

**Université de Montréal**

**Outils de programmation pour systèmes intelligents**

par

**Breandan Considine**

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de

Maître ès sciences (M.Sc.)

en Informatique

juin 2020



# **Université de Montréal**

Faculté des études supérieures et postdoctorales

Ce mémoire intitulé

## **Outils de programmation pour systèmes intelligents**

présenté par

**Breandan Considine**

a été évalué par un jury composé des personnes suivantes :

*Marc Feeley*

---

(président-rapporteur)

*Liam Paull*

---

(directeur de recherche)

*Michalis Famelis*

---

(codirecteur)

*Eugène Syriani*

---

(membre du jury)



## Résumé

---

Les outils de programmation sont des programmes informatiques qui aident les humains à programmer des ordinateurs. Les outils sont de toutes formes et tailles, par exemple les éditeurs, les compilateurs, les débogueurs et les profileurs. Chacun de ces outils facilite une tâche principale dans le flux de travail de programmation qui consomme des ressources cognitives lorsqu'il est effectué manuellement. Dans cette thèse, nous explorons plusieurs outils qui facilitent le processus de construction de systèmes intelligents et qui réduisent l'effort cognitif requis pour concevoir, développer, tester et déployer des systèmes logiciels intelligents. Tout d'abord, nous introduisons un environnement de développement intégré (EDI) pour la programmation d'applications Robot Operating System (ROS), appelé Hatchery ([Chapter 2](#)). Deuxièmement, nous décrivons Kotlin $\nabla$ , un système de langage et de type pour la programmation différentiable, un paradigme émergent dans l'apprentissage automatique ([Chapter 3](#)). Troisièmement, nous proposons un nouvel algorithme pour tester automatiquement les programmes différentiables, en nous inspirant des techniques de tests contradictoires et métamorphiques ([Chapter 4](#)), et démontrons son efficacité empirique dans le cadre de la régression. Quatrièmement, nous explorons une infrastructure de conteneurs basée sur Docker, qui permet un déploiement reproductible des applications ROS sur la plate-forme Duckietown ([Chapter 5](#)). Enfin, nous réfléchissons à l'état actuel des outils de programmation pour ces applications et spéculons à quoi pourrait ressembler la programmation de systèmes intelligents à l'avenir ([Chapter 6](#)).

**Mots-clés:** systèmes intelligents, apprentissage automatique, systèmes de types, systèmes embarqués, systèmes distribués, langages de programmation, programmation fonctionnelle, programmation différentiable, programmation probabiliste, outils de programmation, compilateurs, différenciation automatique, rétropropagation, test automatisé, fuzzing, test métamorphique, test de propriété, modélisation générative, analyse statique, moteur de production, intégration continue, machines virtuelles, ROS, Kotlin, Docker, Duckietown.



## Acknowledgements

---

Je voudrais remercier Gimmey, maman, oncle Mark et papa pour leur amour et leur soutien sans faille. [Hanneli Tavante](#) pour m'avoir enseigné la théorie des types et la beauté de la programmation fonctionnelle. [Xiaoyan Liu](#) pour avoir semé en moi la graine des mathématiques. Oncle Andy pour avoir arrosé la graine pendant de nombreuses années. Tante Shannon, Adam Devoe et Jacquie Kirrane pour m'avoir encouragé à poursuivre des études supérieures. [Arthur Nunes-Harwitt](#) pour m'avoir enseigné la différenciation algorithmique il y a longtemps. [Renee Miller](#) pour avoir suscité mon intérêt pour la science neuronale. [Ian Clarke](#) pour m'avoir montré un nouveau langage intelligent appelé [Kotlin](#). [Hadi Hariri](#) pour me faire plus confiance que je ne le méritais. [Lukas Eder](#) et [Eugene Petrenko](#) pour m'avoir montré la magie des DSL sécurisées et m'avoir donné des conseils sur études supérieures. [Rusi Hristov](#) pour sa patience et son mentorat. [Dmitry Serdyuk](#) et [Kyle Kastner](#) pour m'avoir présenté à Montréal et me souhaitant la bienvenue dans le groupe de lecture de discours. [Isabela Albuquerque](#) et [João Monteiro](#) pour m'avoir montré à quoi ressemblent de bonnes recherches. [Manfred Diaz](#) et [Maxime Chevalier Boisvert](#) pour l'inspiration, les conversations et les commentaires. [Florian Golemo](#) pour ses excellents conseils d'ingénierie et d'architecture. [Ryan Turner](#), [Saikrishna Gottipati](#), [Vincent Mai](#), [Krishna Murthy](#), [Bhairav Mehta](#), [Christos Tsirigotis](#), [Konstantin Solomatov](#) et [Xujie Si](#) pour les conversations intéressantes. [Pascal Lamblin](#), [Olivier Breleux](#) et [Bart van Merriënboer](#) et pour éclairer le chemin entre ML et PL. [Conal Elliott](#) pour m'apprendre l'importance de la simplicité et la sémantique dénotationnelle. [Christian Perone](#) pour m'avoir présenté PyTorch, [Alexander Nozik](#), [Erik Meijer](#), [Kiran Gopinathan](#), [Roman Elizarov](#), [Jacob Miller](#) et [Adam Pocock](#) pour leurs commentaires et retours utiles concernant Kotlin $\nabla$ . [Miltos Allamanis](#) pour m'avoir montré qu'il y a de la place pour SE en ML. [Celine Begin](#) à l'Université de Montréal pour avoir aidé un étranger la veille d'un hiver froid en 2017. [Stefan Monnier](#) pour avoir répondu de manière réfléchie et approfondie à mes courriels errants. [Andrea Censi](#) pour ses conseils et ses encouragements. Enfin et surtout, je tiens à remercier mes brillants conseillers [Liam Paull](#) d'avoir pris une chance sur un échantillon hors distribution, en fournissant de forts gradients et en

me donnant beaucoup plus de crédit que je ne le méritais, et [Michalis Famelis](#) pour m'avoir enseigné la valeur de la logique intuitionniste, des méthodes formelles et de l'autodiscipline. Merci infiniment!

# Contents

---

<b>Résumé .....</b>	v
<b>Acknowledgements .....</b>	vii
<b>List of tables .....</b>	xiii
<b>List of figures .....</b>	xv
<b>Chapter 1. Introduction.....</b>	1
1.1. Conception: Outils de programmation pour la robotique .....	3
1.2. Implémentation: Programmation différenciée par type.....	6
1.3. Vérification: Tester les systèmes intelligents .....	9
1.4. Maintenance: Outils pour une robotique reproductible .....	10
1.5. Contributions .....	14
1.6. Iconographie .....	15
<b>Chapter 2. Outils de programmation pour la robotique.....</b>	17
2.1. Introduction à ROS .....	18
<b>Chapter 3. Type-safe differentiable programming.....</b>	21
3.1. Automatic differentiation .....	22
3.2. Differentiable programming .....	25
3.3. Static and dynamic languages .....	28
3.4. Langages impératifs et fonctionnels .....	29

3.5.	Kotlin .....	30
3.6.	Kotlin $\nabla$ .....	31
3.7.	Usage .....	33
3.8.	Type systems .....	34
3.9.	Shape safety .....	35
3.10.	Testing .....	40
3.11.	Operator overloading .....	41
3.12.	First-class functions .....	42
3.13.	Numeric tower .....	43
3.14.	Algebraic data types .....	44
3.15.	Multiple dispatch .....	45
3.16.	Extension functions .....	46
3.17.	Automatic, symbolic differentiation .....	47
3.18.	Coroutines .....	48
3.19.	Comparison .....	49
3.20.	Travaux futurs .....	51
3.21.	Conclusion .....	52
<b>Chapter 4.</b>	<b>Testing intelligent systems .....</b>	<b>55</b>
4.1.	Background .....	56
4.1.1.	Tests unitaires .....	56
4.1.2.	Tests d'intégration .....	56
4.1.3.	Fuzz testing .....	57
4.1.4.	Property-based testing .....	58

4.1.5.	Metamorphic testing .....	59
4.1.6.	Adversarial testing .....	61
4.1.7.	Generative adversarial testing .....	64
4.2.	Probabilistic adversarial testing .....	66
4.3.	Conclusion .....	73
<b>Chapter 5.</b>	<b>Outils pour la robotique reproductible .....</b>	<b>75</b>
5.1.	Dependency management .....	76
5.2.	Operating systems and virtualization .....	77
5.3.	Containerization .....	78
5.4.	Introduction to Docker .....	80
5.4.1.	Creating an image snapshot .....	82
5.4.2.	Writing an image recipe .....	84
5.4.3.	Layer caching .....	85
5.4.4.	Volume sharing .....	90
5.4.5.	Multi-stage builds .....	90
5.5.	ROS and Docker .....	92
5.6.	Duckiebot development using Docker .....	93
5.6.1.	Flashing a bootable disk .....	94
5.6.2.	Web interface .....	94
5.6.3.	Testing ROS .....	95
5.6.4.	Build and deployment .....	96
5.6.5.	Multi-architecture support .....	96
5.6.6.	Exécution d'un simple serveur de fichiers HTTP .....	97
5.6.7.	Test de la caméra .....	97
5.6.8.	Outils d'interface utilisateur graphique .....	97
5.6.9.	Remote control .....	98

5.6.10. Camera calibration .....	98
5.6.11. Calibrage des roues .....	99
5.6.12. Lane following .....	99
5.7. Retrospective .....	99
5.7.1. Remarques sur la sécurité .....	102
5.8. Travaux futurs .....	103
5.9. Conclusion .....	103
<b>Chapter 6. Conclusion .....</b>	<b>105</b>
6.1. Contributions .....	109
<b>References .....</b>	<b>113</b>
<b>Chapter 7. Type-safe differentiable programming .....</b>	<b>143</b>
7.1. Grammaire .....	143
<b>Chapter 8. Testing intelligent systems .....</b>	<b>145</b>
8.1. Linear regression .....	145
8.1.1. Méthode des différences finies .....	145

## List of tables

---

3.1	Le système de forme de Kotlin $\nabla$ spécifie la forme de sortie pour les expressions tensorielles.....	36
3.2	La forme d'une dérivée du tenseur dépend de la forme de la fonction en cours de différentiation et de la forme de la variable par rapport à laquelle nous nous différencions.....	36
3.3	Comparaison des bibliothèques AD. Bien que nous ne fassions pas de distinction entre AD et SD comme décrit dans § 3.17, nous adoptons ici la nomenclature préférée des auteurs. Nous faisons une distinction entre les bibliothèques de programmation différentiable (§ 3.2) et celles qui construisent simplement des réseaux de neurones. Le symbole  indique un travail en cours.....	50
4.1	DFGs générés par Eq. 4.2.3 avec des tracés en 2D qui les accompagnent.....	70
4.2	Au-dessus: Vérité de base et prédictions de modèles entraînés pour une seule expression. En bas: Une particule unique attaque le modèle en cherchant une erreur plus élevée sur la perte de substitution.....	71



## List of figures

---

2.1	Unique downloads of Hatchery between the time of its release and June 2019. <a href="https://plugins.jetbrains.com/plugin/10290-hatchery">https://plugins.jetbrains.com/plugin/10290-hatchery</a> .....	18
2.2	Une application ROS typique contient un grand graphique de dépendances. ....	19
3.1	<i>Programmation différentiable</i> comprend les réseaux de neurones, mais plus largement, les programmes arbitraires qui utilisent l'optimisation par gradient pour se rapprocher d'une fonction de perte. <i>Programmation probabiliste</i> [Tristan et al., 2014, Carpenter et al., 2017, Gorinova et al., 2019] est une généralisation des modèles graphiques probabilistes qui utilise les méthodes de Monte Carlo (MC) pour approcher une fonction de densité. ....	26
3.2	Deux programmes équivalents, tous deux mettant en uvre la fonction $f(l_1, l_2) = l_1 \cdot l_2$ . ....	30
3.3	Avec l'adaptation de van Merriënboer et al. [2018]. Les modèles Kotlin $\nabla$ sont des structures de données, construites par une DSL intégrée, ardemment optimisées et évaluées paresseusement. ....	32
3.4	Implicit DFG constructed by the original expression, shown above. ....	34
4.1	Nous comparons la dérive numérique entre AD et SD sur une expression gonflée en utilisant une précision fixe et une précision arbitraire (AP). AD et SD présentent toutes deux des erreurs relatives (c'est-à-dire l'une par rapport à l'autre) inférieures de plusieurs ordres de grandeur à leur erreur absolue. Ces résultats sont conformes aux conclusions de Laue [2019]. ....	59
4.2	Pour chaque expression de notre ensemble de données, nous formons un régresseur polynomial à la convergence. ....	69
4.3	By construction, our shrinker detects a greater number of errors per evaluation than one which does not take the gradient into consideration. ....	72

5.1	La virtualisation complète est un processus très gourmand en ressources. La conteneurisation est moins coûteuse, car elle partage un noyau avec le système d'exploitation hôte. L'émulation nous permet d'émuler le matériel comme un logiciel. Chacune de ces méthodes peut être utilisée en conjonction avec n'importe quelle autre. ....	78
5.2	Containers live in user space. Par défaut, ils sont mis en bac à sable à partir de l'OS hôte et des conteneurs frères et surs, mais contrairement aux VM, ils partagent un noyau commun entre eux et avec l'OS hôte. Tous les appels système sont passés par le noyau hôte. ....	79
5.3	Infrastructure des conteneurs. <b>Gauche</b> : La pile ROS cible deux architectures primaires, x86 et ARM. Pour simplifier le processus de construction, nous construisons des artefacts ARM sur x86 en utilisant QEMU [Bellard, 2005]. <b>Right</b> : Pile d'apprentissage de renforcement. Les artefacts de construction sont formés sur un GPU, et transférés sur le CPU pour évaluation. Les modèles d'apprentissage profond peuvent également être exécutés sur un dispositif ARM à l'aide d'un accélérateur. ....	92
5.4	Interface de navigation pour les Duckiebots individuels. Elle est fournie par Portainer, un tableau de bord web RESTful, qui enveloppe le CLI du Docker et offre un support pour la gestion des conteneurs, la configuration, la mise en réseau et l'émulation des terminaux (montré ci-dessus). <a href="http://DUCKIEBOT_NAME:9000/#/container/container">http://DUCKIEBOT_NAME:9000/#/container/container</a> ↗ ....	95
5.5	Premier prototype de la hiérarchie d'images de Docker. L'enchaînement de constructions automatiques non versionnées sans essais unitaires disciplinés crée un effet domino potentiel qui permet aux changements de rupture de se propager en aval, entraînant une cascade de défaillances silencieuses. ....	101
5.6	The AI Driving Olympics, un cas d'utilisation primaire pour le système décrit ci-dessus. ....	102
6.1	Complexité de la détection de divers types d'erreurs de programmation. ....	108
6.2	Many interesting applications lie at the intersection of these three fields. ....	110

# Chapter 1

---

## Introduction

“Il y a une course entre la complexité croissante des systèmes que nous construisons et notre capacité à développer des outils intellectuels pour comprendre leur complexité. Si la course est gagnée par nos outils, les systèmes deviendront finalement plus faciles à utiliser et plus fiables. Sinon, ils continueront à être plus difficiles à utiliser et moins fiables pour tous, sauf pour un ensemble relativement restreint de tâches communes. Étant donné la difficulté de la réflexion, si ces outils intellectuels doivent réussir, ils devront remplacer la pensée par le calcul.”

---

—Leslie Lamport [2002], *Une discussion avec Leslie Lamport*

La complexité du calcul est une telle préoccupation en informatique qu'une grande partie du domaine est consacrée à sa compréhension à travers la lentille de l'analyse des fonctions et de la théorie de l'information. Dans le domaine du génie logiciel, les chercheurs s'intéressent principalement à la complexité de la construction des logiciels, c'est-à-dire la manifestation numérique des algorithmes sur le matériel physique. Un type de complexité logicielle est l'effort cognitif requis pour comprendre un programme. Bien que les logiciels d'aujourd'hui deviennent rapidement plus intelligents, ils ne montrent que peu de signes de devenir plus intelligibles. De meilleurs outils sont nécessaires pour gérer la complexité des systèmes logiciels de construction.

*L'objectif de cette thèse est de développer des méthodes qui réduisent l'effort cognitif nécessaire pour construire des systèmes intelligents, en utilisant des outils de développement, des abstractions de langage de programmation, des tests automatisés et la technologie de virtualisation.*

D'une manière générale, les systèmes intelligents diffèrent des systèmes logiciels ordinaires en ce qu'ils permettent aux machines de détecter des modèles, d'exécuter des tâches et de

résoudre des problèmes qu’elles ne sont pas explicitement programmées pour résoudre et que les experts humains étaient auparavant incapables de résoudre en codant en dur des règles explicites. Généralement, ces systèmes sont capables de:

- (1) apprendre des règles généralisables en traitant de grandes quantités de données
- (2) régler un grand nombre de paramètres libres (des milliers à des milliards)
- (3) surpasser les humains bien formés dans des tâches spécifiques à un domaine

Si l’idée de systèmes intelligents existe depuis des décennies, trois évolutions essentielles ont fait le succès des systèmes intelligents modernes. Premièrement, la puissance de traitement des ordinateurs est devenue plus rapide, moins chère et beaucoup plus facilement disponible. De même, la numérisation de nouveaux ensembles de données a rendu disponibles de vastes quantités d’informations et les coûts de stockage des données ont chuté de façon spectaculaire. (Une clé USB de \$5 a aujourd’hui une capacité de stockage 200 fois supérieure à celle d’un disque dur IBM de 5 MB de 2,000 livres, loué pour \$3,000 par mois en 1956). Plus important encore, a été le développement d’algorithmes d’apprentissage plus efficaces.

Ces dernières années, l’informatique et le génie logiciel ont fait des progrès considérables dans la construction et le déploiement de systèmes intelligents. Presque tous les ordinateurs mobiles du monde sont capables de détecter des objets dans des images, d’effectuer des traductions de la parole au texte et des traductions de langue. Ces percées sont le résultat direct des progrès fondamentaux réalisés dans le domaine des réseaux neuronaux et de l’apprentissage de la représentation. L’adoption de pratiques collaboratives à code source ouvert, dont la communauté du génie logiciel a été la pionnière, a également été la clé du succès des systèmes intelligents modernes. Les ingénieurs en logiciel ont développé des bibliothèques de différenciation automatique comme Theano [Bergstra et al., 2010], Torch [Collobert et al., 2002] et Caffe [Jia et al., 2014], et ont construit de nombreux simulateurs populaires pour l’apprentissage du renforcement. La facilité d’utilisation et la disponibilité de ces outils ont été cruciales pour démocratiser les techniques d’apprentissage approfondi.

Les systèmes intelligents sont largement déployés dans des environnements virtuels comme la science des données et les services en ligne. Mais même avec l’énorme succès des algorithmes d’apprentissage automatique dans des domaines entièrement observables comme la

reconnaissance d’images et le traitement de la parole, les systèmes intelligents n’ont pas encore été largement adoptés en robotique (au moment de la rédaction de cette thèse). Ce dilemme peut être partiellement attribué à divers problèmes théoriques tels que l’adaptation au domaine et l’apprentissage par transfert. Pourtant, grâce aux capacités éprouvées des algorithmes d’apprentissage modernes, à l’augmentation exponentielle de la puissance de traitement et aux efforts déployés depuis des décennies pour construire des agents intelligents physiquement incorporés, nous devrions avoir plus de progrès à montrer. Pourquoi cet objectif a-t-il échappé aux chercheurs pendant si longtemps? L’une des raisons, nous le supposons, est le manque d’outils de programmation et d’abstractions pour concevoir, développer, déployer et évaluer les systèmes intelligents. En pratique, ces activités consomment une grande quantité d’efforts cognitifs sans le bon ensemble d’outils et d’abstractions.

Dans le domaine du génie logiciel traditionnel, le modèle Waterfall est un modèle classique de développement de logiciels comprenant différentes étapes [Royce, 1987]. Nous proposons des contributions à quatre étapes: Premièrement, nous faisons la démonstration d’un environnement de développement intégré pour les logiciels de robotique *conception* (Chapter 2). Ensuite, nous montrons un langage spécifique à un domaine et sans danger pour les programmes différenciables de *implémentation*, un paradigme émergent dans l’apprentissage profond (Chapter 3). Pour *vérifier* cette application, nous utilisons un ensemble de techniques empruntées aux tests basés sur les propriétés [Fink and Bishop, 1997] et à l’apprentissage contradictoire [Lowd and Meek, 2005] (Chapter 4). Les conteneurs Docker [Merkel, 2014] sont utilisés pour automatiser le *maintenance* d’applications robotiques reproductibles sur des plateformes matérielles hétérogènes (Chapter 5). Enfin, nous présentons quelques remarques de conclusion et les enseignements tirés de la construction de ces outils (Chapter 6).

## 1.1. Conception: Outils de programmation pour la robotique

Les systèmes logiciels d’aujourd’hui sont des entités profondément complexes. L’époque où un programmeur solitaire, même très compétent, pouvait assurer seul la maintenance d’un grand système logiciel est révolue. Pour mettre efficacement à l’échelle les systèmes logiciels modernes, les programmeurs doivent mettre en commun leur capacité mentale pour former un graphe de connaissances. Les projets logiciels qui reposent sur un petit nombre de responsables ont tendance à disparaître en raison de ce que l’on appelle le "facteur bus [Cosentino

et al., 2015] – de grandes parties du graphe de connaissances sont enfermées dans la tête de quelqu'un. Les projets logiciels réussis apprennent à distribuer ce graphe et à établir des connexions avec le monde extérieur. Le graphe de connaissances qui s'accumule autour d'un projet logiciel contient des faits, mais aussi des flux de travail pour la programmation, le débogage et la livraison – des chemins communs à travers le labyrinthe du développement logiciel [Naur, 1985]. Les composants de ce graphique peuvent être engagés à l'écriture, mais la documentation prend du temps et devient périmée avec le temps. Ce qu'il faut, c'est un système qui offre les avantages de la documentation sans les inconvénients de la maintenance.

Le développement de systèmes logiciels comporte un deuxième élément, le graphe social. Le graphe social d'un projet logiciel réussi contient les concepteurs de produits, les gestionnaires et les ingénieurs logiciels qui travaillent de concert pour construire un logiciel bien conçu, cohésif et très performant. Cela implique parfois de réviser la spécification pour tenir compte des défis techniques, ou de réécrire le code source pour supprimer la dette technique. La conception de logiciels est un processus d'optimisation à objectifs multiples et nécessite des collaborateurs ayant un large éventail de compétences et un ensemble d'objectifs communs. Pour produire un logiciel qui se rapproche des critères de ses intervenants, les développeurs sont invités à fournir des prototypes rapides et à intégrer en permanence les commentaires des utilisateurs. Pourtant, les systèmes logiciels d'aujourd'hui sont plus grands et plus compliqués que jamais. Il est donc essentiel de trouver des moyens de collaborer plus efficacement pour construire des systèmes plus intelligents.

Examinons tout d'abord le processus mécanique d'écriture de logiciels à l'aide d'un clavier.

Les environnements de développement intégrés (EDI) peuvent aider les développeurs à créer des applications logicielles complexes en automatisant certaines tâches de programmation répétitives. Par exemple, les EDI effectuent des analyses et des inspections statiques pour détecter rapidement les bogues. Ils permettent de compléter, de remanier et de naviguer dans le code source, et ils automatisent le processus de construction, d'exécution et de débogage des programmes. Bien que ces tâches puissent sembler insignifiantes, leur automatisation promet d'accroître la productivité des développeurs en leur permettant de fournir un retour d'information plus tôt, de détecter les erreurs d'écriture et de libérer des ressources mentales qui pourront être utilisées ailleurs. Plutôt que d'être obligés de se concentrer sur la

structure et l'organisation du texte, si les développeurs sont capables de manipuler le code à un niveau sémantique, ils seront beaucoup plus heureux et plus productifs. De plus, en automatisant les tâches mécaniques dans le développement de logiciels, ces outils libèrent l'attention vers l'activité fondamentale de l'écriture et de la compréhension des programmes.

Mais que font réellement les EDI? Ils guident les développeurs à travers le graphe de connaissances d'un projet logiciel. Pensez à ce qu'un nouveau développeur doit apprendre pour se mettre à niveau: en plus d'apprendre le langage, les développeurs doivent apprendre à utiliser des bibliothèques et des cadres (sans doute des langages à part entière). Ils doivent se familiariser avec les outils en ligne de commande pour le développement de logiciels, des outils de construction au contrôle de version et à l'intégration continue. Ils doivent se familiariser avec l'écosystème du logiciel, les styles de programmation, les conventions et les flux de développement. Et ils doivent apprendre à collaborer au sein d'une équipe distribuée de développeurs. En automatisant les tâches courantes dans un environnement de programmation interactif et en rendant explicite la connectivité des graphes grâce au balisage des documents [Goldfarb, 1981] et à l'édition de projets [Voelter et al., 2014], un EDI bien conçu est un outil de traversée de graphes. Il n'est pas surprenant que les EDI soient en fait des bases de données de graphes.

Sous certains aspects, le développement de systèmes intelligents n'est pas différent du génie logiciel classique. Les mêmes principes et meilleures pratiques qui guident le génie logiciel sont également applicables aux systèmes intelligents. Et les mêmes activités, de la conception à la maintenance, continueront à jouer un rôle important dans la construction de systèmes intelligents. Mais à d'autres égards, les outils de programmation génériques utilisés pour développer des logiciels traditionnels nécessiteront des adaptations spécifiques à chaque domaine pour que les systèmes d'apprentissage deviennent des citoyens de premier ordre dans la prochaine génération de logiciels intelligents, notamment dans le cas du développement de la robotique.

Dans ce but, nous avons développé un EDI pour le [Système d'exploitation des robots](#) (ROS) appelé [Hatchery](#). Il prend en charge un certain nombre de flux de travail communs pour le développement des ROS, tels que la création de nuds ROS, l'intégration du simulateur Gazebo, la prise en charge du débogage à distance, l'analyse statique, l'autocomplétion et le refactoring. Dans [Chapter 2](#) nous discutons de la mise en uvre de ces fonctionnalités et

de certains des défis liés à la prise en charge du langage, aux outils de programmation et à l'intégration avec le middleware ROS. Nous soutenons que ces outils réduisent la complexité cognitive de la construction d'applications ROS en adoptant des conventions de codage explicites, en annotant le texte non structuré et en automatisant les tâches de développement répétitives.

## 1.2. Implémentation: Programmation différenciée par type

Aux premiers temps de l'apprentissage machine, on croyait généralement que l'intelligence à l'échelle humaine émergerait d'une logique de premier ordre suffisamment descriptive. En accumulant une base de données de faits et de leurs relations, les chercheurs pensaient pouvoir utiliser un raisonnement symbolique pour contourner l'apprentissage. Cette approche fondée sur des règles a dominé une grande partie des premières recherches sur l'intelligence artificielle et des efforts considérables ont été consacrés à la création d'ontologies propres à chaque domaine pour saisir les connaissances humaines. Malgré les meilleurs efforts des roboticiens, des ingénieurs en traitement du signal et des chercheurs en langage naturel, les *systèmes experts* n'ont pas pu s'adapter aux applications du monde réel, ce qui a provoqué une grande désillusion dans la recherche sur l'intelligence artificielle pendant plusieurs décennies. Alors que les informaticiens ont sous-estimé la difficulté de l'*apprentissage*, les systèmes experts ont excellé dans des domaines où les systèmes actuels d'apprentissage machine ont du mal à s'adapter, comme le raisonnement classique et l'interprétabilité, et il y a de plus en plus de preuves qui suggèrent que beaucoup de ces idées étaient simplement en avance sur leur temps. Dans notre travail, nous nous inspirons de certains travaux antérieurs sur le raisonnement symbolique [Dwyer et al., 1948, Glushkov et al., 1971], les systèmes de types [Lof et al., 1973, Jay and Sekanina, 1997] et la programmation fonctionnelle [McCarthy, 1960, Abelson and Sussman, 1996].

Ce qui a finalement été montré à l'échelle, c'est l'idée de l'apprentissage connexionniste. En imbriquant des approximatrices de fonctions aléatoires, appelés perceptrons, et en mettant à jour les paramètres libres à l'aide de la rétropropagation [Werbos et al., 1990, Rumelhart et al., 1988], le système résultant est capable d'apprendre une quantité surprenante de comportements intelligents. L'approche, appelée réseaux neuronaux artificiels (ANN), remonte au milieu du 20ème siècle [Ivakhenko and Lapa, 1965, Rosenblatt, 1958], mais n'a été

pleinement réalisée *in silico* qu’après la généralisation de l’informatique bon marché et des grands ensembles de données [LeCun et al., 2015]. En théorie, une seule couche d’imbrication est capable d’approcher toute fonction différentiable continue [Hornik et al., 1989], mais en pratique, l’apprentissage nécessite de composer de nombreux approximatrices de ce type de manière profondément imbriquée, d’où le terme, *deep neural networks* (DNNs). L’importance de la profondeur a été soupçonnée pendant de nombreuses années, mais l’algorithme de rétropropagation original avait des difficultés à former les DNN en raison du problème du gradient de disparition [Bengio et al., 1994]. La résolution de ce problème a nécessité un certain nombre d’adaptations et de nombreuses années pour être entièrement débogué. Ce n’est que vers 2013 que l’apprentissage approfondi est devenu compétitif par rapport aux experts humains dans des domaines spécifiques.

Bien qu’il ait fallu une recherche fondamentale en matière d’apprentissage approfondi pour réaliser le plan connexionniste, le succès de l’apprentissage approfondi moderne peut être en partie attribué aux outils logiciels de calcul des dérivés mathématiques, une étape clé de l’algorithme de rétropropagation. Bien qu’il n’ait pas encore été établi si et comment les dérivés peuvent être calculés dans les circuits biologiques, les dérivés sont essentiels pour la formation des ANN. Pendant de nombreuses années, la forme symbolique de ces dérivés a été dérivée analytiquement lors du prototypage d’une nouvelle architecture de réseau neuronal, un processus fastidieux et sujet aux erreurs. Il existe un algorithme bien connu dans la communauté du calcul scientifique qui remonte aux années 1970, appelé *differentiation automatique* (AD) [Linnainmaa, 1976, Griewank et al., 1989], qui est capable de calculer des dérivés pour des fonctions différentiables arbitraires. Mais étonnamment, ce n’est que beaucoup plus tard, après le développement de Theano [Bergstra et al., 2010], que l’AD a été largement adopté par la communauté de l’apprentissage machine. Cette bibliothèque a considérablement accéléré le rythme de la recherche sur l’apprentissage profond et a stimulé le développement d’autres cadres AD comme TensorFlow [Abadi et al., 2016] et PyTorch [Paszke et al., 2019].

Les systèmes intelligents conçus doivent réfléchir attentivement aux langages et aux abstractions. Si les développeurs doivent mettre en uvre la rétropropagation à la main, ils auront peu de temps pour réfléchir aux caractéristiques de haut niveau de ces systèmes. De même, si les abstractions de programmation sont trop spécifiques, de petites variations

nécessiteront une réimplémentation coûteuse. Cela n'est pas différent du génie logiciel traditionnel - en tant qu'ingénieurs, nous devons choisir les bonnes abstractions pour la tâche à accomplir. Trop bas niveau et la conception se perd dans les détails – trop abstrait et les détails se perdent complètement. Avec un apprentissage approfondi, la nécessité de choisir de bonnes abstractions est encore plus importante, car la relation entre le code source et le comportement est déjà difficile à déboguer, en raison de la complexité des réseaux de neurones et de la programmation des tableaux. Une composante de cette complexité se trouve dans le système de types.

La plupart des cadres AD existants pour l'apprentissage machine sont écrits dans des langages à typage dynamique comme Python, Lua et JavaScript, avec quelques premières implémentations, notamment des projets comme [Theano](#) [Bergstra et al., 2010], [Torch](#) [Collobert et al., 2002] et [Caffe](#) [Jia et al., 2014]. Des idées similaires sont apparues dans des langages fonctionnels à caractères statiques, tels que Java ([JAutoDiff](#) [Nureki, 2012], [DL4J](#) [Team, 2016a], [Hipparchus](#) [Andrea and Maisonneuve, 2016]), Scala ([Nexus](#) [Chen, 2017]), F# ([Diff-Sharp](#) [Baydin et al., 2015b]), [Swift](#) [Lattner and Wei, 2018], Haskell ([TensorSafe](#) [Piñeyro et al., 2019]) et al, mais peu d'entre eux sont capables de vérifier la forme des tableaux multidimensionnels dans leur système de types, et ceux qui le font sont implémentés dans des langages expérimentaux avec des types dépendants. Dans notre travail, nous démontrons la viabilité de la vérification de type dans un langage largement utilisé. Cela garantit que les programmes sur les matrices, s'ils se compilent, ont la forme correcte et peuvent être évalués numériquement au moment de l'exécution.

[Kotlin \$\nabla\$](#)  est un langage dédié interne (eDSL) pour la programmation différentiable dans un langage appelé [Kotlin](#), un langage de programmation à caractères statiques prenant en charge la programmation asynchrone et la compilation multi-plateforme. Dans [Kotlin \$\nabla\$](#)  ([Chapter 3](#)), nous décrivons une implémentation algébrique de la différenciation automatique avec des opérations de tenseur de type sécurisé. Notre approche diffère de la plupart des cadres AD existants dans la mesure où [Kotlin \$\nabla\$](#)  est la première bibliothèque de type sécurisé AD à entièrement compatible avec le système de type Java, ne nécessitant aucune métaprogrammation, réflexion ou intervention du compilateur pour être utilisée.

### 1.3. Vérification: Tester les systèmes intelligents

La plupart des phénomènes naturels, en particulier ceux liés à la vision, à la planification et à la locomotion, sont des créatures de grande dimension. Richard Bellman a célébrement appelé ce problème la "fléau de la dimensionnalité". Notre univers physique est peuplé de problèmes simples à poser, mais apparemment impossibles à résoudre en son sein. Claude Shannon, un contemporain de Bellman, a calculé que le nombre de parties d'échecs uniques dépassait  $10^{120}$ , soit plus que le nombre d'atomes dans l'univers d'environ quarante ordres de grandeur [Shannon, 1950]. À l'époque, on pensait que de tels problèmes seraient insurmontables sans percées fondamentales dans les algorithmes et les machines informatiques. En effet, si Bellman ou Shannon n'ont pas vécu pour voir le jour, il n'a fallu qu'un demi-siècle de progrès en informatique [Campbell et al., 2002] avant que des solutions à des problèmes du même ordre de complexité, découverts pour la première fois lors de l'explosion cambrienne il y a 541 millions d'années, soient mises en uvre avec une marge concurrentielle sur les ordinateurs modernes [Pratt, 2015].

