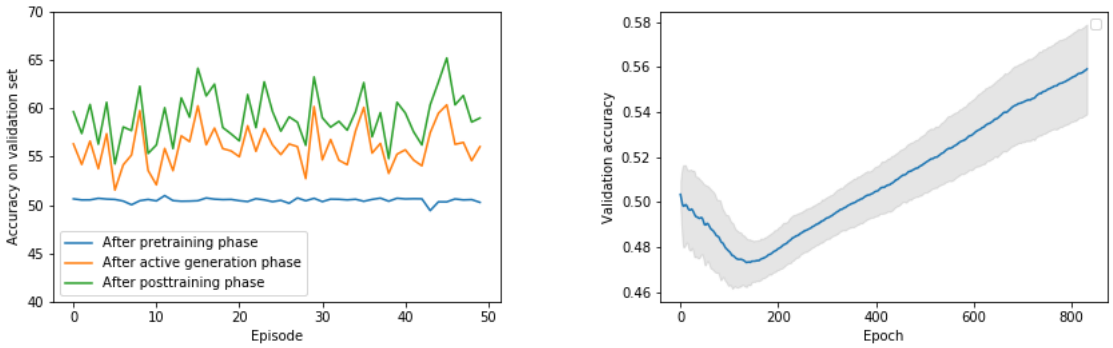


Fig. 4.2.3. Accuracy score on the validation set at the end of each of the three phases when using the RL agent. Volatility in the orange curve is explained by the fact that the training is different for both the RL and predictor during the active generation phase for every episode (sinces it relies highly on the hands being generated).

We show the evolution of the accuracy score on the validation set for each phase (pre-training, active generation and post-training phases) in Figure 4.2.3. In the pre-training phase, the predictor is trained using only the small initial dataset of size 500. We keep the accuracy score on the validation set at the end of this phase, reported with the blue line, for each episode. In the active generation phase, the predictor is iteratively trained on the current dataset (that is updated using the newly generated instances). The RL agent is also updated in this phase, using the reward to improve itself. At the end of the active generation phase of each episode, we keep the accuracy score on the validation set, represented with the orange line. Finally, the post-training phase is a typical supervised learning process where the predictor is trained using the whole dataset, so that it can learn from every instance that was generated during the active generation phase. The accuracy score after this post-training is represented by the green line, for each episode. Note that there are many training epochs for the predictor during each phase. Figure 4.2.3 shows the evolution of the accuracy score at the end of each phase, over the episodes.



(a) Accuracy on validation set after each phase

(b) Mean improvement during the active generation phase over the 50 episodes

Fig. 4.2.4. Accuracy score on the validation set when using a random generator instead of an active generator by RL. The Figure (a) shows the evolution of the validation accuracy over the three phases, for each episode. The Figure (b) shows the average accuracy on the validation set *during* the active generation phase, over the 50 episodes.

We can see a similar behavior in the evolution of the validation accuracy when using a random generator (instead of an RL agent), as pictured in Figure 4.2.4(a). Because of the

volatility on the accuracy score, we show the evolution of the validation accuracy during the active generation phase in Figure 4.2.4(b) by averaging the results over the 50 episodes (we can do this, since there is no alteration of the random agent in between episodes, unlike the RL agent). By analyzing Figures 4.2.3 and 4.2.4, we can see that the results are comparable using the RL agent or using a random generator. Therefore, the RL agent does not seem to help the learning process of the predictor more than a random example generator would. In the next section we present a few observations and discuss hypotheses on why it does not work better than a random generator.

5. Discussion and observations

5.1. Observation 1 : Generating hands already correctly predicted can make the learning process fail

In some experiences, the RL generator fails and gives hands that result in a very poor validation accuracy for the remaining of the episode, as can be seen in Figure 5.1.1. We find that this crash is always linked to the high instant accuracy of the newly generated hands. It means that, even before any learning happens on those new instances, the predictor already knows their label correctly. It thus hurts the predictor's learning by adding irrelevant instances to the dataset. At the moment of the crash, we also found out that, sometimes, the RL generator starts giving hands mainly from one particular class instead of hands from all the classes. We initially thought that it would be a good thing for the RL generator to differ from the random distribution of classes. However, it seems that, with only 3,000 hand-label available pairs, the most effective distribution, in terms of learning progress, is to generate new hands following a similar distribution to the one in the training dataset. It is probably because the predictor needs to learn the obvious classes (the most frequent ones) before taking care of the less frequent ones and that having a similar distribution in the different datasets helps.

Note that we experience similar crashes in the numerous experiments that were made. Even two runs with the exact same hyperparameters could result in one experiment having a crash while the other does not. At least, the link between the instant accuracy (of new instances) and the RL learning process is clear : when the new instances are already being correctly predicted at the time of generation, the RL sticks with this policy and it hurts the

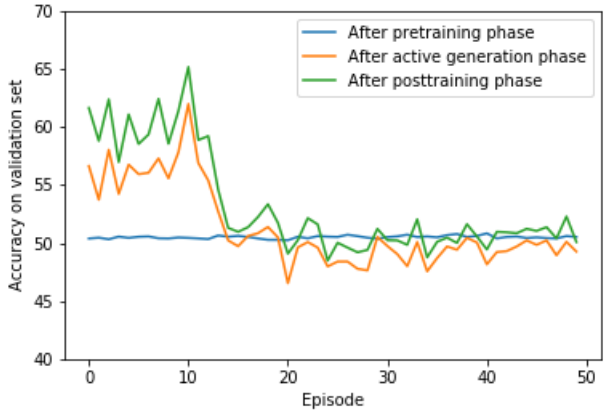


Fig. 5.1.1. Typical *crash* seen through the evolution of the validation accuracy for the 3 phases. It corresponds to the moment when the new instances start to be already correctly classified at the time of generation.

predictor learning process by overfitting to the already correctly predicted hands. To avoid it, we tried adding a term to the reward that would force the RL agent to provide hands whose labels are *not* already correctly predicted by the predictor at the time of generation. In the next subsection, we analyse the impact of the reward choice on the overall performance.

5.2. Observation 2 : Defining the reward is non trivial

We study the effect of several rewards by comparing the evolution of the validation accuracy between the three phases, for each episode. The only difference between the experiments in this section is the choice of reward. All other hyperparameters are the same than described in Section 4.1. The following rewards were tested:

(a) $R_t = Acc_t^{\text{valid}}$

(b) $R_t = -Loss_t^{\text{valid}}$

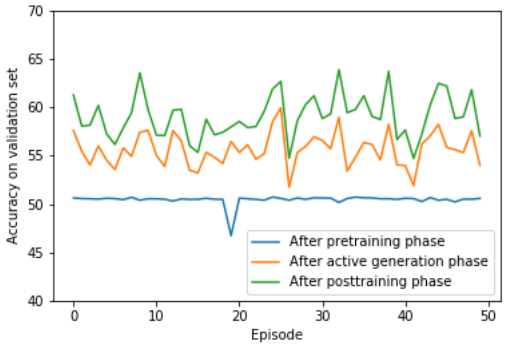
(c) $R_t = Acc_t^{\text{valid}} - Acc_t^{\text{inst}}$

(d) $R_t = Acc_t^{\text{valid}} - 0.2 * Acc_t^{\text{inst}}$

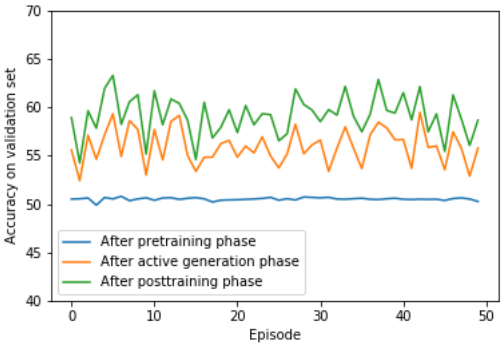
(e) $R_t = Acc_t^{\text{valid}} - Acc_{t-1}^{\text{valid}}$