Alors que l'informatique a fait d'énormes progrès dans la résolution des cas les plus courants, le fléau de la dimensionnalité de Bellman hante toujours la longue queue de l'apprentissage machine, en particulier des distributions très dispersées. Comme la dimensionnalité de nombreuses fonctions du monde réel que nous voudrions approcher est d'une ampleur insurmontable, il est difficile de vérifier le comportement d'une solution candidate dans tous les régimes, en particulier dans des contextes où les échecs sont rares mais catastrophiques. Selon certaines études, les conducteurs humains comptent en moyenne 1,09 décès par cent millions de miles [Kalra and Paddock, 2016]. Un nouveau logiciel pour un véhicule autonome devrait accumuler 8,8 milliards de miles de conduite afin d'approcher le taux de mortalité d'un conducteur humain à 20% près avec un intervalle de confiance de 95%. Le déploiement d'un tel système dans le monde réel serait problématique sur le plan logistique, sans parler de l'éthique.

D'un point de vue réaliste, les systèmes intelligents ont besoin de meilleurs moyens de mettre en pratique leurs compétences et de sonder l'efficacité d'une solution candidate dans le cadre d'un budget de calcul limité, sans nuire aux humains dans le processus. L'objectif de ces tests est de mettre en évidence les erreurs, mais aussi, en fin de compte, de fournir un retour d'information au système. Dans le domaine du génie logiciel, le véritable système testé

est l'écosystème des humains et des machines qui se fournissent mutuellement des moyens de subsistance. Le succès de cet arrangement dépend d'un mécanisme de test externe pour appliquer une barre de rigueur minimale, généralement une forme de test matériel ou humain en boucle. Si le mécanisme de test n'est pas opposé d'une manière ou d'une autre au système testé, un système intelligent peut se tromper, ce qui n'est ni dans l'intérêt du système ni dans celui de ses utilisateurs.

Plus largement, nous pouvons considérer la vérification de type ([Chapter 3](#)) et les tests automatisés ([Chapter 4](#)) comme faisant partie d'un ensemble d'outils plus large pour la vérification et la validation des logiciels. Plus les anomalies sont détectées rapidement, plus elles sont faciles à corriger et plus les systèmes autonomes peuvent devenir sûrs. Les précédentes approches de tests automatisés ont nécessité une expertise de domaine considérable pour être déployées avec succès, mais les progrès récents en matière de tests métamorphiques [[Chen et al., 1998](#)] et d'apprentissage autosurveillé [[Lieb et al., 2005](#)] ont montré des applications dans des domaines de plus en plus généraux [[Zhang et al., 2020](#)]. Dans ce but, nous proposons dans [Chapter 4](#) un nouvel algorithme inspiré des tests basés sur les propriétés et de l'apprentissage contradictoire qui améliore empiriquement l'efficacité des données et nécessite beaucoup moins d'expertise de domaine pour être mis en uvre que les méthodes basées sur les propriétés natives.

## 1.4. Maintenance: Outils pour une robotique reproductible

L'un des défis de la construction de systèmes intelligents et de la programmation en général, est le problème de la reproductibilité. La reproductibilité des logiciels comporte plusieurs aspects difficiles, notamment la compatibilité matérielle, les systèmes d'exploitation, les systèmes de fichiers, les systèmes de construction et le déterminisme d'exécution. Alors que l'écriture de programmes et leur introduction directe dans un ordinateur a pu être une pratique courante par le passé, le code source actuel est bien trop éloigné de sa réalisation mécanique pour être exécuté de manière significative de manière isolée. Les programmes manuscrits d'aujourd'hui sont comme les schémas d'un feu de circulation - construits à l'intérieur d'une usine, et qui nécessitent une infrastructure, des voitures et des règles de circulation digne d'une ville pour remplir leur fonction. Comme les feux de signalisation, le code source n'existe pas dans le vide – construit par des compilateurs, interprété par des

machines virtuelles, exécuté à l'intérieur d'un système d'exploitation, et qui suit un protocole de communication spécifique – les programmes sont essentiellement des abstractions dénuées de sens en dehors de ce contexte.

Comme le veut tout bon schéma, une grande partie des informations nécessaires à la construction d'un programme est divisée en couches d'abstraction. La plupart des instructions de bas niveau exécutées par un ordinateur pendant l'exécution d'un programme n'ont pas été écrites ni destinées à être lues par le programmeur et ont depuis été automatisées et oubliées. Dans un langage de programmation moderne comme Java, C# ou Python, l'ensemble des informations nécessaires à l'exécution d'un programme simple se chiffre souvent en billions de bits. Une partie de ces données concerne le logiciel de construction et d'exécution des programmes, y compris le système de construction, les dépendances logicielles et les outils de développement. Une partie des données concerne le système d'exploitation, les microprogrammes, les pilotes et les logiciels intégrés. Pour la plupart des programmes, tels que ceux que l'on trouve dans un dépôt GitHub typique, une petite partie des informations correspond au code source lui-même.

L'apprentissage machine appliqué partage bon nombre des mêmes défis pratiques que le développement traditionnel de logiciels, avec le code source, la gestion des versions et des dépendances. Le processus actuel de formation d'un modèle d'apprentissage approfondi peut être considéré comme une étape de compilation particulièrement longue, mais il en diffère sensiblement dans la mesure où le code source est un langage de haut niveau qui ne décrit pas directement le calcul effectué, mais est une sorte de méta-programme. Le premier méta-programme décrit la connectivité d'un grand graphe dirigé (c'est-à-dire un graphe de calcul ou un modèle graphique probabiliste), paramétré par des poids et des biais. Le réglage de ces paramètres est un autre méta-programme, qui décrit la séquence d'opérations nécessaires pour se rapprocher d'un programme auquel nous n'avons pas accès, à l'exception de quelques exemples d'entrées-sorties. Les techniques émergentes en matière de méta-apprentissage et d'optimisation des hyper-paramètres (par exemple recherche d'architecture différentiable [Liu et al., 2018]) ajoutent encore d'autres couches de méta-programmation à cette pile, en effectuant des recherches dans l'espace des graphes dirigés eux-mêmes.

Les fabricants de matériel informatique ont mis au point divers accélérateurs spécialisés pour former et exécuter rapidement ces programmes. Mais contrairement à la plupart

des programmes, l'apprentissage profond est un modèle de calcul beaucoup plus simple – tant qu'un ordinateur peut additionner et multiplier, il a la capacité de faire fonctionner un réseau neuronal profond. Cependant, en raison de la variété des plates-formes matérielles qui existent et des changements de logiciels qui y sont associés, la reproduction des modèles d'apprentissage profond peut être laborieusement difficile sur du nouveau matériel, même avec le même code source et les mêmes dépendances. De nombreux formats de graphes, ou *représentations intermédiaires* (IR) dans le langage des compilateurs, promettent la portabilité du matériel mais si les développeurs ne sont pas prudents, leurs modèles peuvent ne pas converger pendant la formation ou peuvent produire des résultats différents sur des matériels différents. Pour compliquer le problème, les IR sont produits par des vendeurs concurrents, qui vendent des puces concurrentes avec des normes incompatibles (par exemple MLIR [Lattnier et al., 2020], ONNX [Bai et al., 2019], nGraph [Cyphers et al., 2018], Glow [Rotem et al., 2018], TVM [Chen et al., 2018] et al.) Si certains ont essayé d'exploiter les compilateurs existants tels que GHC [Elliott, 2018] ou DLVM/LLVM [Wei et al., 2017], il y a peu de signes d'une interopérabilité plus large au moment de la rédaction de cette thèse.

En fin de compte, les chercheurs doivent reproduire les travaux d'autres chercheurs, mais l'effort mental de réimplanter des abstractions de base peut entraver le progrès scientifique. Les outils qui facilitent la reproductibilité et le développement progressif des logiciels sont essentiels. Heureusement, c'est le même problème qui préoccupe l'industrie du logiciel depuis de nombreuses années et qui a produit de nombreux logiciels gestion de versions (VCS). Mais le VCS seul est insuffisant, car ces outils sont principalement destinés à stocker du texte. Les représentations basées sur le texte sont temporairement stables, mais lorsque les dépendances sont mises à jour et reconstruites, des détails importants sur l'environnement de développement original peuvent être égarés. Pour reproduire un programme dans son intégralité, il faut un instantané de toutes les informations numériques disponibles pendant l'exécution, et idéalement, l'ordinateur physique lui-même. En l'absence d'un instantané complet, il est hautement souhaitable de disposer d'un ensemble minimal de dépendances et d'une réplique quasi physique. Toute variabilité dans le graphique des dépendances physiques ou numériques peut être une source de divergences qui nécessite du temps et de l'énergie pour les isoler ultérieurement.

Afin d’atténuer les effets de la variabilité des logiciels et d’aider au développement de systèmes intelligents sur des plateformes hétérogènes, nous utilisons un outil de développement appelé [Docker](#), qui fait partie d’un ensemble d’outils d’automatisation de la construction et des opérations de développement que nous appellerons *container infrastructure*. Docker permet aux développeurs de figer une application logicielle et son environnement hôte, ce qui permet aux développeurs (par exemple en utilisant un environnement différent) de reproduire rapidement ces applications. Docker est en soi une solution technique, mais il comprend également un ensemble de meilleures pratiques et de lignes directrices de nature plus méthodologique. Bien que Docker ne traite pas de l’incompatibilité des normes des fournisseurs et des pilotes de matériel, il rend ces variables explicites et réduit la difficulté de reproduire les artefacts logiciels.

La reproductibilité des logiciels des systèmes intelligents comporte un deuxième élément qui intègre la notion de temps: les simulateurs. Les simulateurs sont utilisés dans presque toutes les disciplines d’ingénierie pour imiter un processus physique dont la réalisation peut être coûteuse, dangereuse ou peu pratique. Par exemple, les simulateurs sont souvent utilisés pour modéliser la dynamique d’une autre architecture de jeu d’instructions [Bellard, 2005], la dynamique des transitoires électromagnétiques [Tavante et al., 2018], la dynamique du mouvement orbital [Bellman et al., 1965], la dynamique des systèmes de transport humain [Ruch et al., 2018], ou la dynamique de la conduite [Chevalier-Boisvert et al., 2018]. Les ordinateurs d’aujourd’hui sont capables d’effectuer des simulations de plus en plus précises, mais la plupart des praticiens s’accordent à dire que la simulation seule ne suffira jamais à saisir la totalité de la distribution des données du monde réel. Dans cette optique, la simulation peut être un outil utile pour détecter les erreurs, mais elle ne peut pas reproduire pleinement toutes les subtilités du monde réel et ne doit pas se substituer aux tests sur des données réelles. D’autres ont suggéré une voie intermédiaire [Bousmalis et al., 2018], où l’utilisation judicieuse de la formation sur simulateur, parallèlement à l’adaptation au domaine, constitue un environnement suffisamment rigoureux pour évaluer les systèmes intelligents. Quelle que soit l’opinion qui prévaut, notre objectif est de fournir un retour d’information rapide aux développeurs et de rendre l’ensemble du processus, des essais au déploiement, aussi reproduitible que possible.

## 1.5. Contributions

Kernighan and Plauger [1976] introduit d’abord le terme *outils logiciels* dans le contexte des utilitaires en ligne de commande Unix, à peu près dans le même esprit que les outils proposés dans cette thèse. Thrun [2000], Erez et al. [2015] développer des outils basés sur le langage et la simulation pour le développement de la robotique dans le même esprit. D’une manière générale, nous considérons tout logiciel qui aide les utilisateurs engagés dans l’activité d’écriture de programmes informatiques, comme un *outil de programmation*.

Dans cette thèse, nous faisons de petits pas vers la réduction de la complexité de la programmation des systèmes intelligents, grâce à des outils de programmation. Tout d’abord, nous présentons un plugin pour la construction d’applications robotiques ([Chapter 2](#)). Ensuite, nous décrivons un langage spécifique à un domaine pour écrire des programmes différentiables ([Chapter 3](#)). En utilisant notre DSL ([Chapter 4](#)) comme véhicule, nous développons un cadre contradictoire pour tester les programmes différentiables et démontrons empiriquement son efficacité par rapport à une méthode d’échantillonnage probabiliste. Nous discutons ensuite d’une solution basée sur un conteneur pour la reproduction de programmes robotiques, et plus largement de tout système logiciel embarqué ayant des capacités visuomotrices ([Chapter 5](#)). Enfin, dans [Chapter 6](#) nous proposons quelques réflexions et prévisions pour l’avenir de la programmation des systèmes intelligents. Il reste encore beaucoup de travail à faire. Nous pensons que l’avenir est brillant et nous espérons que ceux qui se consacrent à sa construction s’inspireront des orientations proposées ici.

## 1.6. Iconographie

Tout au long de cette thèse, l'iconographie suivante est utilisée pour indiquer:

 Commandes shell destinées à un ordinateur, ou sortie dérivée de celui-ci.

 Références GrammarKit `.bnf`<sup>1</sup>

 Soit `Dockerfile`<sup>2</sup> ou Docker Compose<sup>3</sup> syntaxe.

 Commandes shell qui doivent être exécutées sur un Raspberry Pi.<sup>4</sup>

 Commandes shell Duckietown (`dts`).<sup>5</sup>

 Fichiers roslaunch `.launch`.<sup>6</sup>

 Code source de Python.<sup>7</sup>

 Code source de Kotlin.<sup>8</sup>

---

<sup>1</sup>GrammarKit usage notes: <https://github.com/JetBrains/Grammar-Kit/blob/master/HOWTO.md>

<sup>2</sup>Références Dockerfile: <https://docs.docker.com/engine/reference/builder/>

<sup>3</sup>Compose la référence du fichier: <https://docs.docker.com/compose/compose-file/>

<sup>4</sup>Raspberry Pi: <https://www.raspberrypi.org/>

<sup>5</sup>Duckietown Shell: <https://github.com/duckietown/duckietown-shell-commands>

<sup>6</sup>ROSLaunch XML: <https://wiki.ros.org/roslaunch/XML>

<sup>7</sup>Documentation Python: <https://www.python.org/doc/>

<sup>8</sup>Documentation Kotlin: <https://kotlinlang.org/docs/reference/>



# Chapter 2

---

## Outils de programmation pour la robotique

“L’espérance est que, dans quelques années, les cerveaux humains et les machines informatiques seront très étroitement couplés et que le partenariat qui en résultera pensera comme aucun cerveau humain n’a jamais pensé et traitera les données d’une manière qui n’est pas abordée par les machines de traitement de l’information que nous connaissons aujourd’hui.”

---

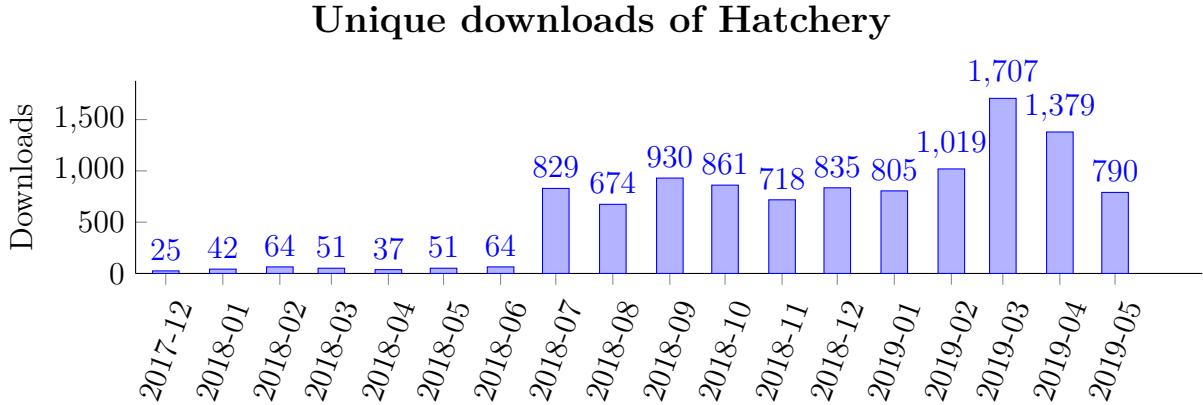
—Joseph Licklider [1992], *Symbiose homme-ordinateur*

Dans ce chapitre, nous aborderons la conception et la mise en œuvre d’un environnement de développement intégré (EDI) pour la construction de logiciels robotiques intelligents. Les robots modernes sont de plus en plus pilotés par des systèmes qui apprennent et s’améliorent au fil du temps. La plupart des chercheurs s’accordent à dire que les systèmes robotiques modernes n’ont pas encore atteint des capacités sensorimotrices biologiquement compétitives et que la plupart des systèmes intelligents ne sont pas physiquement incorporés. Cependant, nous estimons que tout système de contrôle en boucle fermée qui n’est pas explicitement programmé pour effectuer une tâche spécifique, mais qui l’apprend par expérience est un *système intelligent*. De plus, tout système en boucle fermée avec des moteurs physiques est un *système robotique*. Bien que la recherche ait démontré des applications réussies dans ces deux domaines séparément, il est largement admis que l’intégration des systèmes intelligents et de la robotique sera extrêmement fructueuse lorsqu’elle sera pleinement réalisée.

Hatchery est un outil conçu pour aider les programmeurs à écrire des applications robotiques en utilisant le middleware ROS. Au moment de sa sortie, Hatchery était le premier plugin ROS pour la [IntelliJ Platform](#)<sup>1</sup>, et aujourd’hui, est la plus utilisée avec plus de 10

---

<sup>1</sup>Une plateforme EDI pour le développement C/C++, Python et Android, entre autres.

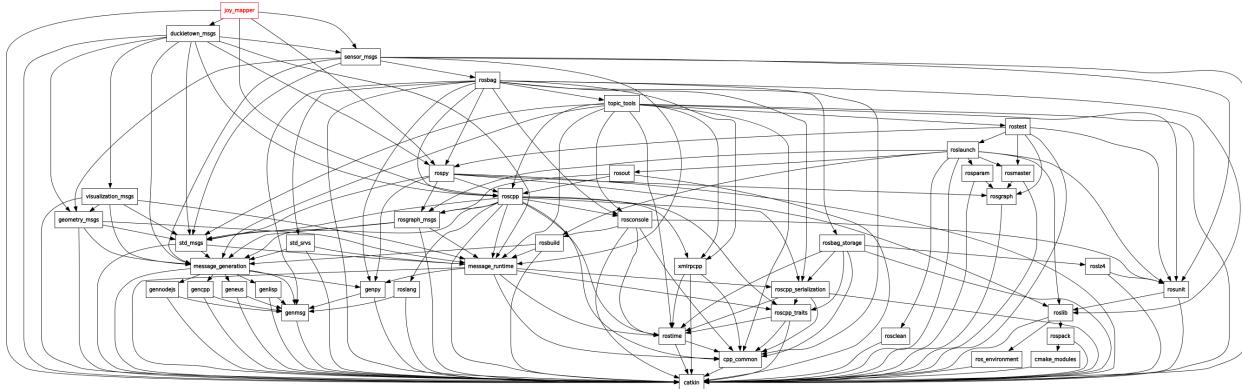


**Fig. 2.1.** Unique downloads of Hatchery between the time of its release and June 2019.  
<https://plugins.jetbrains.com/plugin/10290-hatchery>.

000 téléchargements uniques. Bien que l'idée soit simple, son absence antérieure et son adoption ultérieure suggèrent qu'il existe une demande non satisfaite pour de tels outils dans le développement de systèmes logiciels intelligents, en particulier dans les applications spécifiques à un domaine comme la robotique.

## 2.1. Introduction à ROS

Le système d'exploitation **Robot Operating System** (ROS) [Quigley et al., 2009] est un intericiel populaire pour les applications robotiques. Il fournit une infrastructure logicielle pour la messagerie distribuée, mais comprend également un ensemble de bibliothèques et d'outils graphiques développés par la communauté pour la création d'applications robotiques. ROS n'est pas un système d'exploitation (OS) au sens traditionnel du terme, mais il prend en charge des fonctionnalités similaires telles que la mémoire partagée et la communication entre processus. Contrairement aux systèmes purement axés sur les messages tels que DDS [Pardo-Castellote, 2003] et ZMQ [Hintjens, 2013], ROS fournit, outre l'infrastructure de communication, des API spécifiques pour la construction de systèmes robotiques décentralisés, notamment ceux qui sont capables de se déplacer. Cela comprend des bibliothèques standard pour la sérialisation et la désérialisation de données géométriques, de cadres de coordonnées, de cartes, de messages de capteurs et d'images.



**Fig. 2.2.** Une application ROS typique contient un grand graphique de dépendances.

Le middleware ROS fournit plusieurs frontaux de langage pour la programmation polyglotte. Selon un recensement communautaire effectué en 2018, 55% de toutes les applications ROS sur GitHub sont écrites en C/C++, suivies par Python avec une part de 25% de développeurs. Le code source d'une application ROS typique contient un mélange de code C/C++ et Python, correspondant aux préférences linguistiques respectives des communautés de la robotique et de l'apprentissage machine. Hatchery est compatible avec la plupart des bibliothèques clientes ROS courantes, notamment [rosjava](#) pour Java, [rospython](#) pour Python, [roscpp](#) pour C/C++, et d'autres frontaux de langage.

Un projet ROS typique comporte plusieurs composantes, dont le code source, les fichiers de configuration, l'infrastructure de construction, les artefacts compilés et l'environnement de déploiement. Pour construire une application ROS simple, plusieurs étapes sont nécessaires. Tout d'abord, il faut installer le système ROS, qui n'est officiellement supporté que sur les distributions Linux basées sur Debian.<sup>2</sup> En supposant que le ROS a été installé à l'emplacement par défaut, il peut être localisé comme tel :

```
~$ source /opt/ros/<ROS DISTRO>/setup.bash
```

Une application ROS minimale contient au moins un *éditeur* et un *abonné*, qui font passer les messages sur un canal de communication partagé. L’éditeur peut être défini comme suit :

---

<sup>2</sup>Les instructions détaillées d'installation peuvent être trouvées ici : <https://wiki.ros.org/ROS/>

```
./catkin_ws/src/pubsub/publisher.py  
1  
2  
3  
4  
5  
6  
7  
8  
9  
  
import rospy  
from std_msgs.msg import String  
  
pub = rospy.Publisher("channel", String, queue_size=10)  
rospy.init_node("publisher", anonymous=True)  
rate = rospy.Rate(10)  
while not rospy.is_shutdown():  
    pub.publish("Some message")  
    rate.sleep()
```

Au fur et à mesure que l'éditeur écrit des messages sur

# Chapter 3

---

## Type-safe differentiable programming

Bien que la notation mathématique possède sans aucun doute des règles d'analyse, celles-ci sont assez lâches, parfois contradictoires et rarement clairement énoncées. En raison de leur application à un large éventail de sujets, de leur grammaire stricte et de leur interprétation stricte, les langages de programmation peuvent apporter de nouvelles idées sur la notation mathématique. ”

---

—Kenneth Iverson [1999], *Math pour la Layman*

Dans ce chapitre, nous aborderons la théorie et la mise en uvre d'un langage spécifique à un domaine et sans danger pour les types, destiné à la différentiation automatique (AD), un algorithme ayant diverses applications dans l'optimisation numérique et l'apprentissage machine. L'idée clé de l'AD est assez simple. Un petit ensemble d'opérations mathématiques primitives constitue la base de tous les ordinateurs modernes, et en composant ces opérations sur les nombres réels de manière ordonnée, on peut calculer n'importe quelle fonction calculable. Dans l'apprentissage machine, on nous donne souvent une fonction calculable sous la forme d'un programme qui ne fonctionne pas correctement. Nous voudrions un algorithme pour déterminer comment modifier légèrement l'entrée, afin de produire une sortie plus appropriée.

Un tel algorithme a d'abord été conçu par Wengert [1964], dont la méthode est connue aujourd'hui sous le nom de AD en mode avancé. Peu de temps après, un certain Richard Bellman a reproduit l'algorithme de Wengert pour estimer numériquement la dynamique orbitale d'un système à deux corps, reconnaissant son potentiel pour, "le traitement de grands systèmes d'équations différentielles qui ne pourraient pas être entrepris autrement" [Bellman et al., 1965]. C'est à peu près à la même époque qu'apparurent les principaux détails de l'algorithme de rétropropagation [?]. C'est dans la Linnainmaa [1976] que l'idée

de calculer des dérivées sur des graphiques de calcul a été enregistrée pour la première fois. L'algorithme de Linnainmaa était particulièrement important pour les réseaux de neurones, et est aujourd'hui connu sous le nom de AD en mode inverse. Mais ce n'est qu'en 2010 que les outils logiciels standard [Bergstra et al., 2010] pour l'AD sont devenus largement disponibles dans l'apprentissage machine. C'est ici que commence notre voyage.

### 3.1. Automatic differentiation

Lorsqu'une fonction est dotée d'une certaine entrée, AD nous indique comment modifier l'entrée d'un montant minimal, afin de modifier les sorties au maximum. Supposons qu'on nous donne une fonction  $P_k : \mathbb{R} \rightarrow \mathbb{R}$ , composée d'une série de fonctions imbriquées, chacune de même type:

$$P_k(x) = \begin{cases} p_1 \circ x = x & \text{if } k = 1 \\ p_k \circ P_{k-1} \circ x & \text{si } k > 1 \end{cases} \quad (3.1.1)$$

De la règle de la chaîne, on rappelle que le dérivé d'une composition est un produit des dérivés:

$$\frac{dP}{dp_1} = \frac{dp_k}{dp_{k-1}} \frac{dp_{k-1}}{dp_{k-2}} \cdots \frac{dp_2}{dp_1} = \prod_{i=1}^{k-1} \frac{dp_{i+1}}{dp_i} \quad (3.1.2)$$

Étant donné  $Q(q_1, \dots, q_m) : \mathbb{R}^m \rightarrow \mathbb{R}$ , le *gradient* est une fonction  $\nabla Q : \mathbb{R}^m \rightarrow \mathbb{R} \rightarrow \mathbb{R}^m$  définie comme:

$$\nabla Q = \left[ \frac{\partial Q}{\partial q_1}, \dots, \frac{\partial Q}{\partial q_m} \right] \quad (3.1.3)$$

Le *Hessien* est une fonction  $\mathbf{H} : \mathbb{R}^m \rightarrow \mathbb{R} \rightarrow \mathbb{R}^{m \times m}$  renvoyant une matrice de partiels de second ordre:

$$\mathbf{H}(Q) = \begin{bmatrix} \frac{\partial^2 Q}{\partial x_1^2} & \frac{\partial^2 Q}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 Q}{\partial x_1 \partial x_m} \\ \frac{\partial^2 Q}{\partial x_2 \partial x_1} & \frac{\partial^2 Q}{\partial x_2^2} & \cdots & \frac{\partial^2 Q}{\partial x_2 \partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 Q}{\partial x_m \partial x_1} & \frac{\partial^2 Q}{\partial x_m \partial x_2} & \cdots & \frac{\partial^2 Q}{\partial x_m^2} \end{bmatrix} \quad (3.1.4)$$

Pour les fonctions vectorielles  $\mathbf{f} : \mathbb{R}^m \rightarrow \mathbb{R}^n$ , le Jacobien,  $\mathcal{J}_{\mathbf{f}} : \mathbb{R}^m \rightarrow \mathbb{R}^{n \times m}$  est défini comme:

$$\mathcal{J}_{\mathbf{f}} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \nabla f_1 \\ \vdots \\ \nabla f_m \end{bmatrix} \quad (3.1.5)$$

Pour les fonctions scalaires, la transposition du Hessien est équivalente au Jacobien du gradient:

$$\mathbf{H}(Q)^T = \mathcal{J}_{\mathbf{q}}(\nabla Q) \quad (3.1.6)$$

Pour une fonction vectorielle  $\mathbf{P}_k(\mathbf{x}) : \mathbb{R}^m \rightarrow \mathbb{R}^n$ , la règle de la chaîne de Eq. 3.1.2 s'applique toujours:

$$\mathcal{J}_{\mathbf{P}_k} = \prod_{i=1}^k \mathcal{J}_{p_i} = \underbrace{\left( \left( (\mathcal{J}_{p_k} \mathcal{J}_{p_{k-1}}) \dots \mathcal{J}_{p_2} \right) \mathcal{J}_{p_1} \right)}_{\text{"Reverse accumulation"}} = \underbrace{\left( \mathcal{J}_{p_k} \left( \mathcal{J}_{p_{k-1}} \dots (\mathcal{J}_{p_2} \mathcal{J}_{p_1}) \right) \right)}_{\text{"Forward accumulation"}} \quad (3.1.7)$$

Pour l'exhaustivité, mais rarement utilisé dans la pratique, est le deuxième ordre partiel pour les fonctions vectorielles:

$$\mathbf{H}(\mathbf{f}) = [\mathbf{H}(f_1), \mathbf{H}(f_2), \dots, \mathbf{H}(f_n)] \quad (3.1.8)$$

Nous pouvons utiliser ces outils pour calculer la direction afin d'ajuster les entrées d'une fonction calculable, afin de modifier au maximum la sortie de cette fonction, c'est-à-dire la direction de la descente la plus raide.

Parfois, une fonction a la propriété qu'étant donné une entrée  $a$ , peu importe comment  $a$  est modifié, la sortie reste la même. Nous disons que de telles fonctions ont une pente nulle pour cette entrée.

$$(\nabla F)(a) \approx \mathbf{0} \quad (3.1.9)$$

Le coût du calcul du Hessois,  $\mathbf{H}$  est approximativement quadratique [Griewank, 1993] par rapport au nombre de variables indépendantes sous différentiation. Si  $\mathbf{H}(a)$  est tractable à calculer et inversible, nous pourrions utiliser le test de la dérivée seconde partie pour déterminer que:

- (1) Si toutes les valeurs propres de  $\mathbf{H}(a)$  sont positives,  $a$  est un minimum local
- (2) Si toutes les valeurs propres de  $\mathbf{H}(a)$  sont négatives,  $a$  est un maximum local

- (3) Si  $\mathbf{H}$  contient un mélange de valeurs propres positives et négatives,  $a$  est un *saddle point*

Pour certaines classes de fonctions calculables, de petites modifications de l'entrée produiront une variation soudaine et importante de la sortie. Nous disons que ces fonctions sont non différentiables.

$$\|(\nabla F)(a)\| \approx \pm\infty \quad (3.1.10)$$

La question de savoir si des fonctions non différentiables existent dans le monde réel est ouverte [Buny et al., 2005]. À l'échelle physique (10nm) et temporelle (10ns) actuelle de l'informatique moderne, il n'existe pas de telles fonctions, mais la plupart des ordinateurs modernes sont incapables de rendre compte de la valeur réelle de leurs fonctions à valeur binaire. À toutes fins utiles, les programmes mis en uvre par la plupart des ordinateurs physiques sont des relations discrètes. Néanmoins, les programmes discrets sont capables d'approcher des fonctions bornées sur  $\mathbb{R}^m$  avec une précision arbitraire si l'on tient compte du temps et de l'espace. Pour la plupart des applications, une approximation de faible précision (32-64 bits) est suffisante.