(f) $R_t = -Acc_t^{\text{inst}}$

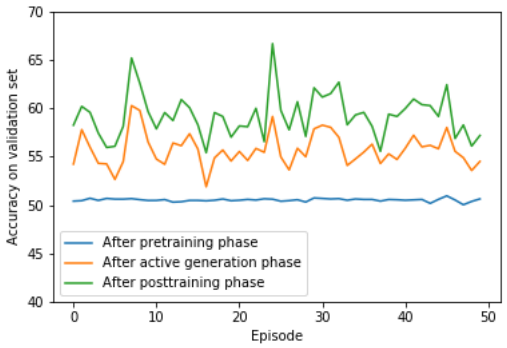
We use Acc_t^{valid}, the accuracy on the validation set at time step t as part of our reward, because this is our final goal. We also tested using the loss function value at time step t, namely $Loss_t^{\text{valid}}$. Because of the crashes, we hypothesized that adding a term about the



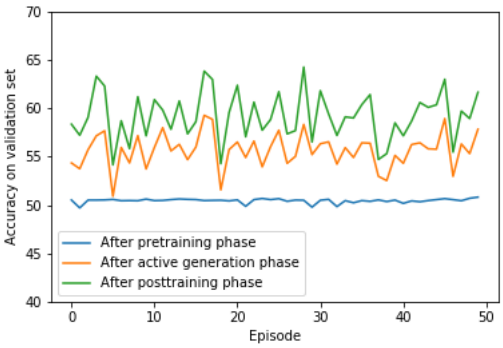
(a) $R_t = Acc_t^{\text{valid}}$



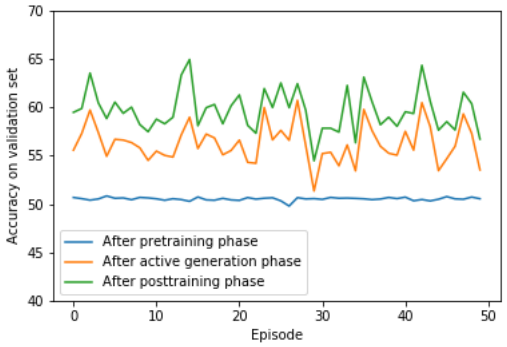
(b) $R_t = -Loss_t^{\text{valid}}$



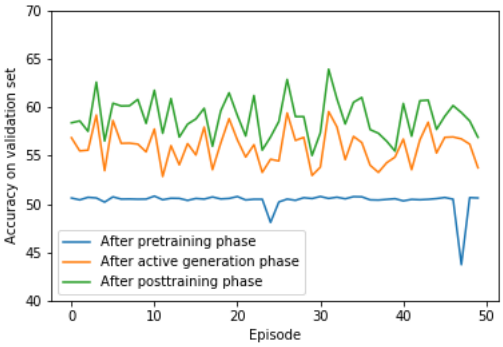
(c) $R_t = Acc_t^{\text{valid}} - Acc_t^{\text{inst}}$



(d) $R_t = Acc_t^{\text{valid}} - 0.2 * Acc_t^{\text{inst}}$



(e) $R_t = Acc_t^{\text{valid}} - Acc_{t-1}^{\text{valid}}$



(f) $R_t = -Acc_t^{\text{inst}}$

Fig. 5.2.1. Evolution of validation accuracy for different choices of rewards.

Notation: R_t is the reward at time step t, Acc_t^{valid} is the accuracy on the validation set at time step t, $Loss_t^{\text{valid}}$ is the loss function's value at time step t and Acc_t^{inst} is the accuracy score for the new instances just after generation, before any updates of the predictor.

instant accuracy on the newly generated instances, namely $-Acc_t^{\text{inst}}$, can help (and we tested using different weights). This accuracy is taken on the new generated instances, right after generation, before any training or parameters updates happens. We also tested the difference of the accuracy validation between two consecutive steps. See Figure 5.2.1 for the evolution of the validation accuracy for each reward. By looking at these plots, we cannot conclude that a specific reward must be used more than others, since the curves (and the numerical results, not shown) are very similar. Following our first thoughts and the simplicity of some rewards, we made most of our experiments using either $R_t = Acc_t^{\text{valid}}$ or $R_t = Acc_t^{\text{valid}} - Acc_t^{\text{inst}}$, but none of those clearly standed out from the other. Because there is no implicit reward in this experiment, and because of the numerous interactions between every part of the whole architecture, defining a relevant reward from which the RL agent can efficiently learn to accomplish his task is far from trivial.

5.3. Observation 3 : Performance depends on budget size

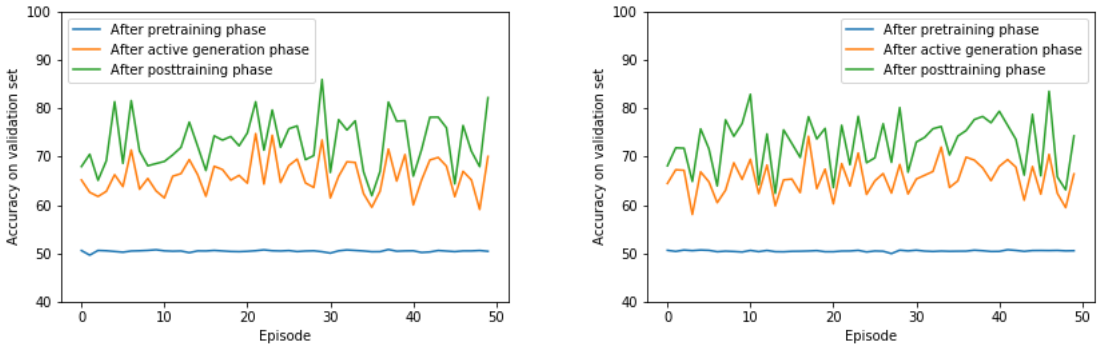
Since all the previous experiments were performed using a total budget of 3,000 hands, we tested the architecture using a budget of 5,000 hands to evaluate the impact of the budget size. We use $R_t = Acc_t^{\text{valid}} - Acc_t^{\text{inst}}$ as reward in this experiment. All the other hyperparameters are the same as those presented in Section 4.1.

In Figure 5.3.1, we present the accuracy on the validation and test datasets using three different methods. The difference between the validation and test accuracy is minim. The active generation by RL method achieves an average accuracy of 72.10% on the test dataset (averaged on the 5 last episodes), while the random generation agent achieves a test accuracy of 70.33%. We can see that there is no significant difference between the RL agent and the random agent considering the high standard deviation. The look of the accuracy evolution curves also look similar, as presented in Figure 5.3.2. Our agent does not learn to provide more relevant instances than the random agent does. Furthermore, with a total budget of 5,000 hands, the standard supervised learning method achieves an average of 80.85% accuracy on the test dataset (averaged on 5 runs). Looking at these results, it seems like, with a budget of 5,000 instances, the general active generation architecture (either with the RL or the random agent) *hurts* the learning process, when compared with standard

supervised learning, where the dataset is completely generated prior to training. However, the RL agent is at least not worse than a random agent.

Method	Valid accuracy		Test Accuracy	
	mean	std	mean	std
Active generation by RL	72.39	6.29	72.10	6.74
Random generation	70.53	7.47	70.33	7.50
Standard supervised learning	80.55	4.01	80.85	3.82

Fig. 5.3.1. Comparative results on the validation and test accuracy using a total budget of 5,000 hands for different methods. Accuracy score are averaged on the 5 last episodes of each experience (or on 5 runs for the standard supervised learning method).