Il existe au cur de l'apprentissage machine un théorème qui énonce une famille simple de fonctions, qui calculent une somme pondérée d'une fonction non linéaire  $\varphi : \mathbb{R} \rightarrow \mathbb{R}$  composée d'une fonction linéaire  $\theta^\top \mathbf{x} + b$ , peut approximer toute fonction bornée  $\mathbb{R}^m \rightarrow \mathbb{R}$  à une précision arbitraire. Plus précisément, le théorème d'approximation universel [Hornik et al., 1989] indique que pour toutes les fonctions continues à valeur réelle  $f : C(\mathbb{I}_m)$ , où  $\mathbb{I}_m = [0, 1]^m \rightarrow [0, 1]$ , il existe un  $\hat{f} : \mathbb{R}^m \times \mathbb{R}^{n \times m} \rightarrow \mathbb{R}$ , paramétrée par  $\Theta \in \mathbb{R}^{n \times m}$ , en prenant une entrée  $\mathbf{x} \in [0, 1]^m$  et des constantes  $n \in \mathbb{N}, \beta \in \mathbb{R}^n, \mathbf{b} \in \mathbb{R}^n, \epsilon \in \mathbb{R}^+$  tel que la déclaration suivante tient:

$$\begin{aligned} \hat{f}(\mathbf{x}; \Theta) &= \beta^\top \varphi_{\odot}(\Theta^\top \mathbf{x} + \mathbf{b}) \\ \forall \mathbf{x} \in \mathbb{I}_m, \quad |\hat{f}(\mathbf{x}) - f(\mathbf{x})| &< \epsilon \end{aligned} \quad (3.1.11)$$

$\varphi_{\odot}$  indique une fonction non linéaire  $\varphi$  appliquée par élément au vecteur. Ce théorème nous dit seulement que  $\Theta$  existe, mais ne nous dit pas comment le trouver et ne fixe pas de limite supérieure à la constante  $n$ , ce qui limite quelque peu son applicabilité pratique. Mais pour des raisons encore mal comprises, les résultats empiriques suggèrent qu'il est possible d'approximer de nombreuses fonctions naturelles en un nombre relativement court d'étapes

en composant plusieurs *couches* de  $\Theta^\top \mathbf{x} + \mathbf{b}$  et  $\varphi$  en alternance, et en mettant à jour chaque  $\Theta$  en utilisant une procédure basée sur la descente de la pente. Le modèle qui en résulte peut être exprimé comme suit<sup>1</sup>,

$$\widehat{\mathbf{P}}_k(\mathbf{x}; \Theta) = \begin{cases} \widehat{\mathbf{p}}_1(\Theta_1) \circ \mathbf{x} & \text{si } k = 1 \\ \widehat{\mathbf{p}}_k(\Theta_k) \circ \widehat{\mathbf{P}}_{k-1}(\Theta_{[1, k-1]}) \circ \mathbf{x} & \text{si } k > 1 \end{cases} \quad (3.1.12)$$

$\Theta = \{\Theta_1, \dots, \Theta_k\}$  sont des paramètres libres et  $\mathbf{x} \in \mathbb{R}^m$  est une entrée unique. Pour obtenir approximativement  $\mathbf{P}(\mathbf{x})$ , il faut obtenir  $\mathbf{X} = \{\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(z)}\}$ ,  $\mathbf{Y} = \{\mathbf{y}^{(0)} = \mathbf{P}(\mathbf{x}^{(0)}), \dots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})\}$  en quantité aussi grande et variée que possible et répéter la procédure suivante jusqu'à ce que  $\Theta$  converge:

$$\Theta \leftarrow \Theta - \alpha \frac{1}{z} \nabla_\Theta \sum_{i=1}^z \mathcal{L}(\widehat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \Theta), \mathbf{y}^{(i)}) \quad (3.1.13)$$

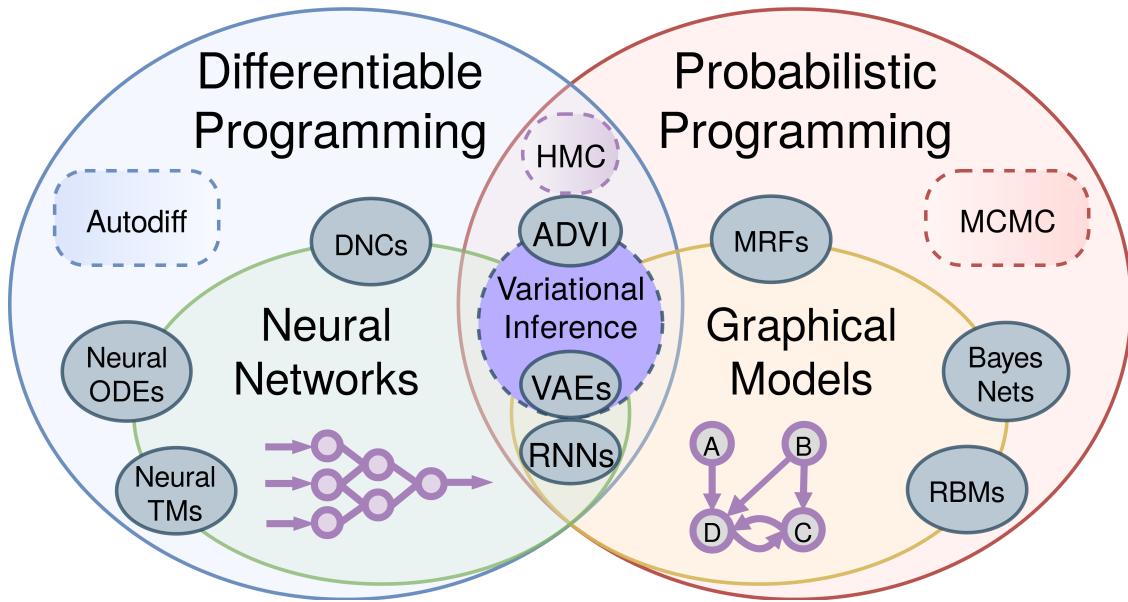
Dans le cas général, nous pouvons résoudre le gradient en utilisant Eq. 3.1.7. Pour les  $\mathcal{L}$  les plus courants, la complexité de cette procédure est linéaire avec  $z$ . Comme  $z$  peut être assez important en pratique, et comme l'obtention du gradient exact n'est pas importante, nous utilisons une variante stochastique en rééchantillonnant un *minibatch*  $\mathbf{X}', \mathbf{Y}'$  composé de paires  $\mathbf{x}^{(i)}, \mathbf{y}^{(i)}$  pour  $i \sim \{0, \dots, z\}$  sans remplacement à chaque étape de mise à jour. C'est légèrement plus bruyant, mais fonctionne beaucoup plus rapidement.

## 3.2. Differentiable programming

La renaissance de l'apprentissage profond moderne est largement attribuée aux progrès réalisés dans trois domaines de recherche : les algorithmes, les données et le matériel. Parmi les algorithmes, la plupart des recherches ont porté sur les architectures d'apprentissage profond et l'apprentissage par représentation. Le rôle que la différentiation automatique (DA) a joué pour faciliter la mise en uvre de ces idées est sans doute tout aussi important. Avant l'avènement des bibliothèques AD à usage général telles que [Theano](#), [PyTorch](#) et [TensorFlow](#), les gradients devaient être dérivés manuellement. L'adoption généralisée des logiciels de DA a simplifié et accéléré le rythme de l'apprentissage automatique basé sur les gradients, permettant aux chercheurs de construire des architectures de réseau plus profondes

---

<sup>1</sup>La notation ci-dessous suppose une certaine familiarité avec le curry et l'application de fonctions partielles, dans lesquelles  $\widehat{\mathbf{P}} : \mathbb{R}^m \rightarrow \mathbb{R}^n$  et  $l'arcade\mathbb{R} \rightarrow \mathbb{R}_m \rightarrow \mathbb{R}^n$ . Pour plus de détails, voir [Schönfinkel \[1924\]](#), [Curry and Feys \[1958\]](#) et al.



**Fig. 3.1.** *Programmation différentiable* comprend les réseaux de neurones, mais plus largement, les programmes arbitraires qui utilisent l'optimisation par gradient pour se rapprocher d'une fonction de perte. *Programmation probabiliste* [Tristan et al., 2014, Carpenter et al., 2017, Gorinova et al., 2019] est une généralisation des modèles graphiques probabilistes qui utilise les méthodes de Monte Carlo (MC) pour approcher une fonction de densité.

et de nouvelles représentations d'apprentissage. Certaines de ces idées ont à leur tour servi de base à de nouvelles méthodes de DA, qui continue d'être un [domaine actif](#) de la recherche dans les communautés du langage de programmation et du calcul scientifique.

Un aspect clé du paradigme connexionniste est la descente en gradient d'une fonction de perte statistique définie sur un réseau neuronal par rapport à ses paramètres libres. Pour que la descente de gradient fonctionne, la représentation doit pouvoir être différentiable presque partout. Cependant, de nombreuses représentations ne sont pas différentiables dans leur domaine naturel. Par exemple, la structure du langage écrit n'est pas facilement différentiable, car de petites modifications de la forme symbolique d'un mot peuvent entraîner des changements soudains de sa sémantique [van Merriënboer, 2018]. L'un des principaux enseignements de l'apprentissage de la représentation est que de nombreux types de données discrètes peuvent être mis en correspondance dans un espace latent plus lisse. Par exemple, si nous représentons les mots comme un vecteur de nombres réels,  $\mathbb{R}^N$ , alors il est possible d'apprendre une correspondance des mots à  $\mathbb{R}^N$  de sorte que les relations sémantiques entre

les mots (telles que définies par leur co-occurrence statistique dans les grands corpus) sont géométriquement préservées dans l'espace vectoriel [Pennington et al., 2014] – les mots ayant une signification similaire sont associés à des vecteurs similaires. De nombreuses classes de problèmes discrets peuvent être relâchées pour devenir des substituts continus en apprenant de telles représentations, ou *embeddings* de manière non supervisée, ou semi-supervisée.

À peu près à la même époque, la communauté d'apprentissage profond a réalisé que la différenciation stricte n'était peut-être pas si importante tout au long du processus. Il a été démontré en pratique que les ordinateurs utilisant l'arithmétique à virgule flottante 8 bits [Wang et al., 2018d] et l'arithmétique des nombres entiers [Wu et al., 2018, Jacob et al., 2017] sont capables de former des réseaux de neurones sans sacrifier les performances. Des hypothèses fortes comme la continuité de Lipschitz et la douceur  $\beta$  autrefois considérées comme indispensables pour l'apprentissage par gradient peuvent être assouplies, tant que le bruit introduit par la quantification est négligeable par rapport aux méthodes de gradient stochastique. Avec le recul, cela aurait dû être moins surprenant, puisque tous les ordinateurs numériques utilisent de toute façon des représentations discrètes et ont été capables de former des réseaux neuronaux pendant près d'un demi-siècle. Cela suggère qu'une différenciation stricte n'était pas aussi importante que d'avoir une bonne métrique. Tant que la surface de perte permet l'apprentissage de la métrique, la descente de gradient est étonnamment résistante à la quantification.

Au fur et à mesure que l'apprentissage en profondeur développait de nouvelles applications, les chercheurs ont observé que les réseaux neuronaux faisaient partie d'une classe plus large d'architectures différentiables qui pouvaient être structurées d'une manière similaire aux programmes informatiques. D'où le terme *programmation différentiable* [Olah, 2015, Baydin, Plotkin, 2018] (DP) est né. Aujourd'hui, la DP a de nombreuses applications, des techniques classiques de CS comme le classement et le tri [Cuturi et al., 2019, Blondel et al., 2020], au pliage des protéines [AlQuraishi, 2018], aux moteurs physiques [Hu et al., 2019, de Avila Belbute-Peres et al., 2018, Degrave et al., 2016] et au rendu graphique [Loper and Black, 2014] au meta-learning [Liu et al., 2018, Chandra et al., 2019]. Ces applications ont toutes des paramètres qui peuvent être appris via la descente de gradient. Pour apprendre des relations discrètes sans intégration ad hoc, des techniques supplémentaires (§ 3.20), telles que la programmation probabiliste, sont probablement nécessaires. Divers langages

de programmation probabiliste, dont Stan [Carpenter et al., 2017], Pyro [Bingham et al., 2019], PyMC4 [Kochurov et al., 2019] et autres, ont également vu le jour. Comme le montre Fig. 3.1, ces deux domaines ont bénéficié de nombreuses collaborations fructueuses ces dernières années.

### 3.3. Static and dynamic languages

La plupart des programmes d'apprentissage machine et de calcul scientifique sont écrits dans des langages dynamiques, tels que Python. En revanche, la plupart des industries utilisent des langages à caractères statiques [Ray et al., 2017]. Selon certaines études, les erreurs liées aux types représentent plus de 15

Le typage statique élimine une large classe d'erreurs d'exécution, permettant aux développeurs et aux outils de raisonner plus soigneusement sur le comportement des programmes sans avoir besoin de les exécuter. En plus d'une validation syntaxique renforcée pour la programmation générale, une bibliothèque bien conçue dans un langage fortement typé peut éliminer les erreurs spécifiques à un domaine liées à une mauvaise utilisation de l'API qui, autrement, nécessiteraient une documentation et des échantillons de code pour les éviter, ce qui améliore la convivialité et réduit la maintenance. En outre, les systèmes de types forts permettent aux EDI de fournir des outils d'analyse statique plus précis, tels que l'autocomplétion pertinente, la navigation dans le code source et la détection plus précoce des erreurs d'exécution.

Une objection courante à l'utilisation de langages à caractères forts est la charge supplémentaire que représente l'annotation manuelle des types [Ore et al., 2018]. Alors que les premiers langages à sécurité typographique comme C++ et Java exigeaient des programmeurs qu'ils annotent de manière exhaustive les déclarations de fonctions et de variables, avec une utilisation judicieuse de l'inférence de type dans les langages modernes comme Kotlin, Scala, Rust et autres, la plupart des signatures de type peuvent être omises sans risque et facilement récupérées dans le contexte environnant. L'inférence de type permet aux langues modernes d'offrir la brièveté des langues à typographie dynamique avec la sécurité d'une vérification statique des types.

### 3.4. Langages impératifs et fonctionnels

La plupart des programmes sont aujourd’hui écrits dans le style impératif, en raison de la prédominance de la machine de Turing et de l’architecture von Neumann [Backus, 1978].  $\lambda$ -calculus fournit une note de bas de page équivalente en ce sens que la machine de Turing et  $\lambda$ -calculus sont tous deux des langages de Turing complets. Pour le calcul, ce qui, selon nous, est une notation plus appropriée pour exprimer des fonctions mathématiques et calculer leurs dérivés. Dans la programmation impérative, le seul but de l’utilisation d’une fonction est de lui transmettre des valeurs, et il n’y a pas moyen de se référer à une fonction sans le faire. Ce qui est plus troublant dans le cas de la DA, c’est que les programmes impératifs ont un état mutable, ce qui nécessite de prendre des précautions supplémentaires lors du calcul de leurs dérivées.

La notion mathématique de composition des fonctions est un citoyen de premier ordre en matière de programmation fonctionnelle. Tout comme en calcul, pour prendre la dérivée d’un programme composé avec un autre programme, on applique simplement la règle de la chaîne (§ 3.1). Comme il n’y a pas d’état mutable dans la PF, aucune structure de données exotiques ou d’astuces de compilation n’est nécessaire.

Par exemple, considérons la fonction vectorielle  $f(l_1, l_2) = l_1 \cdot l_2$ , vue dans Fig. 3.2. Les programmes impératifs, en permettant la mutation, détruisent effectivement les informations intermédiaires. Afin de récupérer le graphique de calcul pour la MA en mode inversé, nous devons soit passer outre l’opérateur d’affectation, soit utiliser une bande pour stocker les valeurs intermédiaires. Dans la programmation fonctionnelle pure, les variables mutables n’existent pas, ce qui nous facilite grandement la vie.

La programmation fonctionnelle permet à Kotlin $\nabla$  d’utiliser la même abstraction pour représenter les fonctions mathématiques et les fonctions de programmation. Toutes les fonctions de Kotlin $\nabla$  sont des fonctions pures, composées d’expressions formant un graphe de flux de données (DFG). Une expression est simplement une **Function**, qui n’est évaluée que lorsqu’elle est invoquée avec des valeurs numériques, par exemple `z(0, 0)`. De cette manière, Kotlin $\nabla$  est similaire à d’autres cadres basés sur des graphes comme [Theano](#) et [TensorFlow](#).

Impératif et fonctionnel	
<pre> 1 <b>fun</b> dot(l1, l2) { 2     <b>if</b> (len(l1) != len(l2)) 3         <b>return</b> error 4     <b>var</b> sum = 0 5     <b>for</b>(i <b>in</b> 0 <b>to</b> len(l1)) 6         sum += l1[i] * l2[i] 7     <b>return</b> sum 8 }</pre>	<pre> <b>fun</b> dot(l1, l2) {     <b>return if</b> (len(l1) != len(l2))         error     <b>else if</b> (len(l1) == 0) 0     <b>else</b>         head(l1) * head(l2) +         dot(tail(l1), tail(l2))</pre>

**Fig. 3.2.** Deux programmes équivalents, tous deux mettant en uvre la fonction  $f(l_1, l_2) = l_1 \cdot l_2$ .

### 3.5. Kotlin

Lors de la programmation dans un langage à caractères statiques, une question courante que l'on peut poser au compilateur est la suivante : "En donnant une valeur, `x`, peut-on assigner `x` à une variable de type `Y`? (par exemple, vérification du type `x instanceof Y`) En Java, cette question s'avère être [ill-posed](#) [Amin and Tate, 2016] et indécidable [Grigore, 2017] dans le cas général. Il est possible de construire un programme Java dans lequel la réponse est "oui" indépendamment de `Y`, ou pour lequel une réponse ne peut pas toujours être déterminée en temps fini. L'indécidabilité n'est pas nécessairement un obstacle, mais le manque de solidité de Java est plus critique et la manière de le corriger n'est pas claire, même si cela se produit rarement dans la pratique.

Kotlin est un langage de type statique qui convient bien à la construction d'applications multiplateformes, avec un support de compilation pour JVM, JavaScript et des cibles natives. Contrairement à la plupart des langages de programmation, Kotlin a été conçu dès le départ avec le support de l'IDE, et a gagné une certaine notoriété dans l'écosystème JVM grâce à son ergonomie. Le système de types de Kotlin [Tate, 2013] est strictement [moins expressif](#), mais totalement interopérable avec celui de Java. On ignore si les mêmes problèmes qui affectent le système de types de Java sont présents dans Kotlin, mais l'interopérabilité avec Java a élargi son adoption et reste un élément clé de la convivialité du langage.

Dans ce travail, nous utilisons plusieurs caractéristiques du langage propres à Kotlin, telles que les fonctions de première classe (§ 3.12), les fonctions d'extension (§ 3.16), la

surcharge des opérateurs (§ 3.11) et les types de données algébriques (§ 3.14). En outre, nous utilisons largement le [support SDL](#) de Kotlin pour mettre en uvre la programmation de tableaux à sécurité de forme. Ensemble, ces caractéristiques de langage fournissent une plate-forme concise, flexible et sans risque de type pour la programmation mathématique.

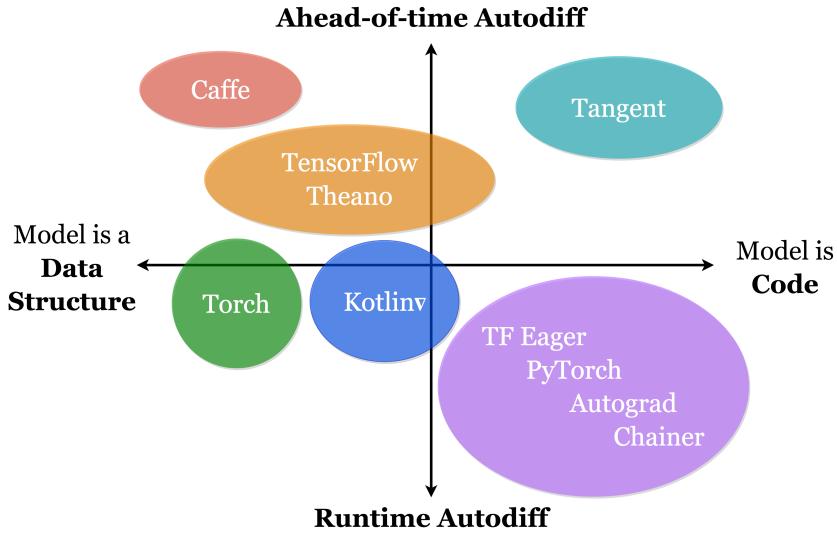
### 3.6. Kotlin $\nabla$

Des travaux antérieurs ont démontré la possibilité d'encoder un langage déterministe sans contexte (DCF) dans le système de type Java comme *interface fluide* [Gil and Levy, 2016, Nakamaru et al., 2017]. Ce résultat a été renforcé pour prouver que le système de types de Java est complet de Turing (TC) [Grigore, 2017], ce qui nous permet d'effectuer des contrôles de forme et des inférences sur des programmes de tableaux écrits en Java. Kotlin est un descendant de Java qui est au moins DCF au niveau du type. Kotlin $\nabla$ , un DSL intégré dans le langage Kotlin est TC au niveau de la valeur et DCF au niveau du type. Une approche similaire est possible dans la plupart des langues avec des types génériques.

La programmation différenciée a une histoire riche parmi les langages dynamiques comme Python, Lua et JavaScript, avec des implémentations précoces comprenant des projets comme [Theano](#) [Bergstra et al., 2010], [Torch](#) [Collobert et al., 2002], et [TensorFlow](#) [Abadi et al., 2016]. Des idées similaires ont été mises en uvre dans des langages fonctionnels tels que Scheme ([Stalin \$\nabla\$](#)  [Pearlmutter and Siskind, 2008b]), et des langages à caractères statiques comme F# ([DiffSharp](#) [Baydin et al., 2015b]) et [Swift](#) [Lattner and Wei, 2018]. Cependant, la majorité des bibliothèques de différenciation automatique (AD) existantes utilisent une DSL à type lâche, et peu d'entre elles proposent des opérations de tenseur à forme sûre dans un langage de programmation largement utilisé.

Les implémentations AD existantes pour la JVM comprennent [Lantern](#) [Wang et al., 2018b], [Nexus](#) [Chen, 2017] et [DeepLearning.scala](#) [Bo, 2018], mais elles sont basées sur Scala et n'interopèrent pas avec d'autres langages JVM. Kotlin $\nabla$  est entièrement interopérable avec Java vanille, ce qui permet une adoption plus large dans les langages voisins. À notre connaissance, Kotlin n'a pas d'implémentation AD préalable. Cependant, le langage possède plusieurs caractéristiques utiles pour la mise en uvre d'un cadre AD natif. Kotlin $\nabla$  repose principalement sur les caractéristiques suivantes du langage:

- Les fonctions



**Fig. 3.3.** Avec l'adaptation de van Merriënboer et al. [2018]. Les modèles Kotlin $\nabla$  sont des structures de données, construites par une DSL intégrée, ardemment optimisées et évaluées paresseusement.

- **Operator overloading and infix functions** permettent une notation concise pour définir des opérations arithmétiques sur des structures tensorielles-algébriques, c'est-à-dire des groupes, des anneaux et des champs.
- **$\lambda$ -fonctions** supportent la programmation fonctionnelle, suivant Pearlmutter and Siskind [2008a,b], Siskind and Pearlmutter [2008], Elliott [2009, 2018], et al.
- **Fonctions d'extension** prennent en charge l'extension des classes avec de nouveaux champs et méthodes qui peuvent être exposés à des appelants externes sans nécessiter de sous-classement ou d'héritage.

Les modèles Kotlin $\nabla$  sont des langages intégrés spécifiques à un domaine (eDSL). Ces langages peuvent apparaître et se comporter différemment du langage hôte, mais ne sont en réalité que des fonctions soigneusement déguisées pour construire un arbre syntaxique abstrait (AST). Souvent, ces AST représentent de simples machines à états, mais sont également utilisés pour intégrer un langage de programmation. Les exemples les plus courants sont SQL/LINQ [Meijer et al., 2006], OptiML [Sujeeth et al., 2011] et autres interfaces fluides [Fowler, 2005]. Dans un langage hôte suffisamment expressif, on peut implémenter n'importe quel langage comme une bibliothèque, sans avoir besoin d'écrire un lexique, un analyseur, un compilateur ou un interpréteur. Et avec un typage approprié, les utilisateurs

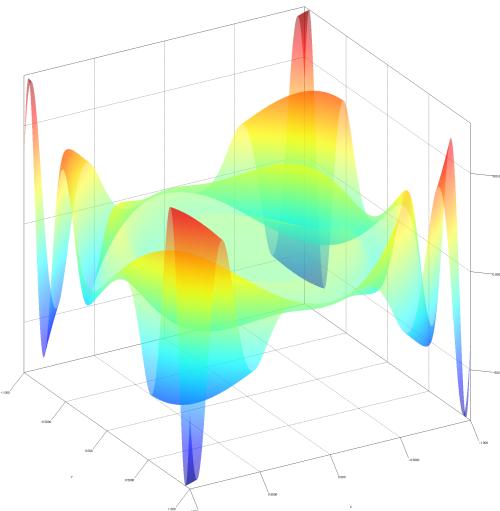
recevront des compléments de code et des analyses statiques de leurs outils de développement préférés. Les langages fonctionnels sont souvent des langages hôtes appropriés [Elliott et al., 2003, Rompf and Odersky, 2010], peut-être en raison de la notion de code en tant que données.

### 3.7. Usage

Kotlin $\nabla$  permet aux utilisateurs de mettre en uvre des programmes différentiables en composant des expressions. Considérons le programme Kotlin $\nabla$  suivant avec deux entrées et une sortie:

```


    1 with(DoublePrecision) { // Utilise des chiffres en double précision
    2     val x par Var() // Déclarer des variables immuables (ces variables
    3     val y par Var() // sont seulement utilisées pour la différentiation)
    4     val z = sin(10 * (x * x + pow(y, 2))) / 10 // évaluation paresseuse
    5     val dz_dx = d(z) / d(x) // Notation Leibniz [Christianson, 2012]
    6     val d2z_dx dy = d(dz_dx) / d(y) // Mélange de partiels d'ordre supérieur
    7     val d3z_d2xdy = grad(d2z_dx dy)[x] // Équivalent à d(f)/d(x)
    8     plot3D(d3z_d2xdy, -1.0, 1.0) // Tracé en espace 3 (-1 < x, y, z < 1)
    9 }
```

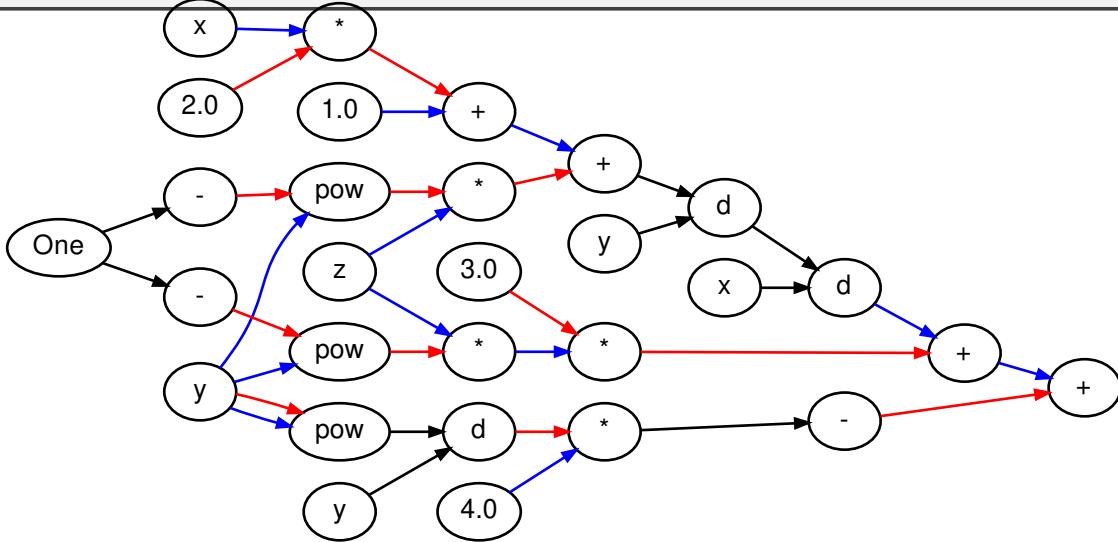


Ci-dessus, nous définissons une fonction à deux variables et prenons une série de dérivées partielles par rapport à chaque variable. Les expressions sont évaluées paresseusement dans



```
val t = (1 + x * 2 + z / y).d(y).d(x) + z / y * 3 - 4 * (y pow y).d(y)
```

1



**Fig. 3.4.** Implicit DFG constructed by the original expression, shown above.

un contexte numérique, qui peut être importé par fichier ou lexicalement pour un contrôle plus fin du comportement d'exécution. La fonction est évaluée numériquement sur l'intervalle  $(-1, 1)$  dans chaque dimension et rendue dans l'espace 3. Pour une grammaire complète, veuillez vous référer à § 7.1. Nous pouvons également tracer des collecteurs à dimensions plus élevées (par exemple, la surface de perte d'un réseau neuronal), projetés en quatre dimensions et rendus en trois, où un axe est représenté par le temps.

### 3.8. Type systems

Les premiers travaux sur l'analyse dimensionnelle sans risque de type se trouvent dans Kennedy [1994, 1996] qui utilise les types pour coder la dimensionnalité et empêcher l'apparition de bogues courants liés à la non-concordance des dimensions, et a été réalisé plus tard dans le langage F# [Kennedy, 2010]. Jay and Sekanina [1997], Rittri [1995], et Zenger [1997] explorent l'application des types de dimension pour l'algèbre linéaire. Plus récemment, Kiselyov [2005], Kiselyov et al. [2009] et Griffioen [2015], montrent comment manipuler des tableaux de manière plus complexe. Avec le regain d'intérêt pour l'algèbre des tenseurs et la programmation des tableaux, Chen [2017] et Rink [2018] montrent comment coder la sécurité des formes pour l'algèbre des tenseurs dans divers systèmes de types.

Le problème que nous essayons de résoudre peut être résumé comme suit. Étant donné deux valeurs  $x$  et  $y$ , et l'opérateur  $\$$ , comment déterminer si l'expression  $z = x \$ y$  est valide, et si oui, quel est le type de résultat de  $z$ ? Pour la multiplication matricielle, lorsque  $x \in \mathbb{R}^{m \times n}$  et  $y \in \mathbb{R}^{n \times p}$ , l'expression est bien tapée et nous pouvons déduire  $z \in \mathbb{R}^{m \times p}$ . Plus généralement, nous voudrions déduire le type de  $z$  pour un opérateur quelconque  $@ : (\mathbb{R}^a, \mathbb{R}^b) \rightarrow \mathbb{R}^c$  où  $a \in \mathbb{N}^q, b \in \mathbb{N}^r, c \in \mathbb{N}^s$  et  $q, r, s \in \mathbb{N}$ . Pour de nombreuses opérations d'algèbre linéaire telles que la multiplication de matrices,  $S(a, b) \stackrel{?}{=} c$  est calculable en  $\mathcal{O}(1)$  – on peut simplement vérifier l'équivalence des dimensions intérieures  $(a_2 \stackrel{?}{=} b_1)$ .

La vérification de forme des opérateurs de tableaux multidimensionnels n'est pas toujours décidable. Pour les fonctions de forme arbitraires  $S(a, b)$ , la vérification  $S(a, b) \stackrel{?}{=} c$  nécessite une machine de Turing. Si  $S$  utilise l'opérateur de multiplication, comme dans le cas de l'arithmétique convolutionnelle [Dumoulin and Visin, 2016], l'inférence de forme devient équivalente à l'arithmétique Peano, qui est indécidable [?]. L'addition, la soustraction, l'indexation et la comparaison d'entiers sont toutes décidables en arithmétique de Presburger [??]. La vérification de l'égalité est trivialement décidable, et peut être mise en œuvre dans la plupart des systèmes de type statique.

L'évaluation d'un  $S$  arbitraire qui utilise la multiplication ou la division (par exemple, l'arithmétique convolutionnelle) nécessite un langage de type dépendant [Xi and Pfenning, 1998, Piñeyro et al., 2019], mais la vérification de l'égalité des formes (e. Le système de types de Java est connu pour être complet à Turing [Grigore, 2017]. Ainsi, l'émulation de types dépendants en Java est théoriquement possible, mais probablement insoluble en raison des limitations pratiques notées par Grigore. De plus, nous pensons que la vérification de forme de l'arithmétique matricielle ordinaire est décidable dans tout système de type librement basé sur le système  $F_{<}$ : [Cardelli et al., 1994]. Nous proposons un système de types pour renforcer la sécurité des formes qui peut être mis en œuvre dans n'importe quel langage avec sous-typage et génériques, comme Java [Naftalin and Wadler, 2007], Kotlin [Tate, 2013], TypeScript [Bierman et al., 2014] ou Rust [Crozet et al., 2019].

### 3.9. Shape safety

Il existe trois grandes stratégies pour traiter les erreurs de forme dans la programmation des tableaux:

Math	Infix	Prefix	Postfix	Operator Type Signature
$A(B)$	<code>a(b)</code>			$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^\pi, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^\tau) \rightarrow (\mathbb{R}^\lambda \rightarrow \mathbb{R}^\pi)$
$A \pm B$	<code>a + b</code> <code>a - b</code>	<code>plus(a, b)</code> <code>minus(a, b)</code>		$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^\pi, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^\tau) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R}^\pi)$
$AB$	<code>a * b</code> <code>a.times(b)</code>	<code>times(a, b)</code>		$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times n}, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^{n \times p}) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R}^{m \times p})$
$\frac{A}{B}$ $AB^{-1}$	<code>a / b</code> <code>a.div(b)</code>	<code>div(a, b)</code>		$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times n}, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^{p \times n}) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R}^{m \times p})$
$\pm A$		<code>-a</code> <code>+a</code>	<code>a.unaryMinus()</code> <code>a.unaryPlus()</code>	$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^\pi) \rightarrow (\mathbb{R}^\tau \rightarrow \mathbb{R}^\pi)$
$\ln(A)$		<code>ln(a)</code> <code>log(a)</code>	<code>a.ln()</code> <code>a.log()</code>	$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times m}) \rightarrow (\mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times m})$
$\log_b A$	<code>a.log(b)</code>	<code>log(a, b)</code>		$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times m}, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^{m \times m}) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R})$
$A^b$	<code>a.pow(b)</code>	<code>pow(a, b)</code>		$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times m}, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R}^{m \times m})$
$\sqrt{a}$ $\sqrt[3]{a}$	<code>a.pow(1.0/2)</code> <code>a.root(3)</code>	<code>a.pow(1.0/2)</code> <code>a.root(3)</code>	<code>a.sqrt()</code> <code>a.cbrt()</code>	$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^{m \times m}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R}^{m \times m})$
$\frac{da}{db}$ $a'(b)$	<code>a.d(b)</code>	<code>grad(a)[b]</code>	<code>d(a) / d(b)</code>	$(a : \mathbb{R}^\tau \rightarrow \mathbb{R}^\pi, b : \mathbb{R}^\lambda \rightarrow \mathbb{R}^\omega) \rightarrow (\mathbb{R}^? \rightarrow \mathbb{R}^{\pi \times \omega})$

**Table 3.1.** Le système de forme de Kotlin $\nabla$  spécifie la forme de sortie pour les expressions tensorielles.

Forme	$\mathbb{R}^? \rightarrow \mathbb{R}$	$\mathbb{R}^? \rightarrow \mathbb{R}^m$	$\mathbb{R}^? \rightarrow \mathbb{R}^{j \times k}$
$\mathbb{R}^? \rightarrow \mathbb{R}$	$\mathbb{R}^? \rightarrow \mathbb{R}$	$\mathbb{R}^? \rightarrow \mathbb{R}^m$	$\mathbb{R}^? \rightarrow \mathbb{R}^{j \times k}$
$\mathbb{R}^? \rightarrow \mathbb{R}^n$	$\mathbb{R}^? \rightarrow \mathbb{R}^n$	$\mathbb{R}^? \rightarrow \mathbb{R}^{m \times n}$	
$\mathbb{R}^? \rightarrow \mathbb{R}^{h \times i}$	$\mathbb{R}^? \rightarrow \mathbb{R}^{h \times i}$		

**Table 3.2.** La forme d'une dérivée du tenseur dépend de la forme de la fonction en cours de différenciation et de la forme de la variable par rapport à laquelle nous nous différencions.

- (1) Cacher l'erreur en remodelant implicitement ou [broadcasting arrays](#).
- (2) Annoncer l'erreur au moment de l'exécution avec un message approprié, par exemple : [InvalidArgumentException](#).
- (3) Ne pas autoriser la compilation de programmes pouvant entraîner une erreur de forme.

La plupart des bibliothèques de programmation de tableaux telles que NumPy [Van Der Walt et al., 2011] ou TensorFlow [Abadi et al., 2016] utilisent la première ou la deuxième stratégie. Dans Kotlin $\nabla$ , nous adoptons la troisième, qui permet à un vérificateur de type incrémental,

comme ceux que l'on trouve généralement dans les EDI modernes, de détecter instantanément quand une opération de matrice est invalide. Prenons l'exemple suivant:

```
1 val vecA = Vec(1.0, 2.0)      // Type inféré : Vec<Int, D2>
2 val vecB = Vec(1.0, 2.0, 3.0) // Type inféré : Vec<Int, D3>
3 val vecC = vecB + vecB
4 val vecD = vecA + vecB // Erreur de compilation: Expected Vec<2>, found Vec<3>
```

Tenter de faire la somme de deux vecteurs dont les formes ne correspondent pas, c'est échouer à la compilation.

```
1 val matA = Mat1x4(1.0, 2.0, 3.0, 4.0) // Type inféré: Mat<Double, D1, D4>
2 val matB = Mat4x1(1.0, 2.0, 3.0, 4.0) // Type inféré: Mat<Double, D4, D1>
3 val matC = matA * matB
4 val matD = matA * matC // Erreur de compilation: Expected <4, *>, found Mat<1, 1>
```

De même, la multiplication de deux matrices dont les dimensions intérieures ne correspondent pas ne permet pas de compiler.

```
1 val matA = Mat2x4(1.0, 2.0, 3.0, 4.0,
2                     5.0, 6.0, 7.0, 8.0)
3 val matB = Mat4x2(1.0, 2.0,
4                     3.0, 4.0,
5                     5.0, 6.0,
6                     7.0, 8.0)
7 val matC: Mat<Double, D2, D2> = a * b // Les types sont facultatifs, mais encouragés
8 val matD = Mat2x1(1.0, 2.0)
9 val matE = matC * matD
10 val matF = Mat3x1(1.0, 2.0, 3.0)
11 val matG = matE * matF // Erreur de compilation: Expected Mat<1, *>, found Mat<3, 1>
```

Il est nécessaire de spécifier les types de paramètres dans une signature de méthode. Les types de retour explicites sont facultatifs mais sont encouragés pour des raisons de lisibilité. S'ils sont omis, le système de types peut souvent les déduire:

```
1 fun someMatFun(m: Mat<Double, D3, D1>): Mat<Double, D3, D3> = ...
2 fun someMatFun(m: Mat<Double, D2, D2>) = ...
```

La sécurité des formes est actuellement prise en charge jusqu'aux tenseurs de rang 2, c'est-à-dire les matrices. Pour effectuer la vérification des dimensions dans notre système de types, nous énumérons d'abord une liste de types littéraux entiers comme une chaîne de sous-types,  $C <: C - 1 <: C - 2 <: \dots <: 1 <: 0$ , où  $C$  est la plus grande dimension de longueur fixe que nous souhaitons représenter, qui peut être spécifiée par l'utilisateur avant la compilation. Cela garantit une complexité linéaire dans l'espace et le temps pour le contrôle des sous-types, avec une limite supérieure constante.

```
 interface Nat<T: D0> { val i: Int }
// Les valeurs Int ont été réifiées pour permettre de les comparer à l'exécution
sealed class D0(open val i: Int = 0) { companion object: D0(), Nat<D0> }
sealed class D1	override val i: Int = 1: D0(i) { companion object: D1(), Nat<D1> }
sealed class D2	override val i: Int = 2: D1(i) { companion object: D2(), Nat<D2> }
sealed class D3	override val i: Int = 3: D2(i) { companion object: D3(), Nat<D3> } //...
sealed class D99	override val i: Int = 99: D98(i) { companion object: D99(), Nat<D99> }
```

1  
2  
3  
4  
5  
6  
7

Ensuite, nous surchargeons l'opérateur d'appel pour émuler l'instanciation d'une collection littérale, en utilisant l'arité pour en déduire la dimensionnalité. Considérons le cas du rang 1 pour l'inférence de longueur sur des littéraux vectoriels:

```
 open class Vec<E, Len: D1> constructor(val contents: List<E>) {
    companion object {
        operator fun <T> invoke(t: T): Vec<T, D1> = Vec(listOf(t))
        operator fun <T> invoke(t0: T, t1: T): Vec<T, D2> = Vec(listOf(t0, t1))
        operator fun <T> invoke(t0: T, t1: T, t2: T): Vec<T, D3> = Vec(listOf(t0, t1, t2))
    }
}
```

1  
2  
3  
4  
5  
6  
7

Enfin, nous surchargeons les opérateurs arithmétiques en utilisant des contraintes de forme génériques. Comme nos entiers de niveau type sont une chaîne de sous-types, nous n'avons besoin de définir qu'un seul opérateur et pouvons compter sur la substitution de Liskov [Liskov, 1987] pour préserver la sécurité des formes pour tous les sous-types.

```
 // <C: D1> acceptera 1 <= C <= 99 via la substitution Liskov
operator fun <E, C: D1, V: Vec<X, C>> V.plus(v: V): V = TODO()
```

1  
2

L'opérateur `+` peut maintenant être utilisé comme tel. Les opérandes incompatibles entraîneront une erreur de frappe:

```
 // Ajout de vecteur à type contrôlé avec inférence de forme
val Y = Vec(0, 0) + Vec(0, 0) // Y: Vec<Float, D2>
val X = Vec(0, 0) + Vec(0, 0, 0) // Erreur de compilation: Vec<Int, D2>, Vec<Int, D3>
```

La construction dynamique de la longueur est également autorisée, bien qu'elle puisse échouer lors de l'exécution. Par exemple, la construction dynamique de la longueur est autorisée, bien qu'elle puisse échouer en cours d'exécution:

```
 val one = Vec(0, 0, 0) + Vec(0, 0, 0) // Fonctionne toujours en toute sécurité
val add = Vec(0, 0, 0) + Vec<Int, D3>(listOf(...)) // Se compile, mais peut échouer à l'
    exécution
val vec = Vec(0, 0, 0) // Type inféré : Vec<3>
val sum = Vec(0, 0) + add // Erreur de compilation: Vec<Int attendu, D2>, Vec<Int trouvé, D3>
```

Les matrices et les tenseurs ont une syntaxe similaire. Par exemple, Kotlin $\nabla$  peut déduire la forme de la multiplication de la matrice, et ne compilera pas si les dimensions internes des arguments sont en désaccord:

```
 open class Mat<X, R: D1, C: D1>(vararg val rows: Vec<X, C>)
fun <X> Mat1x2(d0: X, d1: X): Mat<X, D1, D2> = Mat(Vec(d0, d1))
fun <X> Mat2x1(d0: X, d1: X): Mat<X, D2, D1> = Mat(Vec(d0), Vec(d1))

operator fun <X, Q: D1, R: D1, S: D1> Mat<X, Q, R>.times(m: Mat<X, R, S>): Mat<X, Q, S> =
    Mt( *(rows.indices).map { i -> /* ... */ }.toTypedArray() )

val matM = Mat1x2(0, 0)
val mat0 = matM * matM // Erreur de compilation: Expected Mat<2, *>, found Mat<1, 2>
```

Une technique similaire peut être trouvée dans nalgebra [Crozet et al., 2019], une bibliothèque d'algèbre linéaire à forme vérifiée pour le langage Rust qui utilise également des entiers synthétiques de niveau type. Cette technique trouve son origine dans Haskell, un langage qui prend en charge des formes plus puissantes de calcul au niveau des types, telles que *arithmétique des types* [Kiselyov, 2005]. L'arithmétique des types simplifie la concaténation des tableaux, l'arithmétique convolutionnelle [Dumoulin and Visin, 2016] et d'autres

opérations qui sont actuellement difficiles à exprimer dans Kotlin $\nabla$ , où des fonctions arbitraires de niveau type  $\mathcal{S}(\mathbf{a}, \mathbf{b})$  (ref. § 3.8) peuvent nécessiter l'énumération de fonctions Kotlin allant jusqu'à  $C^{q+r}$  pour effectuer le calcul.

### 3.10. Testing

Kotlin $\nabla$  prétend éliminer certaines erreurs d'exécution, mais comment savoir si la mise en uvre n'est pas incorrecte? Une méthode est connue sous le nom de test basé sur les propriétés (PBT) [Fink and Bishop, 1997] (??), étroitement lié à la notion de test de métamorphose [Chen et al., 1998] (§ 4.1.5). Parmi les mises en uvre notables, citons QuickCheck [Claessen and Hughes, 2000], Hypothesis [MacIver, 2018] et KotlinTest [Samuel and Lopes, 2018], sur lesquels notre suite de tests est basée. PBT utilise des propriétés algébriques pour vérifier le résultat d'un calcul en construisant des expressions sémantiquement équivalentes mais syntaxiquement distinctes. Lorsqu'elles sont évaluées sur les mêmes entrées, elles doivent produire la même réponse, avec une précision numérique. Deux de ces équivalences sont utilisées pour tester Kotlin $\nabla$ :

- (1) **Différenciation analytique** : différencier manuellement des fonctions sélectionnées et comparer le résultat numérique de l'évaluation d'entrées choisies au hasard dans leur domaine avec le résultat numérique obtenu en évaluant la DA sur les mêmes entrées.
- (2) **Approximation par différence finie** : échantillonner l'espace des fonctions symboliques différentiables, en comparant les résultats numériques suggérés par la **méthode par différence finie** et le résultat équivalent de la DA, jusqu'à une approximation de précision fixe.

Par exemple, le test suivant vérifie si la dérivée analytique et la dérivée automatique, lorsqu'elles sont évaluées à des points aléatoires, sont égales avec une précision numérique:

```


val z = y * (sin(x * y) - x)           // Fonction à l'essai
val dz_dx = d(z) / d(x)                 // Dérivé automatique
val manualDx = y * (cos(x * y) * y - 1) // Dérivé manuel

"dz/dx should be y * (cos(x * y) * y - 1)" {
```

1  
2  
3  
4  
5

```

NumericalGenerator.assertAll { x0, y0 -> 
    // Evaluer les résultats à une graine donnée
    val autoEval = dz_dx(x to x0, y to y0)
    val manualEval = manualDx(x to x0, y to y0)
    autoEval shouldBeApproximately manualEval // Fails iff eps < |adEval - manualEval|
}
}

```

6  
7  
8  
9  
10  
11  
12

PBT va rechercher dans l'espace de saisie deux valeurs numériques `x0` et `y0`, qui violent la spécification, puis les "rétrécir" pour découvrir des valeurs limites de type "pass-fail". Nous pouvons construire un test similaire en utilisant la [sec:fdm]méthode des différences finies, par exemple  $f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$ :

```


    val dx = 1E-8
    val f = sin(x)
    val df_dx = d(f) / d(x)
    val fd_dx = (sin(x + dx) - sin(x)) / dx

    "d(sin x)/dx should be equal to (sin(x + dx) - sin(x)) / dx" {
        NumericalGenerator.assertAll { x0 ->
            val autoEval = df_dx(x0)
            val fdEval = fd_dx(x0)
            autoEval shouldBeApproximately fdEval // Fails iff eps < |adEval - fdEval|
        }
    }

```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12

Pour plus de détails sur le PBT, voir [??.](#). Il existe de nombreuses autres façons de vérifier indépendamment le gradient numérique, comme les nombres doubles ou la dérivée d'étape complexe [Martins et al., 2003]. Une autre stratégie consiste à comparer avec un cadre AD bien connu, tel que TensorFlow [Abadi et al., 2016] ou PyTorch [Paszke et al., 2019]. Dans les travaux futurs, nous avons l'intention de procéder à une comparaison plus approfondie de la précision et des performances numériques.

### 3.11. Operator overloading

La surcharge de l'opérateur [Corliss and Griewank, 1993] est l'un des moyens les plus simples de mettre en uvre la différentiation automatique. Nous utilisons la fonctionnalité [surcharge de l'opérateur](#) de Kotlin sur une tour numérique (ref. § 3.13) pour fournir une

notation concise pour les opérations algébriques abstraites. Par exemple, supposons que nous ayons une interface **Group**, qui surcharge les opérateurs `+` et `*`:

```
interface Group<T: Group<T>> {
    operator fun plus(addend: T): T
    operator fun times(multiplicand: T): T
}
```

1  
2  
3  
4

Ici, nous spécifions un type récursif lié en utilisant une méthode connue sous le nom de polymorphisme lié à F [Canning et al., 1989] pour s'assurer que les opérations renvoient la valeur concrète de la variable de type `T`, plutôt que quelque chose de plus abstrait comme **Group** (en fait, `T` est un type `self`). Imaginez une classe **Fun** qui a implémenté **Group**. Elle peut être utilisée comme suit:

```
fun <T: Group<T>> cubed(t: T): T = t * t * t
fun <X: Fun<X>> twiceExprCubed(e: X): X = cubed(e) + cubed(e)
```

1  
2

Comme [Python](#), Kotlin supporte la surcharge d'un ensemble limité d'opérateurs, qui sont évalués en utilisant un [précédent fixe](#). Dans la version actuelle de Kotlin∇, les opérateurs n'effectuent aucun calcul, ils construisent simplement un graphe acyclique dirigé ([Fig. 3.4](#)) représentant l'expression symbolique. Les expressions ne sont évaluées que lorsqu'elles sont invoquées sous forme de fonction.

## 3.12. First-class functions

En soutenant les fonctions d'ordre supérieur et les lambdas, Kotlin traite les fonctions comme des citoyens de première classe. Cela nous permet de représenter des fonctions mathématiques et des fonctions de programmation avec les mêmes abstractions sous-jacentes (c'est-à-dire des FP typés). Suite à un certain nombre d'articles récents sur les fonctions AD [[Pearlmutter and Siskind, 2008a](#), [Wang et al., 2018a](#)], toutes les expressions de Kotlin∇ sont traitées comme des fonctions. Par exemple:

```
fun <T: Group<T>> makePoly(x: Var<T>, y: Var<T>) = x * y + y * y + x * x
val f = makePoly(x, y)
val z = f(1.0, 2.0) // Returns a value
```

1  
2  
3

Actuellement, il est possible de représenter des fonctions où toutes les entrées et sorties partagent un seul type de données. Il peut être possible d'étendre le soutien à la création de fonctions avec différents types d'entrées/sorties et d'appliquer des contraintes sur les deux, en utilisant des limites de type covariantes et contravariantes.

### 3.13. Numeric tower

Kotlin<sup>▽</sup> utilise une tour numérique [St-Amour et al., 2012]. Un premier exemple de ce schéma se trouve dans [Scheme](#) [Sperber et al., 2009]. Cette stratégie est également adaptée aux langages orientés objet [Niculescu, 2003, 2011, Kennedy and Russo, 2005] et appliquée dans des bibliothèques telles que [KMath](#) [Nozik, 2019] et [Apache Commons Math](#) [Developers, 2012].

```
 interface Group<X: Group<X>> {
    operator fun unaryMinus(): X
    operator fun plus(addend: X): X
    operator fun minus(subtrahend: X): X = this + -subtrahend
    operator fun times(multiplicand: X): X
}

interface Field<X: Field<X>> : Group<X> {
    val e: X
    val one: X
    val zero: X
    operator fun div(divisor: X): X = this * divisor.pow(-one)
    infix fun pow(exp: X): X
    fun ln(): X
}
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15

La tour numérique nous permet de définir des comportements communs tels que la soustraction et la division sur des structures algébriques abstraites, par exemple **Group**, **Ring**, et **Field**. Ces abstractions sont extensibles à des systèmes de nombres concrets, tels que les nombres complexes et les quaternions. Par exemple, pour définir plus tard un champ sur des nombres complexes ou des quaternions,<sup>2</sup>, il faut simplement étendre la tour numérique

---

<sup>2</sup>ex. Afin de calculer les dérivées dans un réseau de neurones à quaternions. [Isokawa et al., 2003]

et passer outre l'implémentation par défaut. La plupart des opérations mathématiques peuvent être définies en utilisant un petit ensemble d'opérateurs primitifs, qui peuvent être différenciés de manière générique, plutôt que sur une base ad hoc.

### 3.14. Algebraic data types

Algebraic data types (ADTs) in the form of `sealed classes` (alias types de somme) facilitent une forme limitée de filtrage sur un ensemble fermé de sous-classes. Lors de la comparaison avec les sous-classes d'une classe scellée, le compilateur oblige l'auteur à fournir un flux de contrôle exhaustif sur tous les sous-types concrets d'une classe abstraite. Considérons les classes suivantes:

```
1 class Const<T: Fun<T>>(val number: Number) : Fun<T>()
2 class Sum<T: Fun<T>>(val left: Fun<T>, val right: Fun<T>) : Fun<T>()
3 class Prod<T: Fun<T>>(val left: Fun<T>, val right: Fun<T>) : Fun<T>()
4 class Var<T: Fun<T>> : Fun<T>() { override val variables: Set<Var<X>> = setOf(this) }
5 class Zero<T: Fun<T>> : Const<T>(0.0)
6 class One<T: Fun<T>> : Const<T>(1.0)
```

Lorsqu'on passe au type de classe scellée, les consommateurs doivent explicitement traiter chaque cas, car un flux de contrôle incomplet ne se compilera pas au lieu d'échouer silencieusement au moment de l'exécution. Considérons maintenant une définition simplifiée de `Fun`, une classe scellée qui définit le comportement de l'invocation et de la différenciation des fonctions, en utilisant une forme restreinte de correspondance de motifs. Elle peut être construite avec un ensemble de `Vars`, et peut être invoquée avec une valeur numérique:

```
1 sealed class Fun<X: Fun<X>>(open val variables: Set<Var<X>> = emptySet()): Group<Fun<X>> {
2     constructor(vararg fns: Fun<X>): this(fns.flatMap { it.variables }.toSet())
3     // Les sous-classes de Fun sont un ensemble fermé, aucun else -> ... n'est requis.
4     operator fun invoke(map: Map<Var<X>, X>): Fun<X> = when (this) {
5         is Const -> this
6         is Var -> map.getOrDefault(this) { this } // Une application partielle est autorisée
7         is Prod -> left(map) * right(map) // Le casting intelligent lance implicitement après
8             vérification
9         is Sum -> left(map) + right(map)
10    }
```



```

11
12
13
14
15
16
17
18
19
20
21

fun d(variable: Var<X>): Fun<X> = when(this) {
    is Const -> Zero
    is Var -> if (variable == this) One else Zero
    // Règle du produit: d(u*v)/dx = du/dx * v + u * dv/dx
    is Prod -> left.d(variable) * right + left * right.d(variable)
    is Sum -> left.d(variable) + right.d(variable)
}

operator fun plus(addend: Fun<T>) = Sum(this, addend)
operator fun times(multiplicand: Fun<T>) = Prod(this, multiplicand)
}

```

Le [coulée intelligente](#) de Kotlin est un exemple d'analyse de type sensible au flux [?] où le type abstrait `Fun` peut être traité comme `Sum` après avoir effectué une vérification `is Sum`. Sans le smart casting, il faudrait écrire `(this as Sum).left` pour accéder au membre, `left`, créant une `ClassCastException` potentielle si le casting se trompait.

### 3.15. Multiple dispatch

En conjonction avec les ADT, Kotlin $\nabla$  utilise l'envoi multiple pour instancier le type de résultat le plus spécifique d'une opération arithmétique en fonction du type de ses opérande. Bien que Kotlin ne prenne pas directement en charge la répartition multiple, il peut être émulé en utilisant la répartition unique comme décrit par [Leavens and Millstein \[1998\]](#). En se basant sur § 3.14, supposons que nous souhaitions réécrire une expression algébrique, par exemple pour réduire le gonflement de l'expression ou améliorer la stabilité numérique. Nous pouvons utiliser `when` pour bifurquer sur le type d'une sous-expression au moment de l'exécution:

```

1
2
3
4
5
6
7
8
9
10

override fun times(multiplicand: Fun<X>): Fun<X> =
    when {
        this == zero -> this
        this == one -> multiplicand
        multiplicand == one -> this
        multiplicand == zero -> multiplicand
        this == multiplicand -> pow(two)
        // Sans smart cast: Const((this as Const).number * (multiplicand as Const).number)
        this is Const && multiplicand is Const -> Const(number * multiplicand.number)
        // Simplification supplémentaire est possible grâce aux règles de remplacement
    }

```

```

    else -> Prod(this, multiplicand)
}

val result = Const(2.0) * Sum(Var(2.0), Const(3.0))
//           = Sum(Prod(Const(2.0), Var(2.0)), Const(6.0))

```

11  
12  
13  
14  
15

La répartition multiple nous permet de mettre tous les flux de contrôle connexes sur une seule classe abstraite qui est héritée par des sous-classes, ce qui simplifie la lisibilité, le débogage et le remaniement.

### 3.16. Extension functions

Fonctions d’extension permet d’augmenter les classes externes avec de nouveaux champs et méthodes. En utilisant la programmation orientée vers le contexte [Hirschfeld et al., 2008], nous pouvons exposer des extensions personnalisées (par exemple, par le biais de `DoubleContext`) aux consommateurs sans avoir besoin de sous-classement ou d’héritage.

```

object DoubleContext {
    operator fun Number.times(expr: Fun<Double>) = Const(toDouble()) * expr
}

```

1  
2  
3

Maintenant, nous pouvons utiliser le contexte pour définir une autre extension, `Fun.multiplyByTwo()`, qui calcule le produit à l’intérieur d’un `DoubleContext`, en utilisant la surcharge d’opérateur définie ci-dessus:

```

fun Fun<Double>.multiplyByTwo() = with(DoubleContext) { 2 * this }

```

1

Des extensions peuvent également être définies dans un autre fichier ou contexte et importées à la demande, une approche empruntée à KMath [Nozik, 2019], une autre bibliothèque mathématique pour Kotlin. Cette approche convient également à la définition de méthodes de commodité pour l’affectation de variables et d’adaptateurs de type pour les primitives numériques, en tenant compte du contexte. Par exemple:

```

object DoubleContext: Proto<DConst, Double>() {
    override val Const<DConst, Number>.value: Double
        get() = c.toDouble()
    override fun wrap(default: Number): DConst = DConst(default.toDouble())
}

```

1  
2  
3  
4

```

override val X: X<DConst> = object: X<DConst>(DConst(0.0)) {
    override fun invoke(X: XBnd<DConst>): DConst = X.const
    override fun toString() = "X"
}

override val Y: Y<DConst> = object: Y<DConst>(DConst(0.0)) {
    override fun invoke(Y: YBnd<DConst>): DConst = Y.const
    override fun toString() = "Y"
}

override infix fun X<DConst>.to(c: Double) = XBnd(DConst(c))
override infix fun Y<DConst>.to(c: Double) = YBnd(DConst(c))
}

```

Cette DSL, qui est utilisée pour prendre en charge la capture et le curry de variables, peut être utilisée comme suit:

```

with(DoubleContext) {
    val t = X + Y + 0.0
    val l = t(X to 1.0, Y to 2.0) * t(X to 1.0)(Y to 3.0) // Currying
    val p = t(X to 1.0) // Partial application
    val k = t(Z to 4.0) // Does not compile
}

```

### 3.17. Automatic, symbolic differentiation

S'inspirant de McCarthy [1960], Kotlin $\nabla$  met en uvre une différenciation symbolique, similaire à l'approche trouvée dans Abelson and Sussman [1996, §2.56–2.58]. Les expressions symboliques permettent une meilleure lisibilité, une plus grande précision numérique et une plus grande efficacité de calcul. Motivé par cette observation, nous mettons en uvre des extensions vectorielles et matricielles à la différenciation scalaire telle que décrite par Dwyer et al. [1948] et plus récemment Laue et al. [2018].

La littérature sur la DA affirme depuis longtemps que la différenciation automatique n'est pas une différenciation symbolique [Baydin et al., 2015a]. Beaucoup, y compris l'auteur de cette thèse, ont soupçonné cette affirmation d'être trompeuse. Récemment, cette affirmation a été mise en doute [Wang et al., 2018b] et réfutée [Laue, 2019]. S'il est vrai que certaines implémentations de la différenciation automatique entrelacent l'évaluation numérique

et la différenciation symbolique au moment de l'exécution, cet entrelacement n'est certainement pas une condition préalable pour qu'une bibliothèque de différenciation soit considérée comme *automatique*. De même, comme le suggère la littérature antérieure [Baydin and Pearlmutter, 2014], le problème du gonflement de l'expression n'est pas unique à la différenciation symbolique [Laue, 2019].

La distinction entre AD et SD devient de plus en plus floue lorsque l'on considère les modèles d'exécution plus flexibles [Wang et al., 2018b] et les AD hybrides [Abadi et al., 2016] qui sont capables d'une évaluation à la fois enthousiaste [Paszke et al., 2019, Agrawal et al., 2019] et paresseuse [Neubig et al., 2017, van Merriënboer et al., 2018]. Nous estimons au contraire que la différenciation symbolique est un type de différenciation automatique que la littérature sur la DA a trop vite écarté. SD, en particulier, offre au compilateur beaucoup plus de souplesse pour effectuer des optimisations globales telles que la simplification algébrique [Bergstra et al., 2010], la vectorisation en boucle [Agarwal, 2019] et la compréhension du tenseur [Vasilache et al., 2018, Laue et al., 2020]. Ces optimisations seraient autrement impossibles si leur différenciation symbolique et leur évaluation numérique étaient effectuées en synchronisme, alors que le graphique de flux de données n'est que partiellement disponible.

### 3.18. Coroutines

Les coroutines sont une généralisation des sous-routines pour le multitâche non préventif, généralement mises en uvre à l'aide de continuations [Haynes et al., 1984]. Les continuations sont un mécanisme qui permet aux fonctions d'accéder et de modifier les calculs ultérieurs. Dans le style des continuations [Sussman and Steele, 1975] (CPS), chaque fonction, en plus de ses arguments habituels, prend une autre fonction représentant la routine suivante. Plutôt que de retourner à son interlocuteur après l'achèvement, la fonction invoque sa continuation, et le processus est relancé.

Une forme de continuation, connue sous le nom de "continuations délimitées", suffit pour mettre en uvre la DA en mode inverse avec surcharge de l'opérateur uniquement (sans structures de données supplémentaires) comme décrit par Wang et al. [2018b] et plus tard dans Wang et al. [2018a]. Alors que les rappels dans Kotlin sont par défaut à un seul coup, les continuations délimitées réentrant ou "à plusieurs coups" peuvent également être implémentées à l'aide de [Kotlin Coroutines](#). Les continuations délimitées "multi-shot" simplifieraient

grandement notre mise en uvre de la DA, permettraient un ensemble plus souple de primitives pour la programmation asynchrone et méritent d'être étudiées plus avant.

### 3.19. Comparison

Inspiré par [Stalin \$\nabla\$](#)  [Pearlmutter and Siskind, 2008b], [Autograd](#) [Maclaurin et al., 2015, Maclaurin, 2016], [Theano](#) [Bergstra et al., 2010], [Myia](#) [van Merriënboernboer et al., 2018], [JAutoDiff](#) [Nureki, 2012], [Nexus](#) [Chen, 2017], [Lanterne](#) [Wang et al., 2018b], [Tangent](#) [van Merriënboer et al., 2018], [Elliott](#) [2018], [Halide](#) [Li et al., 2018] et al, [Kotlin \$\nabla\$](#)  tente de transposer les récents développements en matière de différenciation automatique (AD) à la langue Kotlin. Ce faisant, il introduit un certain nombre d'idées expérimentales, notamment [compile-time shape-safety](#), [algebraic simplification](#) et la vérification de la stabilité numérique par [property-based testing](#). Travaux antérieurs, notamment [PyTorch](#) [Paszke et al., 2019], [TensorFlow](#) [Abadi et al., 2016], [Chainer](#) [Tokui et al., 2015], [DL4J Team](#) [2016a] et d'autres ont développé des bibliothèques AD d'usage général dans des langues moins sûres.

Contrairement à la plupart des implémentations existantes, [Kotlin \$\nabla\$](#)  est un AD purement symbolique, basé sur des graphes, qui ne nécessite pas de métaprogrammation de modèles, d'augmentation de la puissance du compilateur ou de réflexion sur l'exécution. Comme nous l'avons vu, cette approche est principalement réalisée par [surcharge d'opérateur](#), polymorphisme paramétrique et [correspondance de modèles](#). L'avantage pratique de cette technique est qu'elle peut être mise en uvre sous la forme d'une simple bibliothèque ou d'un langage intégré spécifique au domaine (eDSL), ce qui permet d'exploiter le système de types du langage hôte pour recevoir gratuitement la complétion de code et l'inférence de type. Notre approche utilise plusieurs idiomes fonctionnels, notamment les expressions lambda, les fonctions d'ordre supérieur, l'application partielle, le curry et les types de données algébriques. Pour une comparaison détaillée de [Kotlin \$\nabla\$](#)  avec les bibliothèques AD existantes, voir [Table 3.3](#).

[Kotlin \$\nabla\$](#)  préconise l'utilisation de la programmation de tableaux fonctionnels et sans danger pour les types, mais n'impose pas ses préférences aux consommateurs. Si l'utilisateur omet la forme, il revient à la vérification de la forme en cours d'exécution. Conformément

Framework	Language	Symbolic Differentiation	Automatic Differentiation	Differentiable Programming	Functional Programming	Type-Safe	Shape-Safe	Dependently-Typed	Multiplatform
Kotlin $\nabla$	Kotlin	✓	✓	✓	✓	✓	✓	✗	⚠
DiffSharp	F#	✗	✓	✓	✓	✓	✗	✗	✗
TensorFlow.FSharp	F#	✗	✓	✓	✓	✓	✓	✗	✗
Nexus	Scala	✗	✓	✓	✓	✓	✓	✗	✗
Lantern	Scala	✗	✓	✓	✓	✓	✗	✗	✗
Tensor Safe	Haskell	✗	✓	✗	✓	✓	✓	✓	✗
Hasktorch	Haskell	✗	✓	✓	✓	✓	✓	✗	✗
Eclipse DL4J	Java	✗	✓	✗	✗	✓	✗	✗	✗
JAutoDiff	Java	✓	✓	✓	✗	✓	✗	✗	✗
Stalin $\nabla$	Scheme	✗	✓	✓	✗	✗	✗	✗	✗
Myia	Python	✓	✓	✓	✓	✗	✗	✗	⚠
JAX	Python	✗	✓	✓	✓	✗	✗	✗	⚠

**Table 3.3.** Comparaison des bibliothèques AD. Bien que nous ne fassions pas de distinction entre AD et SD comme décrit dans § 3.17, nous adoptons ici la nomenclature préférée des auteurs. Nous faisons une distinction entre les bibliothèques de programmation différentiable (§ 3.2) et celles qui construisent simplement des réseaux de neurones. Le symbole ⚡ indique un travail en cours.

à la philosophie du langage hôte, les utilisateurs peuvent utiliser leur style de programmation préféré, en introduisant progressivement des contraintes pour profiter des avantages d'une vérification de type plus forte et se prévaloir de ses caractéristiques de programmation fonctionnelle plus riches.

### 3.20. Travaux futurs

La dérivée, telle que cette notion apparaît dans le calcul différentiel élémentaire, est un exemple mathématique familier d'une fonction pour laquelle le domaine et l'intervalle sont tous deux constitués de fonctions.”

---

—Alonzo Church [1941], *The Calculi of Lambda Conversion*

Le dérivé, tel qu'il est couramment utilisé, est généralement associé au calcul des infinitésimaux. Mais les mêmes règles de différenciation symbolique introduites par Leibniz et Newton il y a plus de trois siècles ont réapparu dans des endroits étranges et merveilleux. Dans Brzozowski [1964], nous rencontrons un exemple de différenciation symbolique dans un cadre discret, c'est-à-dire des expressions régulières. Les travaux de Brzozowski ont des applications importantes et de grande portée dans la théorie des automates [Berry and Sethi, 1986, Caron et al., 2011, Champarnaud et al., 1999] et l'analyse incrémentale [Might et al., 2011, Moss, 2017]. Plus tard, dans Thayse [1981], le calcul différentiel booléen a été introduit pour la première fois,<sup>08em</sup><sup>3</sup> une branche de l'algèbre booléenne qui a des applications importantes dans la théorie de la commutation [Thayse and Davio, 1973] et la synthèse des circuits numériques [Steinbach and Posthoff, 2017]. La différenciation symbolique a des applications utiles dans d'autres contextes mathématiques, notamment  $\lambda$ -calculus [Ehrhard and Regnier, 2003, Cai et al., 2014, Kelly et al., 2016, Brunel et al., 2020], incrémental computation [Alvarez-Picallo et al., 2018, Alvarez-Picallo and Ong, 2019], théorie des types [McBride, 2008, Chen et al., 2012], théorie des catégories [Blute et al., 2009], théorie des domaines [?], théorie des probabilités [?] et logique linéaire [Ehrhard, 2016, ?].

De nombreux autres exemples de différenciation symbolique peuvent être trouvés dans des corps de littérature sans rapport. Ces indices semblent suggérer une connexion non réalisée entre la géométrie différentielle et algébrique, ce qui pourrait apporter des idées importantes pour la programmation différentiable et l'étude de la propagation des changements sur les graphiques de calcul.

Les travaux décrits dans ce chapitre établissent un cadre pour l'exploration de la différenciation symbolique à l'aide de structures algébriques comme **Group**, **Ring**, et **Field** (§ 3.13).

---

<sup>3</sup>les premiers travaux sur le sujet remontent à Talantsev [1959] et ?

Dans nos travaux futurs, nous espérons explorer la relation entre la programmation différentiable et la différentiation symbolique dans d'autres topologies. Il existe peut-être un mécanisme analogue à la descente de gradient qui peut être exploité pour accélérer l'optimisation dans de tels espaces, par exemple pour l'apprentissage des variables booléennes et d'autres structures de données comme les graphiques et les arbres.

Comme le montrent les ouvrages précédents [Bergstra et al., 2010, Baydin et al., 2015a, Laue, 2019], la houle d'expression intermédiaire est un problème pernicieux en algèbre informatique et en différentiation automatique. La procédure de simplification algébrique ad hoc décrite dans § 3.15 est presque certainement inadéquate pour les cas d'utilisation générale. Une direction intéressante serait de former un modèle pour minimiser la dérive numérique, en appliquant des règles de réécriture à usage général. Il existe une longue liste de travaux antérieurs sur les algorithmes de réécriture pour la stabilité numérique, qui remonte à Kahan [1965], Dekker [1971], Ogita et al. [2005] et qui a été explorée plus récemment par Zaremba et al. [2014], Zaremba [2016] et Wang et al. [2019] dans une perspective d'apprentissage machine.

Fournir un type de structure matricielle (par exemple, `Singular`, `Symmetric`, `Orthogonal`) permettrait des spécialisations de la dérivée de matrice (§2.8 of Petersen et al. [2012] pour un examen détaillé des techniques spécifiques de différentiation des matrices structurées). En termes d'amélioration du système de types, Makwana and Krishnaswami [2019] a mis au point un codage de l'algèbre linéaire par types linéaires qui serait également intéressant à explorer.

Du point de vue des performances, la migration vers un backend d'algèbre linéaire dédié tel que ND4J [Team, 2016b], Apache Commons Math [Developers, 2012], EJML [Abeles, 2010] ou JBLAS [Braun et al., 2011] permettrait probablement d'accélérer les choses. À terme, nous prévoyons de compiler vers une représentation intermédiaire dédiée telle que RelayIR [Roesch et al., 2018] ou MLIR [Lattner et al., 2020] afin de recevoir une accélération matérielle sur d'autres plateformes.

### 3.21. Conclusion

Dans ce chapitre, nous avons fait la démonstration de Kotlin $\nabla$ , un langage intégré spécifique à un domaine pour la programmation différentielle et de sa mise en uvre dans le

langage de programmation Kotlin. En utilisant notre DSL comme véhicule, nous avons exploré quelques sujets intéressants en matière de différenciation automatique et de programmation de tableaux de formes sûres. L'auteur souhaite remercier Hanneli Tavante, Alexander Nozik, Erik Meijer, Maxime Chevalier-Boisvert, Kiran Gopinathan, Jacob Miller et Adam Pocock pour leurs précieux commentaires durant le développement de ce projet. Pour plus d'informations sur Kotlin∇, veuillez consulter le site <https://github.com/breandan/kotlingrad>.



# Chapter 4

---

## Testing intelligent systems

Si nous utilisons, pour atteindre nos objectifs, un organisme mécanique dont nous ne pouvons pas interférer efficacement avec le fonctionnement des points, nous devons être sûrs que l'objectif mis dans la machine est celui que nous désirons vraiment.”

---

—Norbert Wiener [1960], *morales et techniques de l'automatisation*

Les réseaux neuronaux profonds actuels sont capables d'apprendre un large éventail de fonctions, mais présentent des faiblesses spécifiques. La formation des réseaux neuronaux qui peuvent être transférés de manière robuste vers de nouveaux domaines où les distributions de formation et de test sont très dissemblables pose un défi important. Ces modèles sont souvent susceptibles d'échouer lorsqu'ils sont présentés avec des entrées soigneusement élaborées. Cependant, les mêmes techniques d'optimisation basées sur les gradients utilisées pour l'entraînement des réseaux neuronaux peuvent également être exploitées pour sonder leurs modes de défaillance.

Dans le domaine du génie logiciel, les techniques de test des logiciels sont de plus en plus automatisées et polyvalentes. Les tests aident à prévenir les comportements régressifs et constituent une forme de spécification dans laquelle le développeur communique le résultat escompté de l'exécution d'un programme. Bien qu'ils soient d'une importance capitale, les tests sont souvent lourds à mettre en uvre. Les techniques récentes de tests automatisés ont permis aux développeurs d'écrire moins de tests avec une couverture plus importante.

Dans ce chapitre, nous proposons un nouvel algorithme de test basé sur les propriétés (PBT) pour les programmes différentiables, et nous montrons que notre méthode améliore empiriquement l'efficacité de l'échantillon par rapport aux tests probabilistes natifs, telle que mesurée par sa capacité à détecter une plus grande proportion d'erreurs violant les contraintes

du test dans un budget donné. Notre algorithme peut être utilisé à la fois pour identifier les limites des régions de confiance, et pour attaquer un modèle préformé étant donné l'accès entrée-sortie et quelques échantillons de la distribution de formation. Nous explorons plus avant la relation entre les méthodes antagonistes dans l'apprentissage machine et le PBT, et montrons comment l'apprentissage antagoniste peut être considéré comme une extension d'une technique PBT connue sous le nom de test métamorphique (MT).

## 4.1. Background

Dans les sections suivantes, nous présentons une série de méthodologies de test de logiciels, par ordre décroissant de complexité cognitive. Nous émettons l'hypothèse que les méthodes suivantes permettent aux développeurs d'atteindre le même niveau d'assurance avec un effort progressivement moindre.

### 4.1.1. Tests unitaires

Dans les tests unitaires traditionnels, chaque sous-programme est accompagné d'un seul test:

```
fun unitTest(subroutine: (Input) -> Output) {  
    val input = Input() // Construct an input  
    val expectedOutput = Output() // Construct an output  
    val actualOutput = subroutine(input)  
    assert(expectedOutput == actualOutput) { "Expected $expectedOutput, got $actualOutput" }  
}
```

1  
2  
3  
4  
5  
6

Les tests unitaires sont efficaces pour valider les convictions d'une personne sur les conditions préalables et postérieures. Le problème, c'est que quelqu'un doit rédiger un tas de cas de tests. Les effets secondaires comprennent une agilité réduite, une aversion pour le remaniement ou le rejet de travaux antérieurs lorsque les tests deviennent obsolètes.

### 4.1.2. Tests d'intégration

Dans les tests d'intégration, nous sommes plus préoccupés par le comportement global d'un programme, plutôt que par le comportement spécifique de ses sous-programmes. Prenons l'exemple suivant:

```


fun <I, O> integrationTest(program: (I) -> O, inputs: Set<I>, checkOutput: (O) -> Boolean) =
    inputs.forEach { input: I ->
        try {
            val output: O = program(input)
            assert(checkOutput(output)) { "Postcondition failed on $input, $output" }
        } catch (exception: Exception) {
            assert(false) { exception }
        }
    }

```

1  
2  
3  
4  
5  
6  
7  
8  
9

Avec cette stratégie, il y a moins de tests à écrire, puisque nous ne nous soucions que du comportement de bout en bout. Les tests d'intégration vérifient simplement un programme pour mettre fin aux exceptions et aux simples conditions de post. Pour cette raison, il est souvent trop grossier.

Pour simplifier, dans les sections suivantes, nous ne considérerons que des exemples de programmes qui sont de pures fonctions, c'est-à-dire qui n'ont pas d'état externe et ne produisent pas d'effets secondaires.

#### 4.1.3. Fuzz testing

Le test Fuzz est une méthodologie de test automatisée qui génère des entrées aléatoires pour tester un programme donné. Prenons par exemple le test suivant:

```


fun <I, O> fuzzTest(program: (I) -> O, oracle: (I) -> O, rand: () -> I) =
    repeat(1000) {
        val input: I = rand()
        assert(program(input) == oracle(input)) { "Oracle and program disagree on $input" }
    }

```

1  
2  
3  
4  
5

Le problème, c'est qu'il nous faut un *oracle*, une hypothèse souvent déraisonnable. C'est ce qu'on appelle le problème de l'oracle. Mais même si nous avions un oracle, puisque l'espace des entrées est souvent grand, il peut falloir beaucoup de temps pour trouver une sortie où ils ne sont pas d'accord. Comme un seul appel à `program(i)` peut être assez coûteux en pratique, cette méthode peut également être assez inefficace.

#### 4.1.4. Property-based testing

Test basé sur la propriété [Fink and Bishop, 1997] (PBT) tente d'atténuer le problème de l'oracle de test en utilisant les *propriétés*. Il se compose de deux phases, la recherche et la réduction. Les utilisateurs spécifient une propriété sur toutes les sorties et le test échoue si un contre-exemple peut être trouvé:

```
1  fun <I, O> gen(program: (I) -> O, property: (O) -> Boolean, rand: () -> I) =  
2    repeat(1000) {  
3      val randomInput: I = rand()  
4  
5      assert(property(program(randomInput))) {  
6        val shrunken = shrink(randomInput, program, property)  
7        "Minimal input counterexample of property: $shrunken"  
8      }  
9    }
```

En gros, `shrink` tente de minimiser le contre-exemple.

```
1  tailrec fun <I, O> shrink(failure: I, program: (I) -> O, property: (O) -> Boolean): I =  
2    if (property(program(decrease(failure)))) failure // Property holds once again  
3    else shrink(decrease(failure), program, property) // Decrease until property holds
```

Par exemple, dans le cas d'un programme `program: (Float) -> Any`, nous pourrions implémenter `decrease` comme ça:

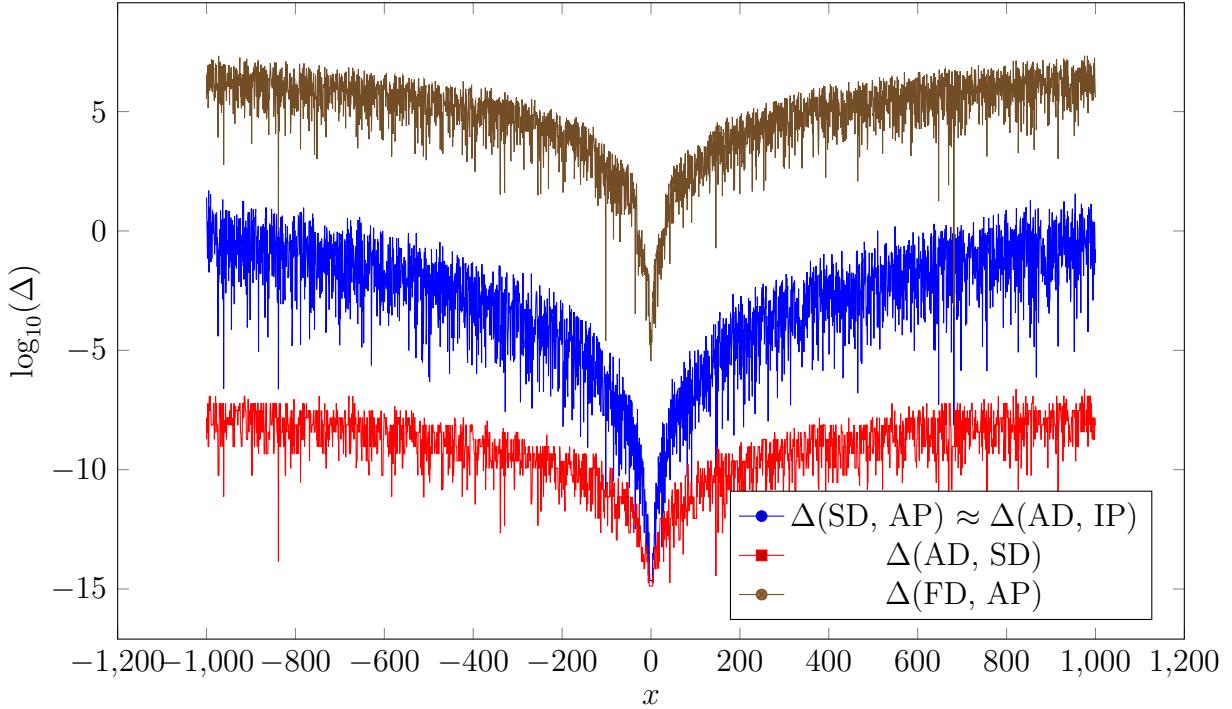
```
1  fun decrease(failure: Float): Float = failure - failure / 2
```

Considérons Fig. 4.3, qui représente la différence logarithmique entre diverses formes de différenciation informatique (évaluées avec une précision standard de 32 bits) et AP (calculée à 30 chiffres significatifs).<sup>1</sup> Étant donné les deux algorithmes de calcul de la dérivée, un test basé sur les propriétés pourrait vérifier si l'erreur est limitée sur toutes les entrées.

Le problème est de savoir quelles sont les bonnes propriétés à tester. Cela demande beaucoup d'efforts et une expertise spécifique au domaine. En outre, l'utilisateur doit spécifier

<sup>1</sup>Pour calculer AP, nous dérivons symboliquement la fonction et l'évaluons numériquement en utilisant approximation par différence définie et l'expansion en série de MacLaurin du sinus et du cosinus à une précision numérique arbitraire.

Erreurs de log entre AD et SD sur  $f(x) = \frac{\sin(\sin(\sin(x)))}{x} + x \sin(x) + \cos(x) + x$



**Fig. 4.1.** Nous comparons la dérive numérique entre AD et SD sur une expression gonflée en utilisant une précision fixe et une précision arbitraire (AP). AD et SD présentent toutes deux des erreurs relatives (c'est-à-dire l'une par rapport à l'autre) inférieures de plusieurs ordres de grandeur à leur erreur absolue. Ces résultats sont conformes aux conclusions de [Laue \[2019\]](#).

un rétrécisseur personnalisé, ce qui n'est pas clair quant à la manière de le mettre en uvre efficacement. Et s'il y avait une meilleure façon de procéder?

#### 4.1.5. Metamorphic testing

Il arrive souvent que l'on veuille tester le comportement d'un programme sans en préciser complètement les propriétés. De nombreux processus générateurs naturels présentent une sorte d'invariance locale: de petites modifications de l'entrée ne changent pas radicalement la sortie. Nous pouvons exploiter cette propriété pour concevoir des méthodes de floutage à usage général en ne tenant compte que de quelques entrées et sorties. Le test métamorphique (MT) est une approche de test basée sur la propriété qui aborde le problème de l'oracle de test et le défi de découvrir à moindre coût des bogues dans le régime de données basses. Elle

a été appliquée avec succès dans les tests de voitures sans conducteur [Zhou and Sun, 2019, Pei et al., 2017, Tian et al., 2018] et d'autres systèmes d'apprentissage profond avec état [Du et al., 2018].

Examinons tout d'abord l'exemple concret suivant, emprunté à ?: supposons que nous ayons mis en uvre un programme qui prend une image d'un véhicule pendant la conduite, et prédit l'angle de braquage simultané du véhicule. Étant donné une seule image et l'angle de braquage correspondant de l'oracle (par exemple un conducteur humain ou un simulateur), notre programme devrait préserver l'invariance sous diverses transformations d'image, telles que des changements d'éclairage limités, des transformations linéaires ou un bruit additif en dessous d'un certain seuil. Intuitivement, l'angle de braquage doit rester approximativement constant, indépendamment de toute transformation ou séquence de transformations appliquée à l'image originale qui satisfont aux critères que nous avons choisis. Si ce n'est pas le cas, cela indique clairement que notre programme n'est pas suffisamment robuste et pourrait ne pas bien répondre au type de variabilité qu'il pourrait rencontrer dans un cadre opérationnel.

Les tests de métamorphose peuvent être exprimés comme suit: Étant donné un oracle  $\mathbf{P} : \mathcal{I} \rightarrow \mathcal{O}$ , et un ensemble d'entrées  $\mathbf{X} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(z)}\}$  et de sorties  $\mathbf{Y} = \{\mathbf{y}^{(1)} = \mathbf{P}(\mathbf{x}^{(1)}), \dots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})\}$ , une relation métamorphique (MR) est une relation  $\mathcal{R} \subset \mathcal{I}^z \times \mathcal{O}^z$  où  $z \geq 2$ . Dans le cas le plus simple, une MR est une relation d'équivalence  $\mathcal{R}$ , c'est-à-dire:  $\langle \mathbf{x}, \mathbf{y}, \mathbf{x}', \mathbf{y}' \rangle \in \mathcal{R} \Leftrightarrow \mathbf{x} \sim_{\mathcal{R}} \mathbf{x}' \text{ gauche - droite } \mathbf{P}(\mathbf{x}) \approx \mathbf{P}(\mathbf{x}')$ .

Supposons que notre RM soit de  $\forall \varphi \in \mathcal{I} : ||\varphi|| \leq \varepsilon, \mathbf{P}(\mathbf{x}) \approx \mathbf{P}(\mathbf{x}' = \mathbf{x} + \varphi) \approx \mathbf{y}$ . Étant donné un programme  $\widehat{\mathbf{P}}$  et un ensemble relativement petit d'entrées  $\mathbf{X}$  et de sorties  $\mathbf{Y}$  de notre oracle  $\mathbf{P}$ , le MR produit un ensemble  $\mathbf{X}'$ ,  $|\mathbf{X}| \ll |\mathbf{X}'|$  sur lequel on peut tester  $\widehat{\mathbf{P}}$ , sans exiger de sorties correspondantes de  $\mathbf{P}$ . Si nous pouvons montrer que  $\exists \mathbf{x}' \in \mathbf{X}' \mid \widehat{\mathbf{P}}(\mathbf{x}') \not\approx \mathbf{P}(\mathbf{x})$ , cela implique au moins l'un des éléments suivants:

- (1)  $\langle \mathbf{x}, \mathbf{P}(\mathbf{x}), \mathbf{x}', \mathbf{P}(\mathbf{x}') \rangle \notin \mathcal{R}$ , c'est-à-dire que nos hypothèses étaient invalides
- (2)  $\widehat{\mathbf{P}}(\mathbf{x}') \not\approx \mathbf{P}(\mathbf{x}')$ , c'est-à-dire que le programme testé n'est pas sain

Dans les deux cas, nous avons obtenu des informations utiles. Si nos hypothèses n'étaient pas valables, nous pouvons renforcer l'invariant,  $\mathcal{R}$ , en supprimant le contre-exemple. Dans le cas contraire, nous avons détecté une erreur et pouvons ajuster le programme pour assurer la conformité - ces deux résultats sont utiles.

Considérons l'exemple suivant d'une MT qui utilise un MR basé sur l'équivalence:

```
1  fun <I, O> mrTest(program: (I) -> O, mr: (I, O, I, O) -> Boolean, rand: () -> Pair<I, O>) =  
2    repeat(1000) {  
3      val (input: I, output: O) = rand()  
4      val tx: (I) -> I = genTX(program, mr, input, output)  
5      val txInput: I = tx(input)  
6      val txOutput: O = program(txInput)  
7      assert(mr(input, output, txInput, txOutput)) {  
8        "<$input, $output> not related to <$txInput, $txOutput> by $mr ($tx)"  
9      }  
10    }
```

Le problème est que la génération de transformations valables est un exercice non trivial. Nous pourrions essayer de générer des transformations aléatoires jusqu'à ce que nous en trouvions une qui réponde à nos critères:

```
1  fun <I, O> genTX(program: (I) -> O, mr: (I, O, I, O) -> Boolean, i: I, o: O): (I) -> I {  
2    while (true) {  
3      val tx: (I) -> I = sampleRandomTX()  
4      val txInput: I = tx(i)  
5      val txOutput: O = program(txInput)  
6      if (mr(i, o, txInput, txOutput)) return tx  
7    }  
8  }
```

Mais cela serait très inefficace et, selon le type d'entrée et de sortie, n'est pas garanti de se terminer. Nous pourrions procéder à une transformation artisanale, mais cela nécessite une connaissance approfondie du domaine. Nous devrions plutôt chercher une méthode plus efficace sur le plan du calcul, fondée sur des principes, et plus générale, pour muter une entrée dans notre ensemble de données afin de découvrir des sorties non valides.

#### 4.1.6. Adversarial testing

Cela nous conduit à des tests contradictoires. Dans le cas général, on nous donne une paire entrée-sortie d'un oracle et un programme se rapprochant de l'oracle, mais pas nécessairement l'oracle lui-même. Notre objectif est de trouver une petite modification de l'entrée d'une fonction, qui produit la plus grande modification de sa sortie, par rapport à la sortie originale.

Imaginez une fonction  $\widehat{\mathbf{P}} : \mathbb{R}^m \rightarrow \mathbb{R}$ , chaque composante  $g_1, \dots, g_m$  que nous cherchons à modifier d'un montant fixe afin de produire directement la plus grande valeur de sortie  $\widehat{\mathbf{P}}(g'_1, \dots, g'_m)$ . Supposons que pour chaque paramètre d'entrée  $g_1, \dots, g_m$ , nous ayons l'un des trois choix suivants: soit nous augmentons la valeur de  $c$ , soit nous la diminuons de  $c$ , soit nous la laissons inchangée. On ne nous donne pas d'autres informations sur  $\widehat{\mathbf{P}}$ . Considérons la solution naïve, qui essaie toutes les combinaisons de perturbations variables et sélectionne l'entrée correspondant à la plus grande valeur de sortie:

---

**Algorithm 1** Brute Force Adversary

---

```

1: procedure BFADVERSARY( $\widehat{\mathbf{P}} : \mathbb{R}^m \rightarrow \mathbb{R}$ ,  $c : \mathbb{R}$ ,  $g_1 : \mathbb{R}$ ,  $g_2 : \mathbb{R}$ ,  $\dots$ ,  $g_m : \mathbb{R}$ ):  $\mathbb{R}^m$ 
2:   if  $m = 1$  then            $\triangleright$  Evaluate  $\widehat{\mathbf{P}}$  and return the best variable perturbation
3:     return argmax $\{\widehat{\mathbf{P}}(g_1 + c), \widehat{\mathbf{P}}(g_1 - c), \widehat{\mathbf{P}}(g_1)\}$ 
4:   else                    $\triangleright$  Appliquer partiellement la perturbation et recurse
5:     return argmax $\{\widehat{\mathbf{P}}(g_1 + c) \circ \text{BFADVERSARY}(\widehat{\mathbf{P}}(g_1 + c), c, g_2, \dots, g_m),$ 
                     $\widehat{\mathbf{P}}(g_1 - c) \circ \text{BFADVERSARY}(\widehat{\mathbf{P}}(g_1 - c), c, g_2, \dots, g_m),$ 
                     $\widehat{\mathbf{P}}(g_1) \circ \text{BFADVERSARY}(\widehat{\mathbf{P}}(g_1), c, g_2, \dots, g_m)\}$ 
6:   end if
7: end procedure

```

---

Comme on peut le voir, **Algorithm 1** est  $\mathcal{O}(3^m)$  par rapport à  $\dim \mathbf{g}$  – ce n'est pas une routine de recherche très efficace, surtout si l'on veut considérer un ensemble de perturbations plus important. Il est clair que si nous voulons trouver la meilleure direction pour mettre à jour  $\mathbf{g}$ , nous devons être plus prudents lors de la recherche.

Même si nous ne pouvons pas calculer une forme fermée pour  $\nabla_{\mathbf{g}} \widehat{\mathbf{P}}$ , si  $\widehat{\mathbf{P}}$  est différentiable presque partout, nous pouvons toujours utiliser la différenciation numérique pour approximer les valeurs ponctuelles du gradient. Considérons **Algorithm 2**, qui utilise la [sec:fdm]méthode de la différence finie pour approcher  $\nabla_{\mathbf{g}} \widehat{\mathbf{P}}$ . Cela nous indique comment modifier au minimum l'entrée pour produire la plus grande sortie à portée, sans avoir besoin de vérifier exhaustivement chaque perturbation comme dans **Algorithm 1**.

---

**Algorithm 2** Finite Difference Adversary

---

```

1: procedure FDADVERSARY( $\widehat{\mathbf{P}} : \mathbb{R}^m \rightarrow \mathbb{R}$ ,  $c : \mathbb{R}$ ,  $g_1 : \mathbb{R}$ ,  $g_2 : \mathbb{R}$ ,  $\dots$ ,  $g_m : \mathbb{R}$ ):  $\mathbb{R}^m$ 
2:   if  $m = 1$  then       $\triangleright$  Calculez la différence finie (centrée) et effectuez la montée du
    gradient
3:     return  $g_1 + \frac{\widehat{\mathbf{P}}(g_1 - c) - \widehat{\mathbf{P}}(g_1 + c)}{2c}$ 
4:   else  $\triangleright$  Appliquer un gradient de montée en une seule étape en utilisant la différence
    finie par composante
5:     return  $g_1 + \frac{\widehat{\mathbf{P}}(g_1 - c, 0, \dots) - \widehat{\mathbf{P}}(g_1 + c, 0, \dots)}{2c}$ , FDADVERSARY( $\widehat{\mathbf{P}}$ ,  $c, g_2, \dots, g_m$ )
6:   end if
7: end procedure

```

---

Nous avons maintenant une procédure qui est  $\mathcal{O}(m)$  par rapport à  $\widehat{\mathbf{P}}$ , mais qui doit être recalculée pour chaque entrée – nous pouvons encore faire mieux en supposant une structure supplémentaire sur  $\widehat{\mathbf{P}}$ . En outre, nous n'avons pas encore intégré de contrainte sur les valeurs d'entrée. Nous pouvons peut-être combiner la notion de test de métamorphose vue dans § 4.1.5 avec l'optimisation sous contrainte pour accélérer la recherche d'exemples contradictoires.

Lors de la rétropropagation, nous effectuons une descente de gradient sur une fonction différentiable en fonction de ses paramètres pour un ensemble spécifique d'entrées. Dans les tests contradictoires basés sur des gradients, nous effectuons une montée de gradient sur une fonction de perte par rapport aux entrées en utilisant un paramétrage fixe. Supposons que nous ayons une fonction vectorielle différentiable  $\widehat{\mathbf{P}} : \mathbb{R}^m \rightarrow \mathbb{R}^n$ , définie comme suit:

$$\widehat{\mathbf{P}}_k(\mathbf{x}; \Theta) = \begin{cases} \widehat{\mathbf{p}}_1(\Theta_1) \circ \mathbf{x} & \text{if } k = 1 \\ \widehat{\mathbf{p}}_k(\Theta_k) \circ \widehat{\mathbf{P}}_k(\Theta_{[1,k]}) \circ \mathbf{x} & \text{if } k > 1 \end{cases} \quad (\text{Eq. 3.1.12 revisited})$$

Dans l'apprentissage profond, les paires données  $\mathbf{X} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(z)}\}$ ,  $\mathbf{Y} = \{\mathbf{y}^{(1)} = \mathbf{P}(\mathbf{x}^{(1)}), \dots, \mathbf{y}^{(z)} = \mathbf{P}(\mathbf{x}^{(z)})\}$  nous voulons trouver  $\Theta^* = \arg \min_{\Theta} \mathcal{L}(\widehat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \Theta), \mathbf{y}^{(i)})$  qui est généralement obtenu en effectuant une descente stochastique du gradient sur la perte par rapport aux paramètres du modèle:

$$\Theta \leftarrow \Theta - \alpha \frac{1}{z} \nabla_{\Theta} \sum_{i=1}^z \mathcal{L}(\widehat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \Theta), \mathbf{y}^{(i)}) \quad (\text{Eq. 3.1.13 revisité})$$

Nous pouvons résoudre le gradient par rapport à  $\Theta$  en multipliant les Jacobiens (Eq. 3.1.7),  $\mathcal{J}_{\mathbf{p}_1} \cdots \mathcal{J}_{\mathbf{p}_k}$ . Dans l'apprentissage contradictoire de la boîte blanche, on nous donne une note fixe  $\Theta$ <sup>2</sup> et contrôlent la valeur de  $\mathbf{x}$ , de sorte que nous pouvons réécrire  $\widehat{\mathbf{P}}_k(\mathbf{x}^{(i)}; \Theta)$  à la place comme  $\widehat{\mathbf{P}}(\mathbf{x})$ , et prendre le gradient directement par rapport à  $\mathbf{x}$ . Notre objectif est de trouver le "pire"  $\mathbf{x}$  à une faible distance de n'importe quel  $\mathbf{x}^{(i)}$ , c'est-à-dire là où  $\mathbf{P}(\mathbf{x})$  ressemble le moins à  $\widehat{\mathbf{P}}(\mathbf{x})$ . Plus concrètement, cela peut être exprimé comme suit

$$\mathbf{x}^* = \arg \max_{\mathbf{x}} \mathcal{L}(\widehat{\mathbf{P}}(\mathbf{x}), \mathbf{y}^{(i)}) \text{ subject to } CS = \{\mathbf{x} \in \mathbb{R}^m \text{ s.t. } \|\mathbf{x}^{(i)} - \mathbf{x}\|_p < \epsilon\} \quad (4.1.1)$$

Pour ce faire, nous pouvons initialiser  $\mathbf{x} \sim U[CS]$  et effectuer une montée en gradient projetée sur la perte:

$$\mathbf{x} \leftarrow \Phi_{CS} \left( \mathbf{x} + \alpha \nabla_{\mathbf{x}} \mathcal{L}(\widehat{\mathbf{P}}(\mathbf{x}), \mathbf{y}^{(i)}) \right) \Phi_{CS}(\phi') = \arg \min_{\phi \in CS} \frac{1}{2} \|\phi - \phi'\|_2^2 \quad (4.1.2)$$

En supposant une connaissance nulle de l'implémentation du programme  $\widehat{\mathbf{P}}$  ou de la distribution des données,  $D_{\widehat{\mathbf{P}}}$ , nous ne pouvons pas faire mieux que la recherche aléatoire [Wolpert and Macready, 1997]. En supposant que  $\widehat{\mathbf{P}}$  est différentiable, étant donné les valeurs d'entrée-sortie, nous pouvons utiliser des techniques d'optimisation d'ordre zéro pour approcher  $\nabla_{\mathbf{x}} \mathcal{L}$ . En supposant que  $\widehat{\mathbf{P}}$  est une source ouverte, nous pourrions utiliser le fuzzing guidé par la couverture pour donner la priorité à la recherche d'entrées plus susceptibles de violer  $\mathbf{T}$ . Si  $\widehat{\mathbf{P}}$  est à la fois open source et différentiable, nous pouvons accélérer la recherche en utilisant la différentiation automatique. Compte tenu des informations supplémentaires sur la distribution de la formation, nous pourrions initialiser la recherche dans des régions invisibles de l'espace d'entrée, par exemple un échantillon de la distribution inverse  $\mathbf{x} \sim \frac{1}{D_{\widehat{\mathbf{P}}}}$ , probablement plus susceptible de provoquer une erreur. Mais tout cela nécessite une grande expertise humaine pour être mis en œuvre efficacement. Et s'il était possible de générer un adversaire au lieu d'en construire un manuellement?

#### 4.1.7. Generative adversarial testing

Quelles sont les propriétés d'un bon adversaire? Pour qu'un adversaire soit considéré comme un bon adversaire, une fraction importante de ses attaques doit enfreindre la spécification du programme. Pour générer des cas de test plausibles, elle doit non seulement être

---

<sup>2</sup> Contrairement à la rétropropagation, où les paramètres  $\Theta$  sont mis à jour.

capable d'exploiter les faiblesses du programme, mais aussi, idéalement, posséder une bonne compréhension de  $p_{data}$ .

Supposons que nous ayons un programme  $D : \mathbb{R}^h \rightarrow \mathbb{B}$ , c'est-à-dire un classificateur binaire. Comment devrions-nous attaquer sa mise en œuvre, sans adversaire habituel, ou définir une distribution préalable sur les entrées? Une solution, connue sous le nom de réseau générateur d'adversaires [Goodfellow et al., 2014] (GAN), propose de former un adversaire "générateur"  $G : \mathbb{R}^v \rightarrow \mathbb{R}^h$  à côté du modèle formé. L'objectif du GAN vanille peut être exprimé comme un problème d'optimisation minimax:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{data}} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}} [\log (1 - D(G(\mathbf{z}^{(i)})))] \quad (4.1.3)$$

Cet objectif peut être atteint en échantillonnant des minibatchs  $\mathbf{x} \sim p_{data}$  et  $\mathbf{z} \sim p_G$ , puis en mettant à jour les paramètres de  $G$  et  $D$  en utilisant leurs gradients stochastiques respectifs:

$$\Theta_D \leftarrow \Theta_D + \nabla_{\Theta_D} \frac{1}{m} \sum_{i=1}^m [\log D(\mathbf{x}^{(i)}) + \log (1 - D(G(\mathbf{z}^{(i)})))] \quad (4.1.4)$$

$$\Theta_G \leftarrow \Theta_G - \nabla_{\Theta_G} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(\mathbf{z}^{(i)}))) \quad (4.1.5)$$

Albuquerque et al. [2019] propose une version augmentée de ce jeu utilisant plusieurs discriminateurs qui reçoivent chacun une projection fixe et aléatoire  $P_k(\cdot)$  de la sortie du générateur, et résout le problème d'optimisation multi-objectifs suivant:

$$\min \mathcal{L}_G(\mathbf{x}) = [l_1(\mathbf{z}), l_2(\mathbf{z}), \dots, l_K(\mathbf{z})], \text{ where } l_k = -\mathbb{E}_{\mathbf{z} \sim p_z} \log D_k(P_k(G(z_k))) \quad (4.1.6)$$

Ce problème peut être résolu en combinant les pertes au moyen d'une forme de maximisation de l'hypervolume:

$$\nabla_{\Theta} \mathcal{L}_G = \sum_{k=1}^K \frac{1}{\eta - l_k} \nabla_{\Theta} l_k \quad (4.1.7)$$

Où  $\eta$  est une limite supérieure commune et fixe pour chaque  $l_k$ . D'autres variantes du GAN, telles que WGAN [Arjovsky et al., 2017], MHGAN [Turner et al., 2019], et al. ont proposé des extensions du GAN vanille pour améliorer la stabilité et la diversité des échantillons. Les GAN ont été appliqués avec succès dans divers domaines allant de la parole [Donahue et al., 2019] à la synthèse de graphes [Wang et al., 2018c]. Une extension

pratique de ce dernier pourrait consister à appliquer le cadre des GAN à la synthèse de programmes et à l'optimisation des compilateurs en choisissant une métrique appropriée et en suivant l'approche proposée par exemple par Adams et al. [2019], Mendis et al. [2019].

Le problème avec les GAN est que nous devons former  $G$  et  $D$  en synchronisation, sinon on devient vite trop fort. Que se passe-t-il si nous voulons attaquer un modèle préformé?

## 4.2. Probabilistic adversarial testing

Désormais, nous appellerons  $\mathcal{L}(\widehat{\mathbf{P}}(\mathbf{x}), \mathbf{y})$  comme  $\mathcal{L}(\mathbf{x})$ . Imaginez un seul test  $\mathbf{T} : \mathbb{R}^m \rightarrow \mathbb{B}$ :

$$\mathbf{T}(\mathbf{x}) = \mathcal{L}(\mathbf{x}) < C \quad (4.2.1)$$

Notre objectif est de trouver un ensemble d'entrées qui satisfont à notre test, compte tenu d'un budget de calcul  $\mathcal{B}_e$  (c'est-à-dire un nombre fixe d'évaluations de programmes) et d'un budget d'étiquetage  $\mathcal{B}_l$  (c'est-à-dire un nombre fixe d'étiquettes).

$$\{D_{\mathbf{T}} : \mathbf{x} \in CS \mid \mathcal{L}(\mathbf{x}) < C\}, \text{ maximize } |D_{\mathbf{T}}| \text{ subject to } \mathcal{B}_e, \mathcal{B}_l \quad (4.2.2)$$

Considérons une extension des méthodes classiques de fuzzing à des fonctions différenciables sur des variables aléatoires continues. Tout d'abord, nous échantillonnons une entrée  $\mathbf{x}_j : \mathbb{R}^m \sim \mathcal{S}_m$  (par ex. uniformément). Si  $\mathcal{L}(\mathbf{x}_j)$  satisfait ??, on monte la perte suivant  $\nabla_{\mathbf{x}} \mathcal{L}$ , sinon on descend et on répète jusqu'à ce que le test "bascule", le gradient disparaît, ou qu'un nombre fixe de pas  $I_{max}$  soit atteint avant de rééchantillonner  $\mathbf{x}_{j+1}$  à partir de  $\mathcal{S}_m$ . Cette procédure est décrite dans [Algorithm 3](#).

Nous émettons l'hypothèse que si la mise en uvre de  $\widehat{\mathbf{P}}$  était imparfaite et qu'il existait un contre-exemple à [Eq. 4.2.1](#), à mesure que la taille de l'échantillon augmenterait, un sous-ensemble de trajectoires ne convergerait pas du tout, un sous-ensemble convergerait vers l'optima local, et les trajectoires restantes découvriraient la limite.

Nous évaluons notre algorithme dans le cadre de la régression, où  $\widehat{\mathbf{P}}$  est un régresseur polynomial (cf. ??) et  $\mathcal{L}$  est la perte d'erreur quadratique moyenne.

Notre ensemble d'apprentissage est constitué de paires d'entrées-sorties provenant d'un ensemble d'expressions algébriques aléatoires. Ces expressions sont produites en générant des arbres binaires parfaits de profondeur 5, dont les nuds de feuilles contiennent avec une probabilité égale soit (1) une variable alphabétique, soit (2) un nombre aléatoire IEEE 754

---

**Algorithm 3** Probabilistic Generator

---