(a) Active generation by RL

(b) Random generation

Fig. 5.3.2. Comparison of the evolution of the validation accuracy per episode, either using the active RL agent or the random agent, with a total budget of 5,000 hands.

We also tried with a total budget of size 1,500. However, with so few datapoints, the predictor sometimes labels every datapoints by the most frequent class, or simply gives random labels between the two most frequent classes. Using the same hyperparameters (except the budget of 1500), the RL agent achieves a test accuracy 51.07% while the random agent achieves 51.45% test accuracy. In comparison, the standard supervised learning achieves 50.56%. Furthermore, regardless of the method, every experiment made with 1500 budget

achieves between 50% and 52% test accuracy. This is all very similar and close to randomly guessing one of the two most frequent classes, so we cannot draw a conclusion about the best method from these results. Looking at the difference in the behavior of our architecture for these budgets, we conclude that it seems very dependent on the allowed budget and that, if the dataset is big enough, the standard supervised learning method yields better accuracy score, in addition of being much simpler (but a different choice of hyperparameters could maybe result in the active architecture being as good as the standard supervised learning method, since the hyperparameters were chosen for 3,000 instance budget). However, recall that, in the original task and in the premises, we supposed that the exact oracle is very expensive, and that is what prevents us from using it, both from directly using it or from directly labelling a big dataset. This is the main reason why we are interested in small budget size and why our methodology has been designed for limited budget and dataset size.

6. Conclusion and future work

In this paper, we propose a novel active learning architecture that formulates the AL problem by metalearning and uses a RL agent aiming to increase the training dataset in a relevant way. This actor-critic agent iteratively generates new instances that are labelled by an oracle before being added to the dataset. For this architecture, there is two requirements for the dataset, namely that there exists (1) a way to generate instances and (2) a way to get the ground-truth labels, e.g., an exact oracle. The predictor needs to understand the relation between instances and labels, as the exact oracle cannot be use at the time of prediction. This architecture differs from typical pool-based active learning techniques, as instances are generated from the actor's output distribution instead of from a pool of unlabelled instances.

The motivating example was a discrete optimization problem with an expensive mathematical solver as oracle. However, we made experiments on the *Poker Hand Dataset*, used as a toy dataset. Simplifications such as the availability of a cheap but exact oracle are used to test the architecture. Since the exact oracle is not expensive, we do not use heuristics as metaproblems in this architecture, only the exact oracle. The results showed that it does not outperform a random generation process or standard supervised learning. We found that there is room for improvement in defining the reward. We believe that reward is crucial and

that we have not found the right one yet. Defining a good state encoder is also a challenge, but it was not deeply studied in this paper.

In many optimization and combinatorial problems, some instances require long computation time, whilst other instances can be solved quickly. However, we only focused on the dataset size on this project, while the cost of labelling (more precisely, the variable labelling cost in between different instances) was not taken into account. In future projects, we could take the computational time of labelling the new instance into account in the reward, as lack of intrinsic computational time associated with the toy experiment made us set this part aside. We also put aside the use of heuristics as metaproblems to remove potential bias, but studying the effect of using different heuristics is also very important when the oracle is expensive. It would be interesting, furthermore, to analyse the advantages of this architecture with classical combinatorial problems, instead of classification problems, since they have an intrinsic labelling cost and are closer to our motivating example. The cost of actively generating data, including the architecture and the learning process of all subnetworks, versus the cost of completely generating the dataset prior to training is also left to investigate, highly linked to the labelling cost.