```
1: procedure PROBGEN( $\mathcal{L} : \mathbb{R}^m \rightarrow \mathbb{R}$ ,  $\mathcal{S}_m$ ,  $\mathbf{T} : \mathbb{R}^m \rightarrow \mathbb{B}$ ,  $\mathcal{B}_e : \mathbb{R}$ )
2:    $D_{\mathbf{T}} \leftarrow \{\}$ ,  $j \leftarrow 0$ 
3:   while  $0 < \mathcal{B}_e$  do                                 $\triangleright$  Iterate until computational budget exhausted
4:      $\mathbf{x}_j \sim \mathcal{S}_m$                                  $\triangleright$  Sample from  $\mathcal{S}_m$ 
5:     if  $\mathbf{T}(\mathbf{x}_j, C)$  then                       $\triangleright$  Inside feasible set, perform gradient ascent
6:        $D_{\mathbf{T}} \leftarrow D_{\mathbf{T}} \cup \text{DIFFSHRINK}(-\mathcal{L}, \mathbf{x}_j, \mathbf{T})$ 
7:     else           $\triangleright$  En dehors de l'ensemble réalisable, effectuer une descente en pente
8:        $D_{\mathbf{T}} \leftarrow D_{\mathbf{T}} \cup \text{DIFFSHRINK}(\mathcal{L}, \mathbf{x}_j, \mathbf{T})$ 
9:     end if
10:     $\mathcal{B}_e \leftarrow \mathcal{B}_e - 1$ 
11:   end while
12:   return  $D_{\mathbf{T}}$ 
13: end procedure
```

---

---

**Algorithm 4** Differential Shrinker

---

```
1: procedure DIFFSHRINK( $\mathcal{L} : \mathbb{R}^m \rightarrow \mathbb{R}$ ,  $\mathbf{x}_1 : \mathbb{R}^m$ ,  $\mathbf{T} : \mathbb{R}^m \rightarrow \mathbb{B}$ )
2:    $i \leftarrow 1, t_i \leftarrow \mathbf{T}(\mathbf{x}_i, C)$            $\triangleright$  Store initial state to detect when test flips.
3:   do
4:      $i \leftarrow i + 1, \mathbf{x}_i \leftarrow \Phi_{CS}(\mathbf{x}_{i-1} - \alpha \nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}_{i-1}))$        $\triangleright$  PGD step (Eq. 4.1.2)
5:     if  $\mathbf{T}(\mathbf{x}_i, C) \neq t_1$  then                   $\triangleright$  Boundary value was found.
6:       return if  $t_1$  then  $\{\mathbf{x}_i\}$  else  $\{\mathbf{x}_{i-1}\}$  end if   $\triangleright$  Always return violation.
7:     end if
8:   while  $i \leq I_{max}$  and  $\epsilon < |\mathcal{L}(\mathbf{x}_i) - \mathcal{L}(\mathbf{x}_{i-1})|$            $\triangleright$  While not converged.
9:   return if  $\neg t_1$  then  $\{\mathbf{x}_{i-1}\}$  else  $\emptyset$      $\triangleright$  Return last iterate or  $\emptyset$  if test passed.
10:  end procedure
```

---

en virgule flottante de 64 bits échantillonné uniformément dans la plage  $[-1, 1]$ . Les nuds internes contiennent avec une probabilité égale un opérateur aléatoire dans l'ensemble  $\{+, \times\}$ . Notre générateur d'expression (Eq. 4.2.3) de type  $G_e : \mathbb{N}^+ \times \mathbb{Z} \rightarrow \mathbb{R}^{[1,26]} \rightarrow \mathbb{R}$  prend une profondeur  $\delta : \mathbb{N}^+$ , une graine aléatoire  $\psi : \mathbb{Z}$ , et renvoie une fonction à valeur scalaire.

$$G_e(\delta, \psi) = \begin{cases} \delta \leq 0 & \begin{cases} \delta \sim_{\psi} \{a, b, \dots, z\} \text{ if } \gamma \sim_{\psi} \{\text{True}, \text{False}\}, \\ \chi \sim_{\psi} U(-1, 1) \text{ otherwise.} \end{cases} \\ \delta > 0 & \begin{cases} G(\delta - 1, \psi + 1) + G(\delta - 1, \psi - 1) \text{ if } \gamma \sim_{\psi} \{\text{True}, \text{False}\}, \\ G(\delta - 1, \psi + 1) \times G(\delta - 1, \psi - 1) \text{ otherwise.} \end{cases} \end{cases} \quad (4.2.3)$$

Une implémentation Kotlin du générateur d’arbres d’expression dans Eq. 4.2.3 est présentée ci-dessous:

```


val sum = { left: SFun<DReal>, right: SFun<DReal> -> left + right }
val mul = { left: SFun<DReal>, right: SFun<DReal> -> left * right }
val operators = listOf(sum, mul)
val variables = ('a'..'z').map { SVar<DReal>(it) }

infix fun SFun<DReal>.wildOp(that: SFun<DReal>) = operators.random(rand)(this, that)

fun randomBiTree(height: Int): SFun<DReal> =
    if (height == 0) (listOf(wrap(rand.nextDouble(-1.0, 1.0))) + variables).random(rand)
    else randomBiTree(height - 1) wildOp randomBiTree(height - 1)

```

Notre ensemble de formation est constitué de paires entrée-sortie produites en liant l’ensemble des variables libres à des valeurs, et en évaluant numériquement l’expression sur des valeurs d’entrée échantillonées à partir de  $[-1, -0.2] \cup [0, 2, 1]$ , puis en redimensionnant toutes les sorties à  $[-1, 1]$  en utilisant la normalisation min-max, c’est-à-dire  $\tilde{G}_e(\delta, \psi) = \frac{G_e(\delta, \psi)}{\max |G_e(\delta, \psi)|_{[-1, 1]}}$ . Chaque expression a un ensemble de validation unique  $x_i \sim [-0.2, 0.2]$ .

Dans Fig. 4.2, nous voyons des pertes de train et de validation pour 200 trajectoires de l’élanc SGD à travers l’espace des paramètres. Pour compenser la différence de magnitude entre l’erreur d’entraînement et de validation, nous normalisons toutes les pertes par leurs valeurs respectives à  $t_0$ . Sur la base de la perte de validation, nous appliquons un arrêt anticipé à environ 50 époques.

---

**Algorithm 5** Attaque de substitution

---

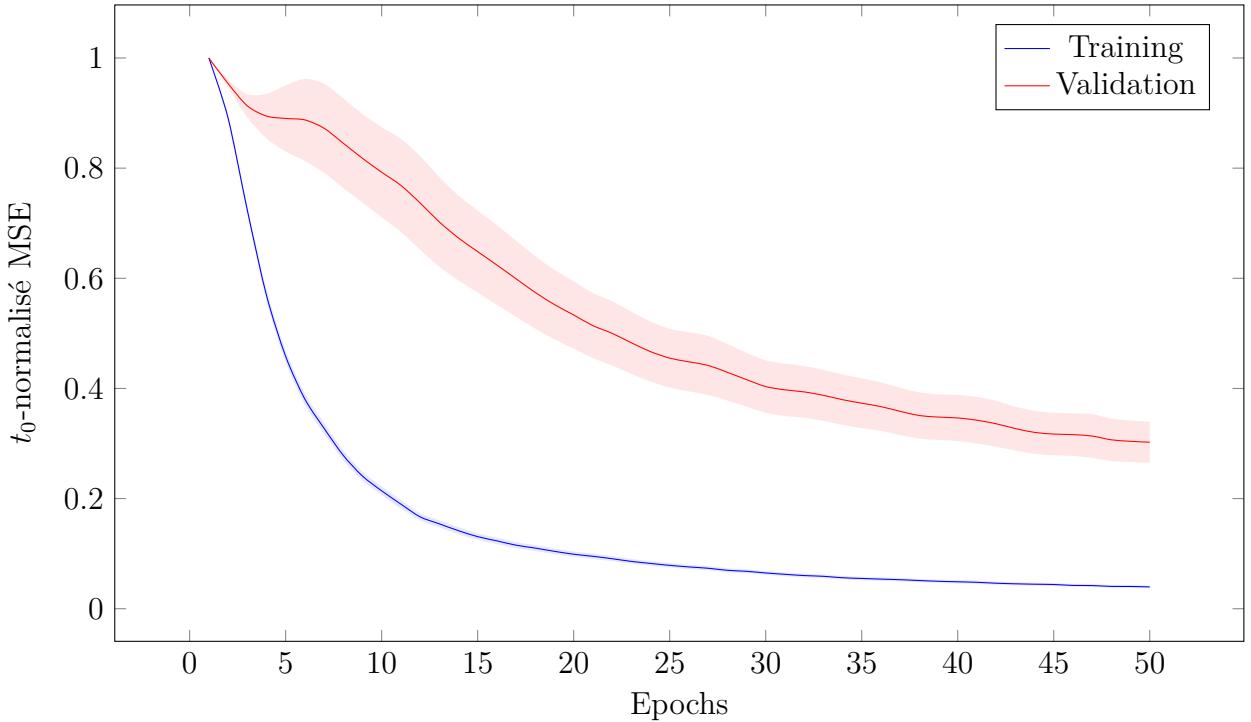
```

1: procedure SURROGATEATTACK( $\Theta : \mathbb{R}^k$ ,  $\hat{\mathbf{f}} : \mathbb{R}^m \times \mathbb{R}^k \rightarrow \mathbb{R}^n$ ,  $\mathcal{S}_m$ ,  $\mathbf{T} : \mathbb{R}^m \rightarrow \mathbb{B}$ ,  $\mathcal{B}_l : \mathbb{R}$ )
2:    $\Theta' \leftarrow \Theta$ 
3:   do
4:      $\mathbf{x} \sim \mathcal{S}_m$ ,  $\mathbf{y} \leftarrow \mathcal{O}(\mathbf{x})$ ,  $\mathcal{B}_l \leftarrow \mathcal{B}_l - 1$             $\triangleright$  Ask for new label from the oracle.
5:      $\Theta' \leftarrow \Theta - \alpha \nabla_{\Theta} \|\hat{\mathbf{f}}(\mathbf{x}; \Theta') - \mathbf{y}\|_2$      $\triangleright$  Mettre à jour les paramètres en utilisant le
         gradient de perte.
6:   while  $0 < \mathcal{B}_l$                                  $\triangleright$  Itérer jusqu'à épuisement du budget d'étiquetage.
7:    $\hat{\mathcal{L}} \leftarrow \|\hat{\mathbf{f}}(\Theta') - \hat{\mathbf{f}}(\Theta)\|_2$            $\triangleright$  Construire la perte de substitution.
8:   return PROBGEN( $\hat{\mathcal{L}}$ ,  $\mathcal{S}_m$ ,  $\mathbf{T}$ ,  $\mathcal{B}_e$ )
9: end procedure

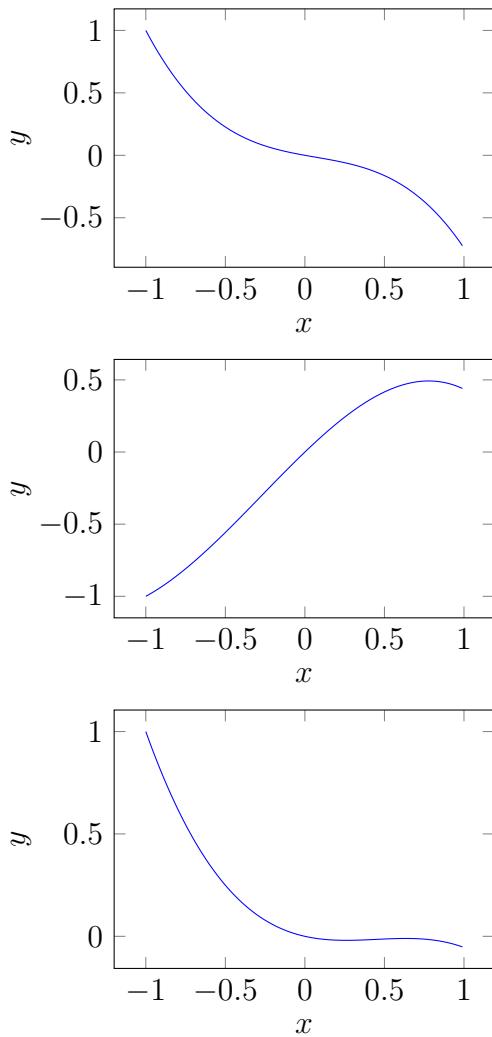
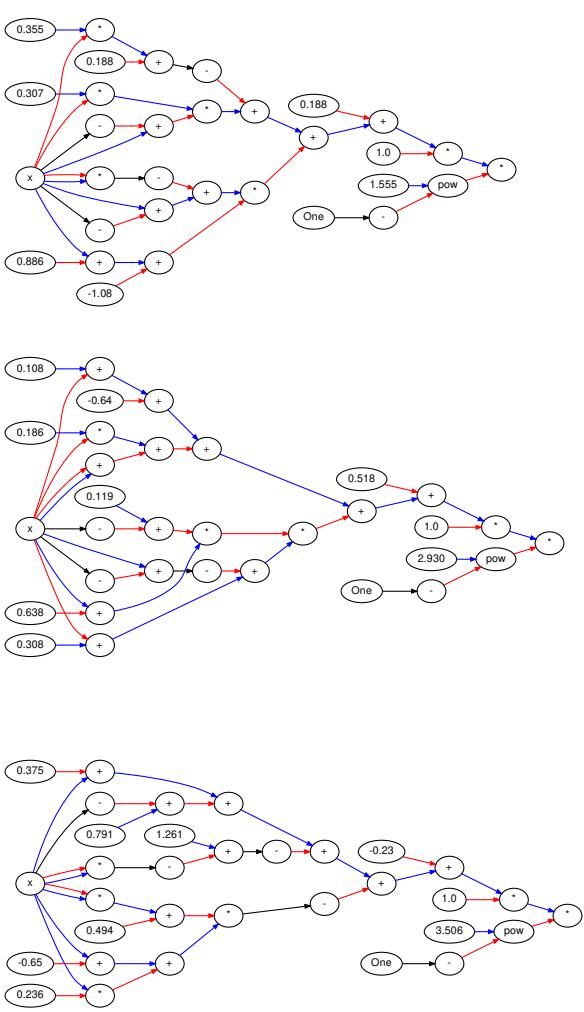
```

---

Courbes de perte moyenne pour la régression polynomiale en utilisant le moment SGD



**Fig. 4.2.** Pour chaque expression de notre ensemble de données, nous formons un régresseur polynomial à la convergence.



**Table 4.1.** DFGs générés par Eq. 4.2.3 avec des tracés en 2D qui les accompagnent.

Ci-dessous, un extrait d'une mise en uvre de la dynamique SGD dans la DSL Kotlin∇:

```

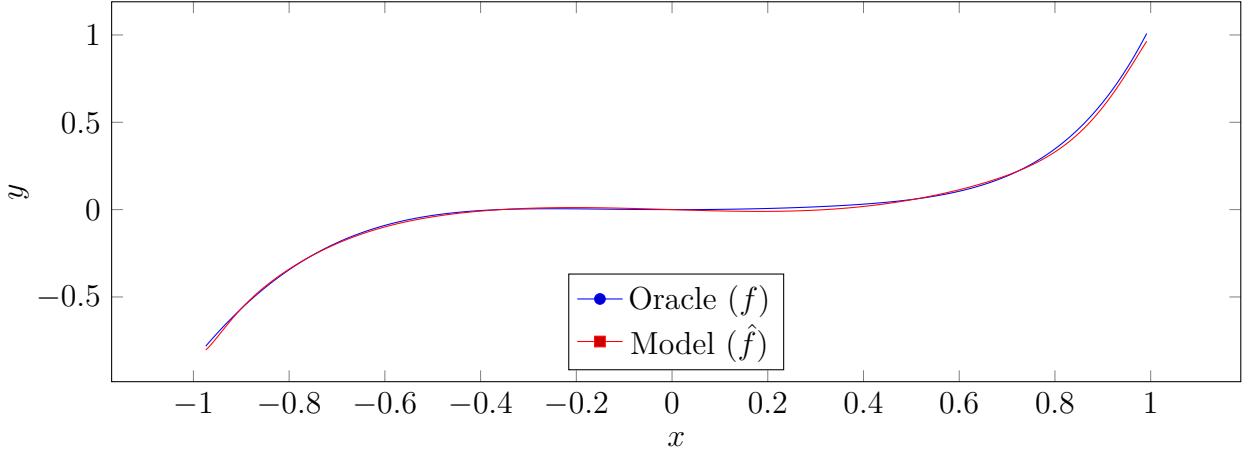

val model = Vec(D30) { x pow (it + 1) } dot weights
var update = Vec(D30) { 0.0 }

batches.forEach { i, batch ->
    val batchInputs = array0f(xBatchIn to batch.first, label to batch.second)
    val batchLoss = (model - label).magnitude()(*batchInputs)
    val weightGrads = (d(batchLoss) / d(weights))(*newWeights)
    update = beta * update + (1 - beta) * weightGrads
    newWeights = newWeights - alpha * update
}

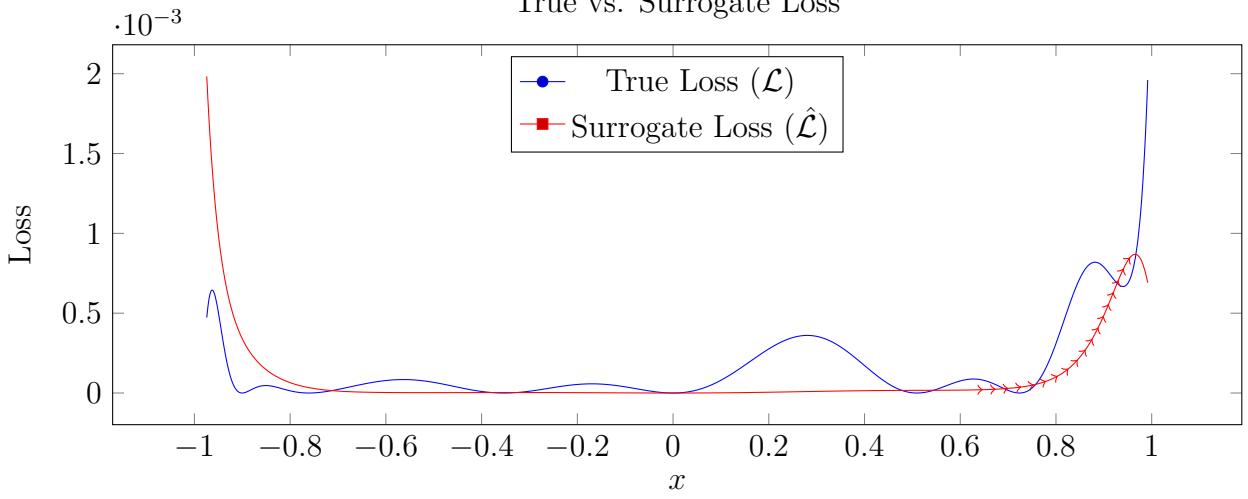
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

Oracle vs. Regression Model



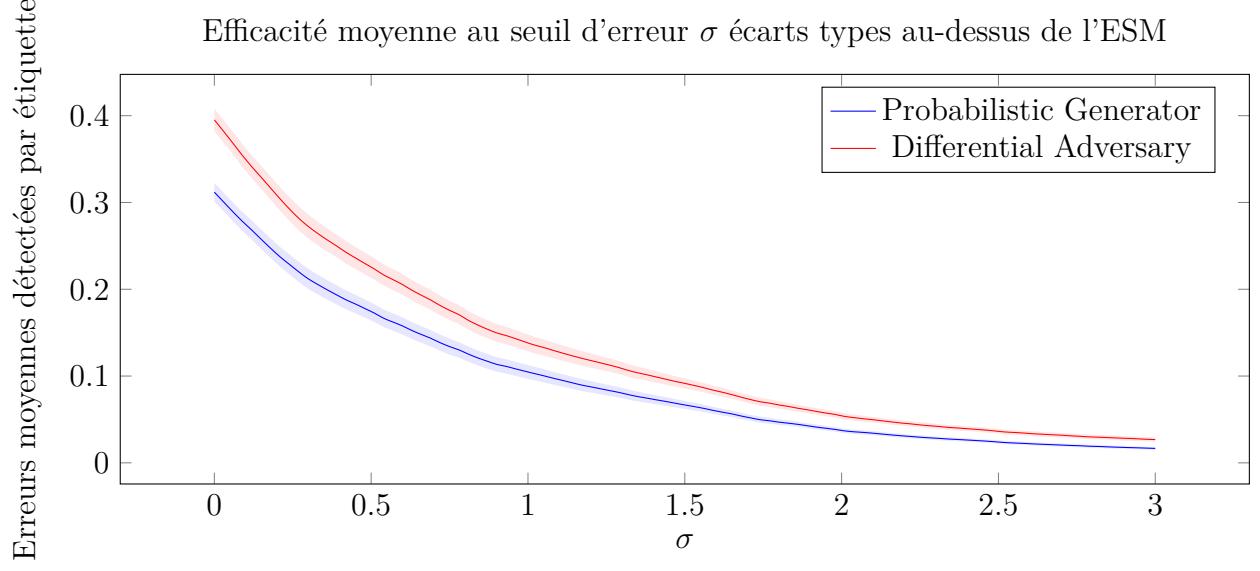
True vs. Surrogate Loss



**Table 4.2.** Au-dessus: Vérité de base et prédictions de modèles entraînés pour une seule expression. En bas: Une particule unique attaque le modèle en cherchant une erreur plus élevée sur la perte de substitution.

Notre adversaire ([Algorithm 5](#)) prend comme entrée le modèle de régression formé  $\hat{f}$ , un ensemble de nouvelles paires entrée-sortie de l'expression de vérité de terrain, et reprend la procédure de formation originale sur  $\hat{f}$  en utilisant les points de données fournis pour un nombre fixe d'époques, pour produire un nouveau modèle  $\hat{f}'$ . Nous utilisons  $\hat{f}'$  pour construire une perte de substitution  $\hat{\mathcal{L}}(x) = (\hat{f}(x) - \hat{f}'(x))^2$ , qui peut être maximisée en utilisant [Algorithm 4](#). La maximisation de la perte de substitution nous permet de construire des exemples contradictoires sans accès direct à l'oracle, une hypothèse souvent peu pratique dans

le monde réel. Pour les tests contradictoires et les stratégies d'échantillonnage uniformes, nous comparons le nombre moyen de violations détectées par évaluation.



**Fig. 4.3.** By construction, our shrinker detects a greater number of errors per evaluation than one which does not take the gradient into consideration.

Ci-dessus, nous montrons le nombre de violations dépassant le seuil d'erreur  $\sigma$  écarts types au-dessus de l'erreur quadratique moyenne (EQM) sur la perte réelle  $\mathcal{L}(x) = (f(x) - \hat{f}(x))^2$ . En moyenne, notre adversaire affiche une amélioration de 28% par rapport à la base de référence probabiliste pour tous les seuils. Nous émettons l'hypothèse que la formation de la perte de substitution à la convergence élargirait encore cette marge, bien que potentiellement au prix d'une généralisation sur d'autres expressions.

De plus, nous supposons que si une fraction suffisamment importante de l'espace d'entrée existait où les  $T$  étaient faux, alors en échantillonnant à partir de cet espace, la probabilité de détection approcherait 1:

Supposons qu'un service d'apprentissage machine dispose d'un budget de  $B$  et de coûts fixes de  $C$  pour l'étiquetage d'un seul point de données. Étant donné un seuil d'erreur  $e_{min}$  et un accès aux nouvelles entrées  $\frac{B}{C}$  de l'oracle, supposons que notre méthode détecte plus d'erreurs qu'une stratégie d'échantillonnage aléatoire uniforme sur l'espace d'entrée. Inversement, pour détecter le même nombre d'erreurs, nous avons besoin d'un budget inférieur à celui d'une stratégie d'échantillonnage aléatoire.

### 4.3. Conclusion

Dans ce chapitre, nous avons examiné quelques idées intéressantes pour valider les systèmes intelligents du point de vue du génie logiciel et de l'apprentissage machine. Nous avons constaté une curieuse ressemblance entre certaines idées nouvelles et anciennes dans le domaine des tests de fuzz et de l'apprentissage contradictoire. Nous avons proposé un nouveau cadre pour évaluer les programmes différentiables de manière peu coûteuse et avons montré que notre approche est plus efficace en termes de données qu'une stratégie de recherche aléatoire, utilisée par la plupart des cadres de test automatisés. Cela nous permet de détecter un plus grand nombre d'erreurs avec un budget de calcul et de collecte de données plus faible. L'auteur tient à remercier Liam Paull pour ses nombreuses suggestions très utiles et Stephen Samuel pour l'excellente bibliothèque [KotlinTest](#) [Samuel and Lopes, 2018]. Ce travail a été en partie inspiré par [Lample and Charton \[2019\]](#), en particulier le générateur d'arbres d'expression de § 4.2.



# Chapter 5

---

## Outils pour la robotique reproductible

La seule façon de réaliser des progrès substantiels dans un domaine quelconque est de s'appuyer sur le travail des autres. Pourtant, la programmation informatique reste une industrie artisanale car les programmeurs insistent pour réinventer des programmes pour chaque nouvelle application, au lieu d'utiliser ce qui existe déjà. Nous devons encourager une manière de regrouper les programmes de manière à ce qu'ils puissent être perçus comme des outils standard, chacun accomplissant sa tâche spécialisée suffisamment bien et s'interfaisant avec d'autres outils de manière si pratique que les programmeurs ressentent rarement le besoin de créer leur propre version à partir de zéro.”

---

—Kernighan and Plauger [1976], *Software tools*

Dans ce chapitre, nous abordons le défi de la reproductibilité des logiciels et la manière dont les meilleures pratiques du génie logiciel, telles que les outils d'intégration continue et de conteneurisation, peuvent aider les chercheurs à atténuer la variabilité associée à la construction et à la maintenance des logiciels de robotique. D'une manière générale, nos travaux tentent d'isoler les sources de variabilité computationnelle, et n'envisagent pas les notions de variabilité statistique découlant de l'incertitude aléatoire ou épistémique [Diaz Cabrera, 2018]. Cependant, la réduction de la variabilité informatique (qui entrave souvent la reproductibilité expérimentale) est une étape clé pour permettre aux chercheurs d'identifier et de diagnostiquer plus rapidement ces variables plus insaisissables en robotique et en apprentissage machine.

Afin d'aborder la question de la reproductibilité des logiciels, nous avons rassemblé un ensemble d'outils et de flux de développement représentant les meilleures pratiques en matière de génie logiciel. Ces outils sont principalement basés sur la *containerisation*, une technologie de virtualisation largement adoptée dans l'industrie du logiciel. Pour abaisser la

barrière d’entrée pour les nouveaux contributeurs et minimiser la variabilité entre les plates-formes matérielles, nous avons développé une infrastructure de conteneurs de pointe basée sur Docker [Merkel, 2014], un moteur de conteneurs très populaire. Docker permet aux utilisateurs de mettre en place des artefacts de déploiement versionnés qui figent efficacement un système de fichiers entier, et de gérer les contraintes de ressources via un environnement d’exécution en "sandboxed".

Le contenu de ce chapitre est organisé comme suit. Dans § 5.1 nous introduisons le problème de la résolution des dépendances et le défi de la construction d’artefacts logiciels reproductibles. Dans § 5.2, nous décrivons une solution générale à ce problème, la virtualisation des logiciels. Ensuite, dans § 5.3, nous discutons d’une approche légère de la virtualisation, connue sous le nom de conteneurisation. Dans § 5.4, nous faisons une visite guidée à travers une implémentation de conteneur, appelée Docker. Enfin, dans § 5.5, nous présentons DuckieOS, un environnement dockerisé pour la construction d’applications robotiques reproductibles pour la recherche et l’utilisation pédagogique.

## 5.1. Dependency management

Une source commune de variabilité dans le développement de logiciels est la dépendance des logiciels. Pendant de nombreuses années, les développeurs ont eu du mal à gérer les dépendances avant qu’on ne découvre que le problème de résolution des dépendances était NP-complete [Abate et al., 2012]. Si nous supposons qu’aucune version de la même dépendance ne peut être installée simultanément, alors pour un ensemble donné de logiciels qui doivent être installés, et de dépendances nécessaires pour les installer, la détermination de la version cohérente la plus récente des dépendances est aussi difficile que les problèmes les plus difficiles de NP. Officieusement, ce problème est connu sous le nom de *dependency hell* et devient de plus en plus problématique à mesure que les projets logiciels se développent et introduisent de nouvelles dépendances.

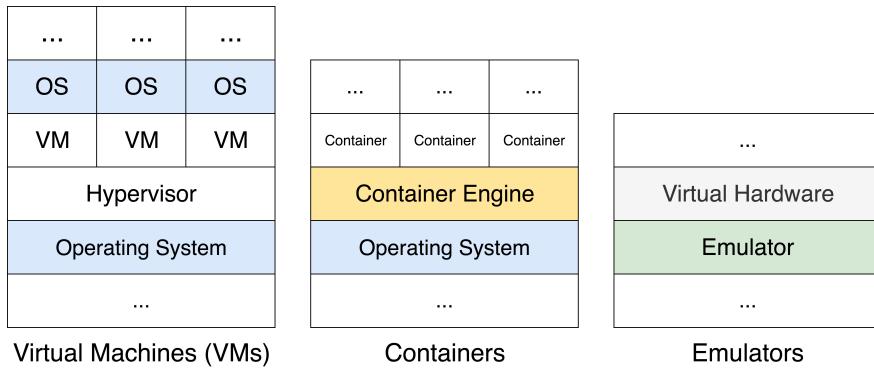
L’enfer des dépendances ne se produit pas seulement à l’intérieur des projets logiciels individuels, mais aussi à travers les projets et les environnements de développement. Des centaines de gestionnaires de paquets ont été développés pour divers systèmes d’exploitation, langages de programmation et cadres de développement. Ubuntu dispose du Advanced Package Tool (`apt`), macOS a Homebrew (`brew`), Windows a Chocolatey (`choco`). La plupart

des écosystèmes de langage de programmation ont leurs propres gestionnaires de paquets sur mesure ; [Conan](#) pour C/C++, [Maven](#) pour Java, et [Cabal](#) pour Haskell. Python a développé de nombreuses solutions qui se chevauchent pour la gestion des paquets, notamment [pip](#), [Anaconda](#), [PyEnv](#), [Virtualenv](#), et d'autres. Certains d'entre eux installent des paquets pour l'ensemble du système, et d'autres fournissent des environnements en ligne de commande. Au cours de la vie d'un système informatique, à mesure que les paquets sont installés et partiellement supprimés, il devient difficile de suivre les changements et leurs effets secondaires.

Le problème provient essentiellement de l'exigence selon laquelle aucune version de la même dépendance ne peut être installée simultanément. En outre, les installateurs de logiciels ont tendance à pulvériser des fichiers dans le système de fichiers, qui peuvent être corrompus et sont difficiles à supprimer complètement si le besoin s'en fait sentir. Pour résoudre ces problèmes, il faut une certaine notion de "contrôle", de sorte que lorsqu'un nouveau logiciel est installé, toute modification ultérieure puisse être retracée et annulée. Des sauvegardes matérielles feraient l'affaire, mais elles sont lourdes à gérer et ne conviennent pas à des fins de développement. Il serait plutôt pratique de disposer d'un outil permettant aux applications de créer un système de fichiers privé, d'installer leurs dépendances et d'éviter de contaminer le système d'exploitation hôte.

## 5.2. Operating systems and virtualization

Avec la croissance des opérations de développement (devops), un certain nombre de solutions ont émergé pour la construction et l'exécution d'artefacts logiciels génériques. La plupart de ces solutions sont des émulateurs, qui simulent complètement une architecture de processeur étrangère, et donc tout logiciel qui s'exécute sur celle-ci. Une autre solution est celle des machines virtuelles (VM), une sorte d'environnement d'exécution isolé qui utilise un *hypervisor* pour faciliter l'accès au matériel, mais qui fonctionne généralement sur du métal nu. L'inconvénient de ces deux méthodes est leur efficacité. Les machines virtuelles contiennent des systèmes d'exploitation à part entière et sont donc lourdes à exécuter et à déboguer. Cela est particulièrement inutile pour construire et exécuter une petite application sur un système d'exploitation étranger. Les émulateurs fonctionnent beaucoup plus lentement que le code machine natif, selon les architectures hôte et cible.