Conclusion

On a proposé dans ce mémoire une architecture de génération active d'exemples permettant de générer des instances durant l'entrainement. On assume dans notre architecture qu'il est possible de générer directement de nouvelles instances (par exemple, à l'aide d'une distribution) et de les étiquetter à l'aide d'un oracle. Le problème de chargement planifié (LPP), problème de d'optimisation discrète à la base de ce projet, respecte d'ailleurs ces critères. Cette architecture active fait intervernir des notions d'apprentissage actif et de méta-apprentissage, et utilise un agent d'apprentissage par renforcement pour générer des instances pertinentes. Combiner ces modèles et types d'apprentissages pour qu'ils effectuent efficacement leur tâche est complexe, car cela fait intervenir plusieurs objectifs distincts. On utilise d'ailleurs un problème jouet, dont le but est de classifier des mains de poker selon l'une des 10 forces possibles, dans une optique exploratoire pour tester l'architecture dans un environnement plus simple. Cependant, pour cet ensemble de données, on n'a pas trouvé d'avantage significatif à utiliser cette architecture de génération active. Trouver une meilleure définition de récompense pour l'agent reste crucial pour obtenir une meilleure politique de génération d'instances. Ces premières expériences ne sont pas concluantes quant aux résultats, mais il serait intéressant d'investiguer davantage les problèmes identifiés (par exemple, la définition du reward et de l'état) et de tester cette architecture avec d'autres problèmes plus similaires au problème d'intérêt, comme un problème avec des coûts d'étiquettage intrinsèques et variables.

En effet, dans plusieurs problèmes d'optimisation ou problèmes combinatoires (comme dans le problème de chargement planifié), certaines instances demandent beaucoup de temps pour être résolues, alors que d'autres peuvent être résolues rapidement. Ce coût d'étiquettage variable est une partie maitresse de ce projet, mais a été mis de côté, faute de coût intrinsèque dans le problème jouet. Ainsi, en testant sur un problème avec coût d'étiquettage

variable, il faudra ajuster le budget pour prendre en compte le temps de calcul de chaque instance, plutôt qu'uniquement le nombre d'instances à étiquetter. De ce fait, le but de l'agent de RL sera non seulement de générer des instances pertinentes pour l'apprentissage du prédicteur, mais également d'inclure la notion de coût d'étiquettage dans la pertinence des instances. Lorsqu'on considère le coût d'étiquettage variable, on ajoute un compromis relié à la pertinence d'ajouter une nouvelle instance par rapport à son coût d'étiquettage (une instance coûteuse a besoin d'être *vraiment* pertinente pour l'ajouter à l'ensemble d'entrainement). Ainsi, prendre en compte de coût d'étiquettage variable, ainsi que les compromis qui s'y rattachent, fait partie des prochaines étapes pour tester la validité de cette architecture.

Bibliographie

Angluin, D. [1988]. Queries and concept learning, *Mach. Learn.* **2**(4): 319–342.
URL: *https://doi.org/10.1023/A:1022821128753*

Bachman, P., Sordoni, A. and Trischler, A. [2017]. Learning algorithms for active learning, *in* D. Precup and Y. W. Teh (eds), *Proceedings of the 34th International Conference on Machine Learning*, Vol. 70 of *Proceedings of Machine Learning Research*, PMLR, International Convention Centre, Sydney, Australia, pp. 301–310.
URL: *http://proceedings.mlr.press/v70/bachman17a.html*

Baram, Y., El-Yaniv, R. and Luz, K. [2004]. Online choice of active learning algorithms, *J. Mach. Learn. Res.* **5**: 255–291.

Barandelaa, R. and Rangela, E. [2002]. Strategies for learning in class imbalance problems.

Bastin, F. [2010]. *Modèles de Recherche Opérationnelle*. `https://www.iro.umontreal.ca/~bastin/Cours/IFT1575/IFT1575.pdf`.

Bengio, Y., Simard, P. and Frasconi, P. [1994]. Learning long-term dependencies with gradient descent is difficult, *IEEE Transactions on Neural Networks* **5**(2): 157–166.

Biggs, J. [2011]. The role of metalearning in study processes, *British Journal of Educational Psychology* **55**: 185 – 212.