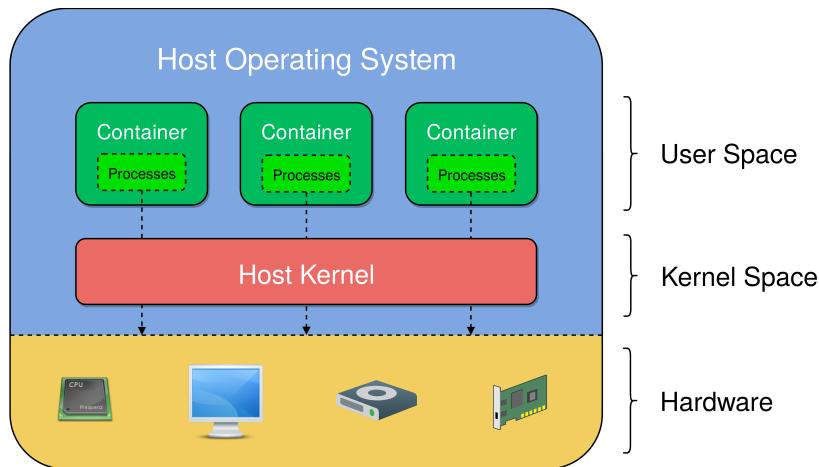
**Fig. 5.1.** La virtualisation complète est un processus très gourmand en ressources. La conteneurisation est moins coûteuse, car elle partage un noyau avec le système d’exploitation hôte. L’émulation nous permet d’émuler le matériel comme un logiciel. Chacune de ces méthodes peut être utilisée en conjonction avec n’importe quelle autre.

En 2006, Linux a introduit plusieurs nouvelles fonctionnalités du noyau pour contrôler des groupes de processus, sous l’égide de **cgroups** [Menage, 2007]. Collectivement, ces fonctionnalités prennent en charge une forme de virtualisation légère, présentant de nombreux avantages des machines virtuelles (VM) tels que le contrôle des ressources et l’isolation de l’espace de noms, sans les surcharges de calcul associées à la virtualisation complète. Ces fonctionnalités ont ouvert la voie à un ensemble d’outils qui sont aujourd’hui connus sous le nom de conteneurs. Contrairement aux machines virtuelles, les conteneurs partagent un noyau commun, mais restent isolés de leur système d’exploitation hôte et des conteneurs frères et surs. Alors que les machines virtuelles nécessitent souvent du matériel de type serveur pour fonctionner correctement, les conteneurs conviennent à une catégorie beaucoup plus large de plateformes mobiles et embarquées en raison de leur faible empreinte sur les ressources.

### 5.3. Containerization

L’un des défis du développement de logiciels distribués sur des plates-formes hétérogènes est le problème de la variabilité. L’accélération du rythme de développement des logiciels s’accompagne d’une charge supplémentaire de maintenance des logiciels. À mesure que les piles de matériel et de logiciels évoluent, le code source doit être périodiquement mis à jour pour être construit et fonctionner correctement. Maintenir une base de code stable et bien documentée peut représenter un défi considérable, en particulier dans un cadre universitaire

où les contributeurs rejoignent et quittent fréquemment un projet. Ensemble, ces défis constituent des obstacles importants à la reproductibilité expérimentale et à la collaboration scientifique.



**Fig. 5.2.** Containers live in user space. Par défaut, ils sont mis en bac à sable à partir de l'OS hôte et des conteneurs frères et surs, mais contrairement aux VM, ils partagent un noyau commun entre eux et avec l'OS hôte. Tous les appels système sont passés par le noyau hôte.

Les conteneurs Docker sont des environnements d'exécution en bac à sable qui sont portables, reproductibles et dont la version est contrôlée. Chaque environnement contient toutes ses dépendances, mais reste isolé du système d'exploitation et du système de fichiers hôte. Docker fournit un mécanisme pour contrôler les ressources auxquelles chaque conteneur est autorisé à accéder, et fournit un espace de noms Linux séparé pour chaque conteneur, isolant efficacement le réseau, les utilisateurs et les montages du système de fichiers du système d'exploitation hôte. Contrairement aux machines virtuelles, les outils de virtualisation basés sur des conteneurs comme Docker sont adaptés aux SBC portables et peuvent fonctionner avec une surcharge proche de zéro par rapport aux processus Linux natifs. Un seul Raspberry Pi est capable d'exécuter simultanément des centaines de conteneurs sans dégradation notable des performances. Espace-.08em<sup>1</sup>

---

<sup>1</sup><https://blog.docker.com/2015/09/update-raspberry-pi-dockercon-challenge/>

Si la conteneurisation simplifie considérablement le processus de création et de déploiement des applications, elle introduit également une certaine complexité supplémentaire dans le cycle de vie du développement des logiciels. Docker, comme la plupart des plates-formes de conteneur, utilise un système de fichiers en couches. Cela permet à Docker de prendre une "image" existante et de la modifier en installant de nouvelles dépendances ou en modifiant ses fonctionnalités. Les images sont généralement construites comme une séquence de couches, chacune devant être mise à jour périodiquement. Il faut veiller, lors de la conception du pipeline de développement, à ce que ces mises à jour ne brisent pas silencieusement une couche suivante, comme nous le décrivons dans § 5.7.

## 5.4. Introduction to Docker

Supposons qu'il existe un programme connu pour fonctionner sur un ordinateur quelconque. Il serait bon de donner à un autre ordinateur – n'importe quel ordinateur connecté à Internet – une courte chaîne de caractères ASCII, d'appuyer sur les touches  et de revenir pour voir ce même programme s'exécuter. Peu importe où le programme a été construit ou quel logiciel était en cours d'exécution à ce moment-là. Cela peut sembler trivial, mais c'est un problème monumental de génie logiciel. Divers gestionnaires de paquets ont tenté de résoudre ce problème, mais même lorsqu'ils fonctionnent comme prévu, ils ne prennent en charge que les binaires compilés en natif sur les systèmes d'exploitation ayant le même gestionnaire de paquets.

Docker<sup>2</sup> est un outil de calcul portable et reproductible. Avec Docker, les utilisateurs peuvent exécuter n'importe quel programme Linux sur presque tous les appareils informatiques en réseau de la planète, quel que soit le système d'exploitation ou l'architecture matérielle sous-jacents. Toutes les étapes de préparation, d'installation et de configuration de l'environnement peuvent être automatisées du début à la fin. Selon la largeur de bande disponible sur le réseau, cela peut prendre un certain temps, mais les utilisateurs n'auront jamais besoin d'intervenir dans le processus d'installation.

Pour installer Docker lui-même, exéutez la commande suivante sur un shell conforme à POSIX de n'importe quelle [Plateforme supportée par Docker](#):

---

<sup>2</sup>Le tutoriel suivant utilise Docker, mais le flux de travail décrit est similaire à celui de la plupart des plateformes de conteneurs.



```
~$ curl -sSL https://get.docker.com/ | sh
```

1

Un Docker *image* est essentiellement un instantané du système de fichiers – un fichier unique qui contient tout ce qui est nécessaire pour faire fonctionner un certain conteneur du Docker. Les images des Dockers sont hébergées dans des *registres*, similaires aux dépôts Git ou aux serveurs VCS. La commande suivante permet d'extraire une image Docker, par exemple **daphne/duck** du dépôt par défaut Docker Hub:



```
~$ docker pull daphne/duck
```

1

Chaque image du Docker a un identifiant, un nom et une étiquette:



REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
daphne/duck	latest	ea2f90g8de9e	1 day ago	869MB

1

2

3

Pour lancer un conteneur Docker, utilisez la commande suivante:



```
~$ docker run daphne/duck
```

1

La commande suivante permet de vérifier que le conteneur est bien en cours d'exécution:



CONTAINER ID	IMAGE	...	NAMES
52994ef22481	daphne/duck	...	happy_hamster

1

2

3

Notez que l'image de Daphné possède un identifiant alphanumérique du conteneur, **52994ef22481**, une image de base, **daphne/duck**, et un nom mémorable, **happy\_hamster**. Ce nom est un alias pour l'ID du conteneur, qui peut être utilisé de manière interchangeable pour faire référence au conteneur.

Les images de docker peuvent être créées de deux manières différentes. Tout d'abord, dans § 5.4.1, nous verrons comment créer une image Docker en prenant un instantané d'un conteneur en cours d'exécution, puis dans § 5.4.2, comment créer un nouveau conteneur Docker en utilisant un type de recette spécial, appelé **Dockerfile**.

### 5.4.1. Creating an image snapshot

, comment créer un nouveau conteneur Docker en utilisant une recette spéciale, appelée **Dockerfile**.

Lorsqu'un conteneur du Docker écrit sur son propre système de fichiers, ces modifications ne sont pas persistantes, sauf si elles sont appliquées à une nouvelle image. Par exemple, démarrez un conteneur avec un shell interactif:

```
~$ docker run -it daphne/duck /bin/bash  
root@295fd7879184:/#
```

Notez l'ID du conteneur : **295fd7879184**. Si on écrit sur le disque et qu'on laisse le conteneur,

```
root@295fd7879184:/# touch new_file && ls -l  
total 0  
-rw-r--r-- 1 root root 0 May 21 20:52 new_file  
root@295fd7879184:/# exit
```

**new\_file** ne sera pas persisté. Si nous relançons la même commande:

```
~$ docker run -it daphne/duck /bin/bash  
root@18f13bb4571a:/# ls  
root@18f13bb4571a:/# touch new_file1 && ls -l  
total 0  
-rw-r--r-- 1 root root 0 May 21 21:32 new_file1
```

Il semble que le "nouveau fichier" ait disparu ! Remarquez que l'ID du conteneur (**18f13bb4571a**) est maintenant différent. C'est parce que la commande **docker run daphne/duck** a créé un nouveau conteneur à partir de l'image de base **daphne/duck**, plutôt que de redémarrer le conteneur précédent. Pour voir tous les conteneurs sur un hôte Docker, exécutez la commande suivante:

```
~$ docker container ls -a  
CONTAINER ID        IMAGE               STATUS            NAMES  
295fd7879184        daphne/duck       Exited (130)      merry_manatee  
18f13bb4571a        daphne/duck       Up 5 minutes    shady_giraffe
```

52994ef22481	daphne/duck	Up 10 minutes	happy_hamster
--------------	-------------	---------------	---------------

5

Il semble que `295fd7879184` alias `merry_manatee` ait survécu, mais il ne fonctionne plus. Chaque fois que le processus principal d'un conteneur (rappelons que nous avons exécuté `merry_manatee` avec `/bin/bash`) se termine, le conteneur s'arrête, mais il ne cesse pas d'exister. En fait, nous pouvons reprendre le conteneur arrêté là où il s'est arrêté:

```
~$ docker start -a merry_manatee
root@295fd7879184:/# ls -l
total 0
-rw-r--r-- 1 root root 0 May 21 20:52 new_file
```

1  
2  
3  
4

Rien n'a été perdu ! Pour le vérifier, nous pouvons vérifier quels autres conteneurs sont en cours d'utilisation:

```
~$ docker ps
CONTAINER ID        IMAGE               ...        NAMES
295fd7879184        daphne/duck       ...        merry_manatee
18f13bb4571a        daphne/duck       ...        shady_giraffe
52994ef22481        daphne/duck       ...        happy_hamster
```

1  
2  
3  
4  
5

Supposons maintenant que nous voulions partager le conteneur `shady_giraffe` avec quelqu'un d'autre. Pour ce faire, nous devons d'abord prendre un instantané du conteneur en cours d'utilisation, ou le placer dans une nouvelle image avec un nom et une balise. Cela créera un point de contrôle que nous pourrons restaurer ultérieurement:

```
~$ docker commit -m "forking daphne/duck" shady_giraffe user/duck:v2
```

1

Pour faire référence au conteneur, on peut utiliser soit `18f13bb4571a` soit le nom désigné, c'est-à-dire `shady_giraffe`. Ce dépôt d'images sera appelé `user/duck`, et possède un identifiant de version optionnel, `:v2`, qui peut être poussé dans le registre Docker Hub:

```
~$ docker push user/duck:v2
~$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
```

1  
2  
3

```

daphne/duck    latest      ea2f90g8de9e   1 day ago     869MB
user/duck      v2          d78be5cf073e   2 seconds ago  869MB
~$ docker pull user/duck:v2
~$ docker run user/duck:v2 ls -l
total 0
-rw-r--r-- 1 root root 0 May 21 21:32 new_file1

```

C'est un moyen pratique de partager une image avec des collègues et des collaborateurs. Toute personne ayant accès au dépôt peut retirer cette image et continuer là où nous nous sommes arrêtés, ou créer une autre image basée sur le dessus.

#### 5.4.2. Writing an image recipe

La deuxième façon de créer une image Docker est d'écrire une recette, appelée **Docker file**. Un **Dockerfile** est un fichier texte qui spécifie les commandes requises pour créer une image du Docker, généralement en modifiant une image de conteneur existante à l'aide d'une interface de script. Ils comportent également des **mots-clés** spéciaux (qui sont traditionnellement **CAPITALIZED**), comme **FROM**, **RUN**, **ENTRYPOINT** et ainsi de suite. Par exemple, créez un fichier appelé **Dockerfile** avec le contenu suivant:



```

FROM daphne/duck      # Defines the base image
RUN touch new_file1   # new_file1 will be part of our snapshot
CMD ls -l             # Default command run when container is started

```

Maintenant, pour construire l'image, nous pouvons simplement courir:



```
~$ docker build -t user/duck:v3 .
```

Le **.** indique le répertoire courant, qui doit être le même que celui contenant notre **Docker file**. Cette commande devrait produire quelque chose comme la sortie suivante:



```

Sending build context to Docker daemon 2.048kB
Step 1/3 : FROM daphne/duck
--- ea2f90g8de9e
Step 2/3 : RUN touch new_file1
--- e3b75gt9zyc4

```

```

Step 3/3 : CMD ls -l
--- Running in 14f834yud59
Removing intermediate container 14f834yud59
--- 05a3bd381fc2
Successfully built 05a3bd381fc2
Successfully tagged user/duck:v3

```

La commande, `docker images` devrait afficher une image appelée `user/duck:v3`:

```

~$ docker images
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
daphne/duck     latest       ea2f90g8de9e   1 day ago    869MB
user/duck        v2          d78be5cf073e   5 minutes ago 869MB
user/duck        v3          05a3bd381fc2   2 seconds ago 869MB

```

Cette procédure est identique à la technique d'instantané réalisée dans § 5.4.1, mais le résultat est plus propre. Plutôt que de conserver une image de 869 Mo, on peut se contenter de stocker le fichier texte de 4 Ko. Pour exécuter l'image résultante, nous pouvons simplement utiliser la même commande qu'auparavant:

```

~$ docker run -it user/duck:v3
total 0
-rw-r--r-- 1 root root 0 May 21 21:35 new_file1

```

Notez que dès que nous lançons le conteneur, Docker exécute la commande `ls -l` comme spécifié par le `Dockerfile`, révélant que `new_file1` a bien été stocké dans l'image. Cependant, cette commande par défaut peut être remplacée par une commande personnalisée:

```

~$ docker run -it user/duck:v3 [custom command]

```

### 5.4.3. Layer caching

Layers sont un concept important à comprendre quand on travaille avec Docker. Une façon de penser à une couche est comme un commit Git – un ensemble de modifications

à une image ou une couche précédente, identifiée de façon unique par un code de hachage. Dans un **Dockerfile**, les calques commencent par un [mot-clé](#).



```
FROM daphne/duck
RUN touch new_file1          # Defines a new layer
RUN mkdir config && mv new_file1 config    # Layers can chain commands
RUN apt-get update && apt-get install -y wget # Install a dependency
RUN wget https://get.your.app/install.sh      # Download a script
RUN chmod +x install.sh && ./install.sh     # Run the script
```

1  
2  
3  
4  
5  
6

Pour construire cette image, nous pouvons exécuter la commande suivante:



```
~$ docker build -t user/duck:v4 .
Sending build context to Docker daemon 2.048kB
Step 1/6 : FROM daphne/duck
--> cd6d8154f1e1
...
Removing intermediate container 8fb56ef38bc8
--> 3358ca1b8649
Step 5/6 : RUN wget https://get.your.app/install.sh
--> Running in e8284ff4ec8b
...
2018-10-30 06:47:57 (89.9 MB/s) - 'install.sh' saved [13847/13847]
Removing intermediate container e8284ff4ec8b
--> 24a22dc2900a
Step 6/6 : RUN chmod +x install.sh && ./install.sh
--> Running in 9526651fa492
# Executing install script, commit: 36b78b2
...
Removing intermediate container 9526651fa492
--> a8be23fea573
Successfully built a8be23fea573
Successfully tagged user/duck:v4
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21

Les couches sont mises en cache de manière pratique par le [Docker daemon](#). Si nous devons exécuter deux fois la même commande, Docker utilisera le cache au lieu de reconstruire l'image entière:

```
 Sending build context to Docker daemon 2.048kB  
Step 1/6 : FROM daphne/duck  
---> cd6d8154f1e1  
Step 2/6 : RUN touch new_file1  
---> Using cache  
---> 0473154b2004  
...  
Step 6/6 : RUN chmod +x index.html && ./index.html  
---> Using cache  
---> a8be23fea573  
Successfully built a8be23fea573  
Successfully tagged user/duck:v4
```

Si nous devons apporter une modification au [Dockerfile](#), Docker ne reconstruira l'image qu'à partir de la première étape modifiée. Supposons que nous ajoutions une nouvelle commande `RUN` à la fin de notre [Dockerfile](#) et que nous déclenchions une reconstruction de la même manière:

```
 ~$ echo 'RUN echo "Change here!"' >> Dockerfile  
~$ docker build -t user/duck:v4 .  
Sending build context to Docker daemon 2.048kB  
...  
Step 6/7 : RUN chmod +x index.html && ./index.html  
---> Using cache  
---> a8be23fea573  
Step 7/7 : RUN echo "Change here!"  
---> Running in 80fc5c402304  
Change here!  
Removing intermediate container 80fc5c402304  
---> c1ec64cef9c6  
Successfully built c1ec64cef9c6
```



Si Docker devait relancer l'ensemble du [fichier] en ligne de haut en bas à chaque reconstruction, cela serait lent et peu pratique. Au lieu de cela, Docker met en cache les étapes non modifiées par défaut, et ne refait que le minimum d'étapes nécessaires à la reconstruction. Cela peut parfois donner des résultats inattendus, surtout lorsque la cache est périmée. Pour ignorer le cache et forcer une reconstruction propre, utilisez l'indicateur `--no-cache` lors de la construction d'un `Dockerfile`.

Que prend en compte Docker pour décider d'utiliser le cache? Tout d'abord, le `Dockerfile` lui-même - lorsqu'une étape d'un `Dockerfile` change, elle et toutes les étapes suivantes devront être relancées lors de la compilation. Docker vérifie également le `contexte de construction` pour les changements. Lorsque `docker build -t TAG .` est écrit, le `.` indique le contexte de construction, ou le chemin où la construction doit avoir lieu. Souvent, ce chemin contient des artefacts de compilation. Par exemple:

```
 FROM daphne/duck
COPY duck.txt .
RUN cat duck.txt
```

1  
2  
3

Maintenant, si nous ajoutons un message dans `duck.txt` et reconstruisons notre image, le fichier sera copié dans l'image du Docker, et son contenu sera imprimé:

```
 ~$ echo "Make way!" > duck.txt && docker build -t user/duck:v5 .
Sending build context to Docker daemon 3.072kB
Step 1/3 : FROM daphne/duck
--> cd6d8154f1e1
Step 2/3 : COPY duck.txt .
--> e0e03d9e1791
Step 3/3 : RUN cat duck.txt
--> Running in 590c5420ce29
Make way!
Removing intermediate container 590c5420ce29
--> 1633e3e10bef
Successfully built 1633e3e10bef
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12

Tant que les trois premières lignes du `Dockerfile` et du `duck.txt` ne sont pas modifiées, ces couches seront mises en cache et Docker ne les reconstruira pas. Si le contenu du fichier `duck.txt` est modifié par la suite, cela déclenchera une reconstruction. Par exemple, si nous ajoutons au fichier et le reconstruisons, seules les deux dernières étapes devront être exécutées:

```
~$ echo "Thank you. Have a nice day!" >> duck.txt
~$ docker build -t user/duck:v5 .
Sending build context to Docker daemon 3.072kB
Step 1/3 : FROM ubuntu
--> cd6d8154f1e1
Step 2/3 : COPY duck.txt .
--> f219efc150a5
Step 3/3 : RUN cat duck.txt
--> Running in 7c6f5f8b73e9
Make way!
Thank you. Have a nice day!
Removing intermediate container 7c6f5f8b73e9
--> e8a1db712aee
Successfully built e8a1db712aee
Successfully tagged user/duck:v5
```

Une erreur courante lors de l'écriture de `Dockerfiles` consiste à `COPY` plus de fichiers que ce qui est strictement nécessaire pour effectuer l'étape de construction suivante. Par exemple, si `COPY ..` est écrit au début du `Dockerfile`, chaque fois qu'un fichier est modifié dans le contexte de la compilation, cela déclenchera une reconstruction de toutes les étapes de compilation suivantes. Afin de maximiser la réutilisation du cache et de réduire au minimum le temps de reconstruction, les utilisateurs doivent être aussi prudents que possible et ne `COPY` que le minimum de fichiers nécessaires pour accomplir l'étape de construction suivante.

#### 5.4.4. Volume sharing

Il existe une deuxième méthode pour déposer les données dans un conteneur, qui ne nécessite pas de les intégrer dans l'image mère au moment de la compilation. Cette méthode est plus appropriée pour les données qui sont nécessaires au moment de l'exécution, mais non essentielles pour la construction. Elle prend la forme suivante:

```
~$ docker run user/duck:v6 -v HOST_PATH:TARGET_PATH
```

1

Supposons que nous ayons un **Dockerfile** qui fournit une instruction **CMD** par défaut:

```
de daphne/duck
CMD /bin/bash -c "/launch.sh"
```

1

2

Si nous construisions cette image et essayions de l'exécuter, le fichier `/launch.sh` serait manquant:

```
~$ docker build -t user/duck:v6 && docker run user/duck:v6
bash: /launch.sh: No such file or directory
```

1

2

Au lieu de cela, lorsque nous utilisons le conteneur, nous devons partager le fichier via le Docker CLI:

```
~$ echo -e '#!/bin/bash\necho Launching...' >> launch.sh && \
chmod 775 launch.sh && \
docker run user/duck:v6 -v launch.sh:/launch.sh
Launching...
```

1

2

3

4

De cette façon, le fichier local `launch.sh` sera disponible pour être utilisé à l'intérieur du conteneur au chemin désigné, `/launch.sh`.

#### 5.4.5. Multi-stage builds

Le système de fichiers de Docker est additif, donc chaque couche ne fera qu'augmenter la taille de l'image finale. Pour cette raison, il est souvent nécessaire de mettre de l'ordre dans les fichiers inutiles après l'installation. Par exemple, lors de l'installation de dépendances sur des images basées sur Debian, il est courant de les exécuter:



```
RUN apt-get update && apt-get install ... && rm -rf /var/lib/apt/lists/*
```

1

Cela garantit que la liste des paquets n'est pas intégrée à l'image (Docker ne vérifie le calque qu'une fois chaque étape terminée). Les constructions peuvent souvent consommer plusieurs étapes, bien qu'elles ne produisent qu'un seul artefact. Au lieu d'enchaîner plusieurs commandes et de nettoyer les changements en une seule étape, les constructions à plusieurs étapes nous permettent de construire une série d'images à l'intérieur d'un **Dockerfile**, et de copier les ressources de l'une à l'autre, en éliminant tous les artefacts de construction intermédiaires:



```
FROM user/duck:v3 as template1  
  
FROM daphne/duck as template2  
COPY --from=template1 new_file1 new_file2  
FROM donald/duck as template3  
COPY --from=template2 new_file2 new_file3  
CMD ls -l
```

1

2

3

4

5

6

7

Nous pouvons maintenant construire et faire fonctionner cette image comme suit:



```
~$ docker build . -t user/duck:v4  
Sending build context to Docker daemon 2.048kB  
Step 1/6 : FROM user/duck:v3 as template1  
--- e3b75ef8ecc4  
Step 2/6 : FROM daphne/duck as template2  
--- ea2f90g8de9e  
Step 3/6 : COPY --from=template1 new_file1 new_file2  
---> 72b96668378e  
Step 4/6 : FROM donald/duck:v3 as template3  
---> e3b75ef8ecc4  
Step 5/6 : COPY --from=template2 new_file2 new_file3  
---> cb1b84277228  
Step 6/6 : CMD ls  
---> Running in cb1b84277228
```

1

2

3

4

5

6

7

8

9

10

11

12

13

14



```

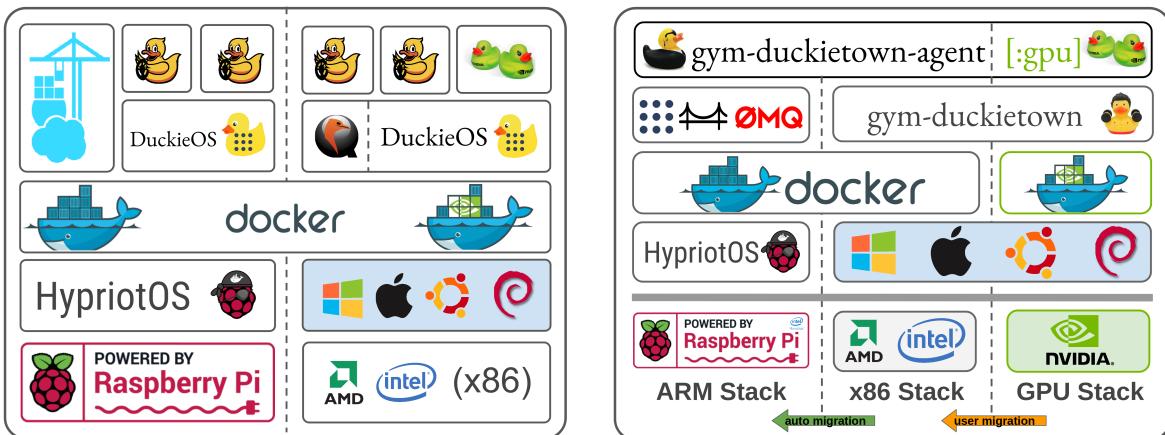
Removing intermediate container cb1b84277228
---> c7dc5dd63e77
Successfully built c7dc5dd63e77
Successfully tagged user/duck:v4
~$ docker run -it user/duck:v4
total 0
-rw-r--r-- 1 root root 0 Jul  8 15:06 new_file3

```

15  
16  
17  
18  
19  
20  
21

Une des applications des constructions en plusieurs étapes consiste à compiler une dépendance de projet à partir de son code source. En plus de tout le code source, le processus de compilation pourrait introduire des gigaoctets d'artefacts de construction et de dépendances transitives, juste pour construire un seul binaire. Les compilations multi-étapes nous permettent de construire le fichier, et de le copier sur une nouvelle couche, libérée des fichiers intermédiaires.

## 5.5. ROS and Docker



**Fig. 5.3.** Infrastructure des conteneurs. **Gauche** : La pile ROS cible deux architectures primaires, x86 et ARM. Pour simplifier le processus de construction, nous construisons des artefacts ARM sur x86 en utilisant QEMU [Bellard, 2005]. **Right** : Pile d'apprentissage de renforcement. Les artefacts de construction sont formés sur un GPU, et transférés sur le CPU pour évaluation. Les modèles d'apprentissage profond peuvent également être exécutés sur un dispositif ARM à l'aide d'un accélérateur.

White and Christensen [2017] a précédemment exploré les ROS Dockerizing, dont les travaux constituent la base des nôtres, qui étendent leur mise en uvre à la plateforme Duckietown [Paull et al., 2017], un ensemble d'applications ROS plus spécifiques au matériel et au domaine.

La [plateforme Duckietown](#) supporte deux architectures de jeu d'instructions primaires : x86 et ARM. Pour assurer la compatibilité d'exécution des paquets de Duckietown, nous effectuons une construction croisée en utilisant la virtualisation matérielle pour garantir que les artefacts de construction peuvent être exécutés sur l'une ou l'autre des architectures cibles. L'émulation en temps réel d'artefacts étrangers est également possible, en utilisant une technique similaire. Pour plus d'informations, cette technique est décrite plus en détail à l'URL suivante : <https://www.balena.io/blog/building-arm-containers-on-any-x86-machine-even-dockerhub/>. Par souci de performance et de simplicité, nous n'utilisons l'émulation que lorsque cela est nécessaire (par exemple sur les appareils x86). Sur ARM-native, le système d'exploitation de base est HypriotOS, une distribution Debian légère pour le Raspberry Pi et autres SBCs basés sur ARM, avec un support natif pour Docker. Pour les systèmes x86 et ARM, Docker est la plate-forme de conteneur sous-jacente sur laquelle toutes les applications utilisateur sont exécutées, à l'intérieur d'un conteneur. Comme ROS et Docker ont tous deux des interfaces de ligne de commande étendues, une interface unifiée, le [Duckietown Shell](#) ([dts](#)), est fourni pour envelopper leur fonctionnalité et effectuer des tâches communes.

## 5.6. Duckiebot development using Docker

Le développement du logiciel pour la plate-forme Duckietown nécessite les objets physiques suivants:

- (1) Duckiebot (y compris appareil photo, roues et Raspberry Pi 3B+)<sup>3</sup>
- (2) Carte Micro SD (16GB+ recommandé)
- (3) Ordinateur personnel
- (4) Routeur Internet
- (5) Adaptateur de carte MicroSD

En outre, nous supposons que les dépendances logicielles suivantes ont été installées sur (3):

- (a) [Docker CE](#)

---

<sup>3</sup>La liste complète des matériaux peut être trouvée à l'URL suivante : <https://get.duckietown.org/>

- (b) POSIX-compliant shell
- (c) `dts`, the Duckietown shell<sup>4</sup>
- (d) un navigateur Web (par exemple [Chrome](#) ou [Firefox](#))
- (e) `wget/curl`

Le flux de travail suivant a été testé de manière approfondie sur des hôtes Linux fonctionnant sous Ubuntu 16.04 (et dans une moindre mesure, sous Mac OS X et VM). Aucune autre dépendance n'est supposée ou requise.

### 5.6.1. Flashing a bootable disk

L'une des premières étapes du Duckiebook exige des utilisateurs qu'ils installent manuellement un système d'exploitation personnalisé sur un support amorçable, un processus fastidieux et long. Le script d'installation suivant a été écrit pour automatiser ce processus, permettant aux utilisateurs de configurer plus facilement un environnement logiciel reproduitible:

```
 ~$ bash -c "$(wget -O- h.ndan.co)"
```

1

Maintenant, avec le [Duckietown Shell](#), la commande suivante est tout ce qu'il faut:

```
 dt> init_sd_card [--hostname "DUCKIEBOT_NAME"] [--wifi "username:password"]
```

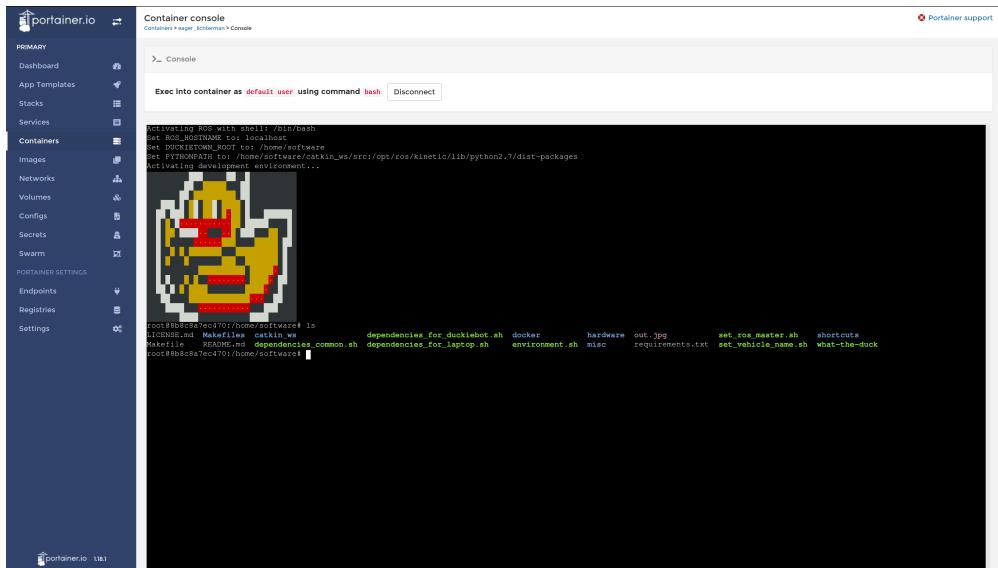
Les utilisateurs doivent insérer une carte SD et suivre les instructions fournies. Une fois terminée, la carte est retirée et insérée dans la fente pour carte SD du Raspberry Pi. Au premier démarrage, il faut veiller à ce que l'appareil soit alimenté en continu pendant au moins dix minutes afin de permettre la fin de l'installation et d'éviter la corruption du système de fichiers.

### 5.6.2. Web interface

Pour accéder à l'interface web de DuckieOS, les utilisateurs peuvent visiter l'URL suivante dans n'importe quel navigateur web compatible JavaScript : [http://DUCKIEBOT\\_NAME:9000/](http://DUCKIEBOT_NAME:9000/). Si le processus d'installation s'est achevé avec succès et que le réseau est correctement configuré, l'application web affichée dans [Fig. 5.4](#) devrait être accessible. Cette application

---

<sup>4</sup>May be obtained at the following URL : <https://github.com/duckietown/duckietown-shell>



**Fig. 5.4.** Interface de navigation pour les Duckiebots individuels. Elle est fournie par [Portainer](#), un tableau de bord web RESTful, qui enveloppe le CLI du Docker et offre un support pour la gestion des conteneurs, la configuration, la mise en réseau et l'émulation des terminaux (montré ci-dessus). [http://DUCKIEBOT\\_NAME:9000/#/container/container\\_name](http://DUCKIEBOT_NAME:9000/#/container/container_name)

permet aux utilisateurs non familiers avec le CLI de gérer les conteneurs Docker depuis leur navigateur préféré.