Breiman, L. [1996]. Bagging predictors, *Mach. Learn.* **24**(2): 123–140.
URL: *https://doi.org/10.1023/A:1018054314350*

Cai, W., Zhang, Y. and Zhou, J. [2013]. Maximizing expected model change for active learning in regression, *2013 IEEE 13th International Conference on Data Mining*, pp. 51–60.

Caruana, R. [1995]. Learning many related tasks at the same time with backpropagation, *in* G. Tesauro, D. S. Touretzky and T. K. Leen (eds), *Advances in Neural Information Processing Systems 7*, MIT Press, pp. 657–664.
URL: *http://papers.nips.cc/paper/959-learning-many-related-tasks-at-the-same-time-with-backpropagation.pdf*

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H. and Bengio, Y. [2014]. Learning phrase representations using RNN encoder-decoder for statistical machine translation, *CoRR* **abs/1406.1078**.

URL: *http://arxiv.org/abs/1406.1078*

Dertat, A. [2017]. Applied deep learning - part 1: Artificial neural networks.

URL: *https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6*

Dua, D. and Graff, C. [2017]. UCI machine learning repository.

URL: *http://archive.ics.uci.edu/ml*

Freund, Y., Seung, H. S., Shamir, E. and Tishby, N. [1992]. Information, prediction, and query by committee, *NIPS*.

Goodfellow, I., Bengio, Y. and Courville, A. [2016]. *Deep Learning*, MIT Press. `http://www.deeplearningbook.org`.

Hillier, F. S. and Lieberman, G. J. [2015]. *Introduction to Operations Research, Tenth Edition*, McGraw-Hill, 2 Penn Plaza, New York, NY 10121.

Hochreiter, S. [1991]. Untersuchungen zu dynamischen neuronalen netzen.

Hochreiter, S., Bengio, Y., Frasconi, P. and Schmidhuber, J. [2001]. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

Hochreiter, S. and Schmidhuber, J. [1997]. Long short-term memory, *Neural Computation* **9**(8): 1735–1780.

URL: *https://doi.org/10.1162/neco.1997.9.8.1735*

Hochreiter, S., Younger, A. S. and Conwell, P. R. [2001]. Learning to learn using gradient descent, *Proceedings of the International Conference on Artificial Neural Networks*, ICANN '01, Springer-Verlag, Berlin, Heidelberg, p. 87–94.

Hsu, W.-N. and Lin, H.-T. [2015]. Active learning by learning.

URL: *https://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9636*

Huang, C., Li, Y., Loy, C. C. and Tang, X. [2016]. Learning deep representation for imbalanced classification, *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5375–5384.

Kotsiantis, S., Kanellopoulos, D. and Pintelas, P. [2005]. Handling imbalanced datasets: A review, *GESTS International Transactions on Computer Science and Engineering* **30**: 25–36.

Larsen, E., Lachapelle, S., Bengio, Y., Frejinger, E., Lacoste-Julien, S. and Lodi, A. [2018]. Predicting tactical solutions to operational planning problems under imperfect information, *CoRR* **abs/1807.11876**.

URL: *http://arxiv.org/abs/1807.11876*

Lemke, C., Budka, M. and Gabrys, B. [2015]. Metalearning : a survey of trends and technologies, **44**: 117–130.

Lewis, D. D. and Gale, W. A. [1994]. A sequential algorithm for training text classifiers, *CoRR* **abs/cmp-lg/9407020**.

URL: *http://arxiv.org/abs/cmp-lg/9407020*

Mantovani, S., Morganti, G., Umang, N., Crainic, T. G., Frejinger, E. and Larsen, E. [2018]. The load planning problem for double-stack intermodal trains, *European Journal of Operational Research* **267**(1): 107 – 119.

URL: *http://www.sciencedirect.com/science/article/pii/S0377221717310275*

Nakajima, H. and Iima, H. [2017]. The solution of combinatorial optimization problems based on reinforcement learning, pp. 78–82.

Olah, C. [2015]. Understanding lstm networks.

URL: *https://colah.github.io/posts/2015-08-Understanding-LSTMs/*

Roy, N. and McCallum, A. [2001]. Toward optimal active learning through sampling estimation of error reduction, *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, p. 441–448.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. [1986]. Learning representations by back-propagating errors, *nature* **323**(6088): 533–536.

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D. and Lillicrap, T. [2016a]. Meta-learning with memory-augmented neural networks, *in* M. F. Balcan and K. Q. Weinberger (eds), *Proceedings of The 33rd International Conference on Machine Learning*, Vol. 48 of *Proceedings of Machine Learning Research*, PMLR, New York, New York, USA, pp. 1842–1850.

URL: *http://proceedings.mlr.press/v48/santoro16.html*

Santoro, A., Bartunov, S., Botvinick, M., Wierstra, D. and Lillicrap, T. [2016b]. One-shot learning with memory-augmented neural networks, *Proc. 33nd International Conference on Machine Learning* .

Settles, B. [2009]. Active learning literature survey, *Computer Sciences Technical Report 1648*, University of Wisconsin–Madison.

Settles, B., Craven, M. and Friedland, L. [2008]. Active learning with real annotation costs, *Proceedings of the NIPS workshop on cost-sensitive learning*, Vancouver, CA:, pp. 1–10.

Settles, B., Craven, M. and Ray, S. [2008]. Multiple-instance active learning, *in* J. C. Platt, D. Koller, Y. Singer and S. T. Roweis (eds), *Advances in Neural Information Processing Systems 20*, Curran Associates, Inc., pp. 1289–1296.

URL: *http://papers.nips.cc/paper/3252-multiple-instance-active-learning.pdf*

Seung, H. S., Opper, M. and Sompolinsky, H. [1992]. Query by committee, *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, Association for Computing Machinery, New York, NY, USA, p. 287–294.

URL: *https://doi.org/10.1145/130385.130417*

Shannon, C. E. [1948]. A mathematical theory of communication, *Bell system technical journal* **27**(3): 379–423.

Srivastava, N. [2013]. Improving neural networks with dropout.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. [2014]. Dropout: A simple way to prevent neural networks from overfitting, *Journal of Machine Learning Research* **15**(56): 1929–1958.

URL: *http://jmlr.org/papers/v15/srivastava14a.html*

Sutton, R., Mcallester, D., Singh, S. and Mansour, Y. [2000]. Policy gradient methods for reinforcement learning with function approximation, *Adv. Neural Inf. Process. Syst* **12**.

Sutton, R. S. and Barto, A. G. [2018]. *Reinforcement Learning: An Introduction*, second edn, The MIT Press.

URL: *http://incompleteideas.net/book/the-book-2nd.html*

Thrun, S. and Pratt, L. [1998]. *Learning to Learn: Introduction and Overview*, Kluwer Academic Publishers, USA, p. 3–17.

Vilalta, R. and Drissi, Y. [2002]. A perspective view and survey of meta-learning, *Artificial Intelligence Review* **18**: 77–95.

Werbos, P. J. [1990]. Backpropagation through time: what it does and how to do it, *Proceedings of the IEEE* **78**(10): 1550–1560.

Williams, R. J. [1992]. Simple statistical gradient-following algorithms for connectionist reinforcement learning, *Mach. Learn.* **8**(3–4): 229–256.

URL: *https://doi.org/10.1007/BF00992696*

Woodward, M. and Finn, C. [2017]. Active one-shot learning.

Young, T., Hazarika, D., Poria, S. and Cambria, E. [2017]. Recent trends in deep learning based natural language processing, *CoRR* **abs/1708.02709**.

URL: *http://arxiv.org/abs/1708.02709*

Zhu, X., Lafferty, J. and Ghahramani, Z. [2003]. Combining active learning and semi-supervised learning using gaussian fields and harmonic functions.