### 5.6.3. Testing ROS

Pour vérifier que Docker fonctionne correctement, lancez un conteneur à distance, de manière interactive, comme ceci:

```
~$ docker -H DUCKIEBOT_NAME run -it --privileged --net host \
duckietown/rpi-ros-kinetic-base:master18
```

Le drapeau `-H` indique un hôte Docker distant sur le réseau local où la commande Docker doit être exécutée. Pour que l'adresse `DUCKIEBOT_NAME` fonctionne, mDNS doit être correctement configuré dans les paramètres du réseau, sinon une adresse IP est nécessaire.

1  
2

#### 5.6.4. Build and deployment

Les images du Docker peuvent être compilées de manière croisée en incluant la partie spécifique à l'ARM du `Dockerfile` avec les instructions `RUN ["cross-build-start"]` et `RUN ["cross-build-end"]`. La commande suivante peut être utilisée pour le déploiement:



```
~$ docker save TAG_NAME | ssh -C duckie@DUCKIEBOT_NAME docker load
```

1

Il est également possible de construire directement sur les appareils ARM en créant un fichier nommé, par exemple `Dockerfile.arm`, en ajoutant une image de base et des instructions de construction, puis en exécutant la commande:



```
~$ docker build --file=FILE_PATH/Dockerfile.arm [--tag TAG_NAME] .
```

#### 5.6.5. Multi-architecture support

À partir de la version 18.09.6 de Docker, les fichiers ARM spécifiques `Dockerfile` ne seront pas construits sur les machines x86,<sup>5</sup>, et tenter d'en construire une produira l'erreur suivante lors de l'exécution de `docker build`:



```
standard_init_linux.go:175: exec user process caused "exec format error"
```

1

Afin de contourner cette restriction, les `Dockerfile` spécifiques à l'ARM peuvent être portés pour fonctionner sur x86 en utilisant les directives `RUN ["cross-build-start"]` et `RUN ["cross-build-end"]`, après les instructions `FROM` et avant les instructions `CMD`. Voir ?? pour plus de détails.

Toutes les images du Docker de Duckietown sont envoyées avec l'émulateur QEMU [Bellard, 2005] – cela nous permet d'exécuter directement des images ARM sur x86. Pour exécuter un nud ROS de calcul pur (c'est-à-dire qui ne nécessite aucun accès à une caméra ou à un moteur) sur une plate-forme x86, les développeurs doivent fournir un point d'entrée personnalisé à Docker lors de l'exécution de l'image en utilisant le drapeau de point d'entrée comme suit:

<sup>5</sup>À l'exception du client Docker de Mac OS, qui offre [support multi-architecture](#). Des versions plus récentes de Docker Desktop pour Mac OS et Windows ont [introduit une émulation ARM native](#).



```
~$ docker run ... --entrypoint=qemu3-arm-static IMAGE [RUN_COMMAND]
```

1

Ici, `RUN_COMMAND` peut être un shell tel que `/bin/bash` ou une autre commande telle que `/bin/bash -c "roscore"`. Le point d'entrée fait référence à l'émulateur ARM intégré dans l'image de base, `duckietown/rpi-ros-kinetic-base`, qui permet aux binaires ARM d'être exécutés sur des hôtes x86.

#### 5.6.6. Exécution d'un simple serveur de fichiers HTTP

Toutes les données persistantes sont stockées dans `/data`. Pour servir ce répertoire, un serveur web est fourni:



```
~$ docker -H DUCKIEBOT_NAME run -d -v /data:/data -p 8082:8082 \
duckietown/rpi-simple-server:master18
```

1

2

Pour accéder ensuite à ce répertoire, visitez l'URL suivante : [http://DUCKIEBOT\\_NAME:8082/](http://DUCKIEBOT_NAME:8082/)

#### 5.6.7. Test de la caméra

La commande suivante peut être utilisée pour tester le bon fonctionnement de la caméra. Par défaut, les images seront hébergées sur : [http://DUCKIEBOT\\_NAME:8081/figures/image.jpg](http://DUCKIEBOT_NAME:8081/figures/image.jpg)



```
~$ docker -H DUCKIEBOT_NAME run -d --privileged -v /data:/data -p 8081:8081 \
duckietown/rpi-docker-python-picamera:master18
```

1

2

Comme la plupart des commandes, un shell en Python est fourni pour le confort de l'utilisateur:



```
dt> duckiebot demo --demo_name camera --duckiebot_name DUCKIEBOT_NAME
```

#### 5.6.8. Outils d'interface utilisateur graphique

Pour utiliser les outils d'interface graphique, il faut d'abord autoriser les connexions X entrantes de l'hôte. Sur les hôtes Linux, cela peut se faire en exécutant `xhost +` en dehors

du Docker.<sup>6</sup> Un conteneur avec des plugins ROS GUI communs peut être lancé avec la commande suivante:



```
~$ docker run -it --rm --net host \
--env ROS_MASTER_URI=http://DUCKIEBOT_IP:11311 \
--env ROS_IP=LAPTOP_IP \
--env="DISPLAY" \
--env="QT_X11_NO_MITSHM=1" \
--volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \
duckietown/rpi-gui-tools
```

1  
2  
3  
4  
5  
6  
7

Cette image contient des plugins ROS courants qui peuvent être exécutés dans des environnements graphiques. Un shell est également fourni pour plus de commodité:



```
dt> start_gui_tools DUCKIEBOT_NAME rqt_image_view
```

La commande ci-dessus ouvre un shell ROS qui se connectera au nœud maître ROS de **DUCKIEBOT**. Pour tester le fonctionnement de la connexion ROS, lancez **rosrun tf**.

### 5.6.9. Remote control

Le conteneur suivant lance la démo du joystick (le joystick USB doit être connecté):



```
~$ docker -H DUCKIEBOT_NAME run --privileged --net host -v /data:/data \
duckietown/rpi-duckiebot-joystick-demo:master18
```

1  
2



```
dt> duckiebot demo --demo_name joystick --duckiebot_name DUCKIEBOT_NAME
```



```
dt> duckiebot keyboard_control DUCKIEBOT_NAME
```

### 5.6.10. Camera calibration

Le conteneur suivant va lancer la procédure de calibrage extrinsèque:

---

<sup>6</sup>See [https://wiki.ros.org/docker/Tutorials/GUI#The\\_safer\\_way](https://wiki.ros.org/docker/Tutorials/GUI#The_safer_way) pour une alternative plus sûre.



```
~$ docker -H DUCKIEBOT_NAME run -it --privileged --net host -v /data:/data  
duckietown/rpi-duckiebot-calibration:master18
```

1  
2

Le passage `-v /data:/data` est nécessaire pour que tous les paramètres d'étalement soient préservés. Lorsqu'elles sont placées sur le modèle de calibrage, les commandes suivantes lanceront une séquence de calibrage interactive pour la caméra.



```
dt> duckiebot calibrate_extrinsics DUCKIEBOT_NAME
```



```
dt> duckiebot calibrate_intrinsics DUCKIEBOT_NAME
```

### 5.6.11. Calibrage des roues

Pour calibrer le gain et le trim des moteurs de roue, les commandes suivantes sont nécessaires:



```
dt> duckiebot demo --demo_name base --duckiebot_name DUCKIEBOT_NAME
```



```
~$ rosservice call /DUCKIEBOT_NAME/inverse_kinematics_node/set_gain --GAIN
```

1



```
~$ rosservice call /DUCKIEBOT_NAME/inverse_kinematics_node/set_trim --TRIM
```

1

### 5.6.12. Lane following

Une fois calibré, le couloir suivant la démo peut être lancé comme suit:



```
~$ docker -H DUCKIEBOT_NAME run -it --privileged --net host -v /data:/data  
duckietown/rpi-duckiebot-lanefollowing-demo:master18
```

1  
2



```
dt> duckiebot demo --demo_name lane_following --duckiebot_name DUCKIEBOT_NAME
```

## 5.7. Retrospective

Un des problèmes rencontrés lors du développement de l'infrastructure du Docker de Duckietown était de savoir s'il fallait stocker le code source à l'intérieur ou à l'extérieur

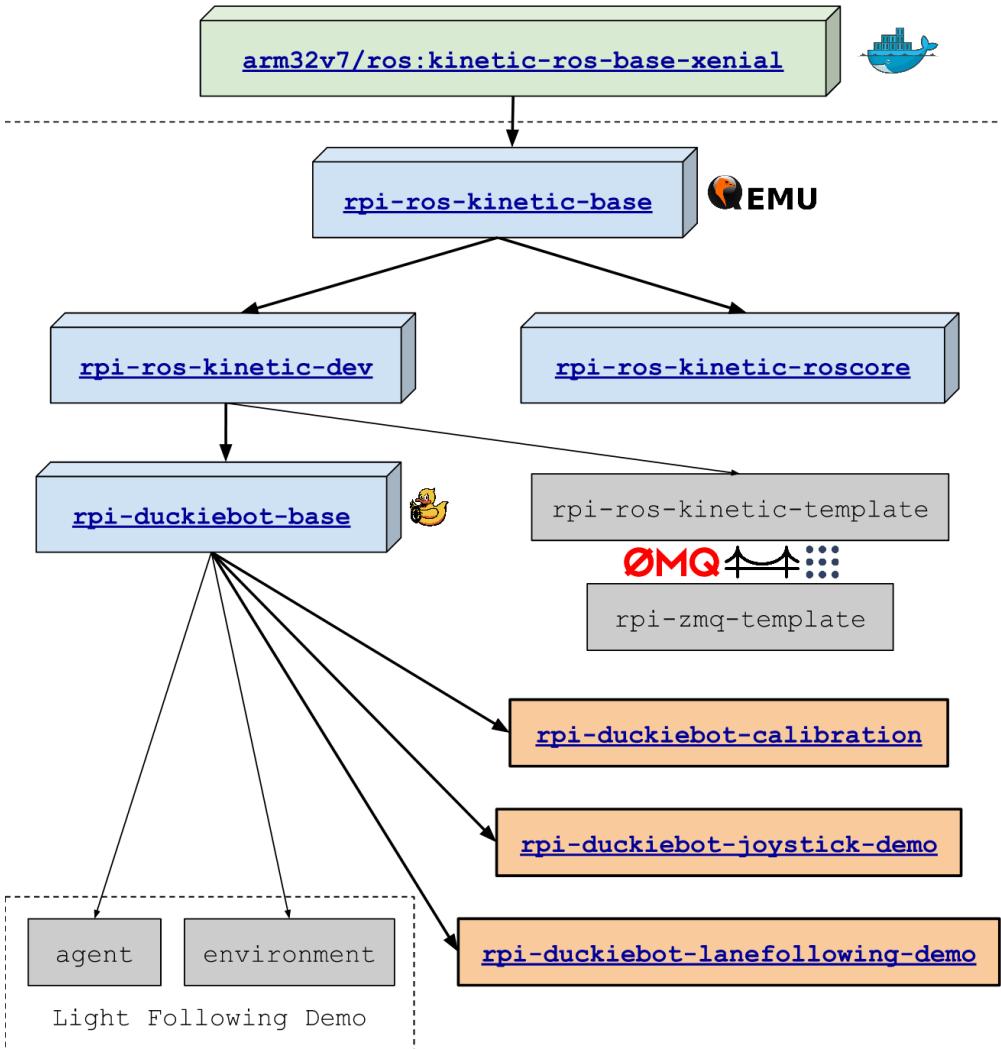
du conteneur (comme décrit par exemple dans § 5.4.4). S'il est stocké à l'extérieur, un développeur peut toujours charger le code source dans un volume partagé et le reconstruire au démarrage du conteneur. Les deux approches peuvent produire des artefacts reproductibles si elles sont correctement versionnées, mais les images Docker se lancent plus rapidement lorsque les images sont entièrement pré-construites et ont tendance à être plus inspectables avec les sources incluses.

Au départ, nous avons pris la décision explicite d'expédier le code source de l'utilisateur directement à l'intérieur de l'image. En conséquence, toute modification du code source déclencherait une reconstruction ultérieure, liant les sources et l'image Docker ensemble. Bien que l'inclusion des sources facilite le dépannage et les diagnostics, elle ajoute également une certaine friction au cours du développement, ce qui a entraîné des problèmes de configuration de l'environnement et du Docker pour les utilisateurs.

La cause profonde de cette friction est le produit d'un versionnage imprécis et d'une sur-automatisation. Comme les balises de version ont été initialement omises, toutes les images ont été extraites et construites à partir du dernier commit sur la branche de développement principale. La fonction de construction automatique du serveur de CI a entraîné des modifications en amont pour la cascade d'images en aval. Notre solution à court terme a été de désactiver la construction automatique et de pousser manuellement les constructions locales vers le serveur, mais la correction a nécessité de repenser le rôle du versionnage et du test des constructions Docker dans la chaîne d'outils CI.

Une solution plus stable consiste à stocker toutes les sources sur l'environnement de développement local et à ne reconstruire l'image que lorsque ses dépendances en amont changent. L'image ne contient que ses dépendances en amont compilées et n'est couplée avec le code source qu'au moment de l'exécution.

L'un des principaux cas d'utilisation de l'infrastructure de conteneurs de Duckietown est une compétition biennuelle de robotique autonome appelée AI Driving Olympics [Zilly et al., 2019] (AIDO). Pour participer, les concurrents doivent soumettre une image Docker (différents modèles sont fournis pour [apprentissage de renforcement](#), [apprentissage de simulation](#) et [robotique classique](#)). L'image soumise, avec un dépôt Git et un hachage de commit, constitue une soumission AIDO. La soumission est récupérée par les organisateurs et évaluée sur une carte aléatoire dans le [simulateur](#) [Chevalier-Boisvert et al., 2018] de Duckietown.



**Fig. 5.5.** Premier prototype de la hiérarchie d’images de Docker. L’enchaînement de constructions automatiques non versionnées sans essais unitaires disciplinés crée un effet domino potentiel qui permet aux changements de se propager en aval, entraînant une cascade de défaillances silencieuses.

Cette évaluation produit un score numérique dans plusieurs catégories. Les soumissions valides peuvent également être effectuées sur un *robotarium* physique. Les soumissions les mieux classées sont évaluées lors d’un tour final au NeurIPS et à l’ICRA.



**Fig. 5.6.** The [AI Driving Olympics](#), un cas d'utilisation primaire pour le système décrit ci-dessus.

### 5.7.1. Remarques sur la sécurité

Un défaut technique regrettable du système Docker est sa dépendance aux priviléges des super-utilisateurs. Bien que Docker prenne toute une série de mesures préventives pour s'assurer que les habitants des conteneurs ne puissent pas obtenir des priviléges accrus, de nombreuses attaques par évasion ont été découvertes [Martin et al., 2018] dans la nature. Tout processus pouvant contourner la sécurité des conteneurs obtient un accès sans entrave au système d'exploitation hôte, ce qui rend Docker particulièrement inadapté au déploiement dans les environnements de cloud computing, de grille et de cluster computing.

En outre, Docker fournit un mécanisme permettant de contourner ses propres mesures de sécurité, permettant aux applications du conteneur de s'exécuter comme si elles étaient des processus racine sur le système d'exploitation hôte : le drapeau

Les priviléges inutilement élevés de Docker et sa vulnérabilité aux abus sont des problèmes sérieux. Bien que les erreurs de l'opérateur puissent être en partie responsables, ces vulnérabilités sont principalement le résultat de mauvais choix de mise en uvre. La violation flagrante par Docker du principe du moindre privilège [Saltzer and Schroeder, 1975] compromet effectivement l'ensemble du modèle de sécurité de Linux.

Pour résoudre ces problèmes, diverses plates-formes de conteneurs, dont [Shifter](#) [Gerhardt et al., 2017] et [Singularity](#) [Kurtzer et al., 2017], ont émergé et gagné en popularité dans la communauté du calcul scientifique, en raison de leurs priviléges moindres et de leur compatibilité avec les distributions Linux héritées utilisées par de nombreux environnements informatiques universitaires. Depuis lors, Docker a également introduit un [mode sans racine](#), mais il reste expérimental au moment de la rédaction de cette thèse.

## 5.8. Travaux futurs

Duckietown encourage les utilisateurs à former des modèles d'apprentissage de renforcement à l'intérieur d'un simulateur de conduite. Comme les agents apprennent une politique pour conduire un Duckiebot, nous envisageons qu'il est également possible de former un agent à effectuer des tâches dans l'environnement Docker. Les agents, dotés de commandes shell rudimentaires, recevraient une récompense basée sur le code de sortie d'un programme souhaité que nous souhaitons exécuter. Cela peut être étendu à un environnement entièrement automatisé, où l'agent a accès à un clavier et une souris virtuels et apprend à configurer un environnement de bureau pour exécuter un programme souhaité. Actuellement, ce processus implique qu'un étudiant diplômé essaie diverses commandes de StackOverflow. Il est évident que le même résultat peut être obtenu par tout processus stochastique qui sélectionne des commandes à partir d'une base de connaissances et qui apprend de l'expérience passée. Des travaux préliminaires dans ce domaine sont déjà en cours [Henkel et al., 2020], probablement par un étudiant diplômé dans une situation similaire.

## 5.9. Conclusion

Dans ce chapitre, nous avons fait une visite guidée du processus de conteneurisation et démontré l'efficacité des conteneurs pour la construction de logiciels de robotique reproductibles - une étape clé dans la quête plus large de la reproductibilité expérimentale. Nous proposons un ensemble de meilleures pratiques et de leçons apprises lors de la conception, du développement et du déploiement des conteneurs Docker pour la plate-forme Duckietown [Paull et al., 2017]. Nous recommandons également un certain nombre d'outils et de techniques pour la reproductibilité des logiciels, un élément clé dans la quête plus large de la reproductibilité de la recherche. L'auteur tient à remercier Rusi Hristov pour son assistance technique inestimable au cours des premières étapes de ce projet et Florian Golemo pour la planification et l'assistance architecturale. Pour plus d'informations sur la plate-forme Duckietown et le développement de logiciels reproductibles à l'aide de Docker, veuillez consulter le site <https://docs.duckietown.org>



# Chapter 6

---

## Conclusion

“Nous sommes tous façonnés par les outils que nous utilisons, en particulier : les formalismes que nous utilisons façonnent nos habitudes de pensée, pour le meilleur ou pour le pire, et cela signifie que nous devons être très prudents dans le choix de ce que nous apprenons et enseignons, car le désapprentissage n'est pas vraiment possible.”

---

—Edsger W. Dijkstra [2000], *Réponses aux questions des étudiants en génie logiciel*

Dans cette thèse, nous avons exploré quatre outils de programmation différents issus du génie logiciel pour le développement de systèmes intelligents, en abordant de manière générale la complexité cognitive apparaissant dans les quatre phases de la méthode de la cascade de Royce. Ces outils ont des degrés variables de praticabilité, allant de très théoriques (par exemple l'essai contradictoire de programmes différenciables Chapter 4) à plus pragmatiques (par exemple la conteneurisation Chapter 5). Dans chaque chapitre, nous fournissons quelques exemples motivants qui démontrent les principales lacunes des outils de programmation de pointe pour les systèmes intelligents et nous proposons des solutions viables qui remédient à quelques-unes de ces lacunes. Bien que nous espérons que les programmeurs de systèmes intelligents (par exemple les roboticiens et les praticiens de l'apprentissage machine) puissent tirer une certaine valeur des outils eux-mêmes, notre intention est d'être *illustratif* plutôt que *prescriptif*.

En construisant des outils et en validant leur efficacité sur des applications de jouets, nous espérons que les développeurs d'outils examineront attentivement comment les outils logiciels peuvent introduire et atténuer la complexité cognitive. Des outils bien conçus peuvent augmenter la capacité cognitive des humains à raisonner sur des faits en présence d'incertitude [Famelis et al., 2012], et fournir une assistance ergonomique de débogage et de visualisation (par exemple Chapter 2). Nous espérons également faire comprendre

l’importance de la conception notationnelle. Une bonne notation oblige les auteurs à réfléchir soigneusement à leurs abstractions, fait ressortir les erreurs logiques et les aide à comprendre les implications des premiers choix de conception. Nous espérons que les outils de programmation présentés dans cette thèse inciteront les développeurs à réimaginer le potentiel de la programmation assistée par ordinateur dans la conception de logiciels pour les systèmes intelligents.

En complétant les capacités cognitives des programmeurs humains – qui excellent dans la résolution créative de problèmes et le raisonnement abstrait de haut niveau – avec les capacités de traitement symbolique brut des outils de programmation, nous pouvons accélérer la conception [Chapter 2](#), le développement [Chapter 3](#), la validation [Chapter 4](#) et le déploiement [Chapter 5](#) de systèmes intelligents dans des applications du monde réel. Ce processus est un cycle vertueux qui mérite des outils et des pratiques spécifiques à chaque domaine en raison des possibilités offertes par les systèmes intelligents et de l’interaction unique entre l’intelligence humaine et l’intelligence des machines. Alors que nous commençons à développer des systèmes autonomes qui jouent un rôle de plus en plus actif dans la société, tant les ingénieurs en logiciel que les praticiens de l’apprentissage machine doivent jouer un rôle tout aussi actif dans le façonnage du comportement de ces systèmes.

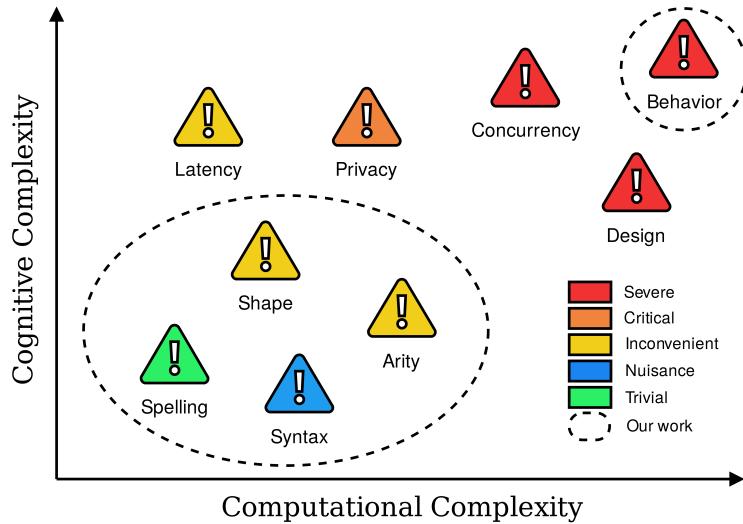
Les ingénieurs en logiciel ont un certain nombre de leçons à tirer des systèmes intelligents. Les concepteurs de langage feraient bien de considérer la valeur des outils de développement intelligents ([Chapter 2](#)) pour faciliter le dialogue entre l’intelligence humaine et l’intelligence des machines. Les langages devraient s’efforcer d’intégrer les connaissances humaines par le biais de systèmes de programmation et de systèmes experts différenciables, afin d’aider à raisonner sur la composition et la justesse de la description ([Chapter 3](#)). Des tests automatisés via des simulateurs et des cadres de tests de propriétés sont nécessaires pour raisonner sur l’exactitude opérationnelle sans spécification exhaustive ([Chapter 4](#)). Enfin, l’intégration continue, les tests automatisés et les meilleures pratiques dans les opérations des développeurs ([Chapter 5](#)) sont nécessaires pour garantir la reproductibilité des artefacts en présence de variabilité logicielle et matérielle.

Les praticiens de l’apprentissage machine ont également un certain nombre de leçons à tirer du génie logiciel. Le génie logiciel traditionnel prescrit un modèle de processus et une méthodologie de test rigoureux qui ont guidé de nombreuses générations de projets

logiciels. Pour devenir une véritable discipline d'ingénierie, l'apprentissage machine devra adopter une approche plus systématique de la construction de systèmes autonomes. Les modèles d'apprentissage machine sont formés sur des *fonctions-objectives*, qui sont généralement des fonctions à faible dimension mesurant les performances d'un système et renvoyant une valeur scalaire connue sous le nom de *erreur* ou *perte*. En pratique, les systèmes intelligents doivent satisfaire à un ensemble de critères multiobjectifs [?], notamment l'efficacité énergétique [Paull et al., 2010], la mémoire [Mitliagkas et al., 2013], re/usability [Breuleux and van Merriënboer, 2017, Deleu et al., 2019], predictability [Turner and Neal, 2017], latency [Ravanelli et al., 2018], robustness [Pineau et al., 2003], reproductibilité [Pineau et al., 2020], explicabilité [Turner, 2016], traçabilité [Guo et al., 2017, Tsirigotis et al., 2018], incertitude [Diaz Cabrera, 2018], simplicité [Kastner et al., 2019], fiabilité [Xu, 2017], transférabilité [Mehta et al., 2019], évolutivité [Luan et al., 2019] et bien d'autres facteurs.

Dans le domaine du génie logiciel traditionnel, il est raisonnable de supposer que ceux qui mettent en uvre un nouveau système ont une certaine connaissance implicite du domaine et sont des êtres humains bien intentionnés travaillant à un objectif commun. Si on leur donne une description grossière, ils peuvent remplir les blancs. Lors de la construction d'un système intelligent, il serait plus sûr de supposer que les exigences sont mises en uvre par un génie. Compte tenu de certaines données et d'une métrique d'optimisation, il faudra prendre tous les raccourcis possibles pour exaucer nos souhaits. Si nous ne prenons pas soin d'énoncer nos exigences, cette entité produira une solution qui ne fonctionne tout simplement pas (dans le meilleur des cas), ou qui semble fonctionner mais qui est vraiment maudite [Bellman, 1957].

Lors de la construction d'un système intelligent, les développeurs doivent soigneusement se demander : "Quel est le comportement souhaité du système que nous concevons ? Cette question est souvent très gênante, car nos exigences approximatives doivent être traduites en contraintes précises sur l'espace de la solution. Par exemple, lors de la conception d'un véhicule autotracé, nous devons clairement optimiser la sécurité des passagers, mais en faisant cela, nous formerons un véhicule qui ne bouge jamais ou qui cède toujours aux véhicules qui passent. À défaut d'une spécification exhaustive, comment pouvons-nous être sûrs que le système qui en résultera répondra à nos exigences ? La plupart des êtres humains sont capables de conduire un véhicule en toute sécurité, mais même les meilleurs ingénieurs



**Fig. 6.1.** Complexité de la détection de divers types d’erreurs de programmation.

ont du mal à écrire un algorithme de conduite. L’étiquetage des données à la main est trop coûteux. La vérification formelle est tout simplement impossible.

Les systèmes de types, les compilateurs et les fuzzers font tous partie d’une catégorie plus large d’outils de validation et de vérification. L’objectif de ces outils est d’échanger la complexité cognitive contre la complexité informatique. Certaines erreurs (par exemple, les erreurs syntaxiques), sont des nuisances mineures et peuvent être détectées avec un bon analyseur incrémental (??). D’autres, comme indiqué dans Fig. 6.1, présentent une complexité cognitive plus élevée mais peuvent être détectées par un calcul de dépenses. Nous soutenons que ce coût de calcul est souvent justifié car le calcul est bon marché et les bogues peuvent avoir des conséquences catastrophiques. Des études montrent que plus les bogues sont détectés tôt, plus ils ont de chances d’être corrigés [Distefano et al., 2019] – gagner des minutes de développement pourrait sauver des vies pendant l’exploitation. Le calcul des dépenses permet également de libérer de précieuses ressources cognitives pour d’autres tâches.

Les tests Fuzz restent une alternative économique et informatique efficace à la vérification formelle. Comme le montre § 4.2, nous pouvons détecter des erreurs plus graves avec un budget fiscal et de calcul plus faible en faisant quelques hypothèses pratiques sur le modèle et l’oracle. Alors que les ingénieurs d’aujourd’hui commencent à ajouter des capacités d’apprentissage aux systèmes robotiques critiques de demain, nous pensons que l’assurance

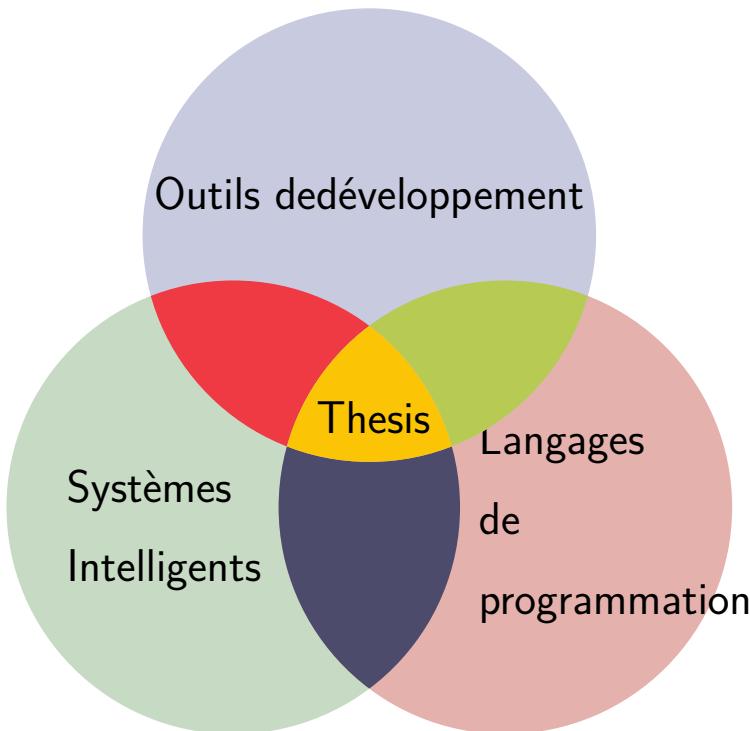
accrue que fournissent les outils intelligents de validation et de vérification sera indispensable pour la mise à l'échelle de ces systèmes cyberphysiques adaptatifs complexes.

Il reste beaucoup de travail à accomplir pour le lecteur intéressé. Une grande partie du travail dans le domaine de l'apprentissage machine consiste à concevoir des représentations adaptées aux tâches en aval et des fonctions de perte qui mesurent avec précision les performances de ces tâches. La construction de représentations et de fonctions de perte qui capturent la gamme complète des objectifs peut être un processus de débogage laborieux. Nous encourageons les ingénieurs à réfléchir attentivement au processus de débogage des modèles d'apprentissage machine et à la manière dont nous pouvons accélérer le cycle de vie, de l'exploration et de l'analyse des données à l'évaluation et au déploiement.

Les chercheurs en apprentissage machine feraient bien de considérer la valeur de la sémantique dénotationnelle pour la mise à la terre et le raisonnement sur les spécifications. Alors que les outils de vérification de la théorie des types sont actuellement limités à des propriétés simples, leurs abstractions sont très puissantes. Qu'il s'agisse de systèmes de type ou de systèmes experts, les outils de raisonnement assisté par ordinateur joueront un rôle important dans le développement de systèmes intelligents sûrs. Nous encourageons le lecteur à examiner attentivement la valeur de ces systèmes et, lorsqu'ils ne sont pas adaptés, à envisager d'utiliser des techniques de vérification des propriétés et des méthodes d'intégration continue pour garantir l'exactitude fonctionnelle.

## 6.1. Contributions

Il existe de nombreux problèmes de codage intéressants à l'intersection des outils, des langages et des systèmes (Fig. 6.2). Dans ce travail, nous examinons la théorie et la mise en œuvre des outils de programmation pour les systèmes intelligents. La direction opposée est également un sujet intrigant, mais reste en dehors de la portée de cette thèse. Les concepteurs de langages ont récemment commencé à explorer la signification des langages "outillables" et des langages améliorés par des outils [Chatley et al., 2019]. La recherche en programmation orientée langage [Dmitriev, 2004] et en ingénierie dirigée par les modèles [Famelis et al., 2015] a également envisagé des outils pour les concepteurs d'API et de PL. Les ingénieurs en logiciel ont étudié un certain nombre d'outils pour les systèmes intelligents, notamment



**Fig. 6.2.** Many interesting applications lie at the intersection of these three fields.

les ordinateurs portables [Chattopadhyay et al.], les REPL et les environnements de programmation interactifs. Enfin, les langages et les systèmes intelligents ont bénéficié d'une collaboration fructueuse en matière de programmation différentiable et probabiliste (§ 3.2). Chacun de ces domaines constituerait une thèse intéressante en soi.

Nos contributions à cette thèse particulière sont au nombre de quatre. Dans [Chapter 2](#), nous présentons un nouveau plugin pour la plateforme IntelliJ, un environnement de développement intégré avec support du système d'exploitation du robot. En plus des cadres de travail axés sur les applications comme le ROS, plusieurs langages spécifiques à un domaine pour les systèmes intelligents ont récemment fait leur apparition (§ 3.2). Dans [Chapter 3](#), nous en introduisons un autre, un DSL intégré dans le langage Kotlin permettant aux utilisateurs d'écrire des programmes différentiables sans risque de forme dans une notation mathématique idiomatique.

La reproductibilité est un vaste défi dans la conception de systèmes intelligents, nécessitant une solution à plusieurs volets. Nous pensons que les tests et la validation des systèmes intelligents joueront un rôle important dans les applications critiques pour la sécurité. Des tests et des simulations automatisés, ainsi que des outils de construction et de déploiement

reproductibles seront essentiels pour la robustesse. Dans [Chapter 4](#), nous introduisons un algorithme de test général basé sur les propriétés et montrons empiriquement une amélioration de l'efficacité des données en détectant une plus grande proportion d'erreurs dans un budget de calcul fixe. Enfin, dans [Chapter 5](#), nous présentons un environnement de construction entièrement conteneurisé et un flux de travail d'intégration continu, améliorant la réutilisation et la reproductibilité des applications logicielles sur la plateforme Duckietown. Ensemble, ces contributions contribuent à réduire la complexité cognitive lors de la conception, du développement, des tests et du déploiement de systèmes intelligents.



## References

---

- Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL <https://dl.acm.org/citation.cfm?id=3026877.3026899>.
- Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, 2012.
- Peter Abeles. Efficient Java Matrix Library, 2010. URL <http://ejml.org/>.
- Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2<sup>nd</sup> edition, 1996. ISBN 0262011530.
- Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize Halide with tree search and random programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, July 2019. ISSN 0730-0301. doi: 10.1145/3306346.3322967. URL <https://doi.acm.org/10.1145/3306346.3322967>.
- Ashish Agarwal. Static automatic batching in TensorFlow. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36<sup>th</sup> International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 92–101, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL <http://proceedings.mlr.press/v97/agarwal19a.html>.

Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shangqing Cai. TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning. In Proceedings of the 2<sup>nd</sup> SysML Conference, 2019. URL <https://www.sysml.cc/doc/2019/88.pdf>.

Isabela Albuquerque, Joao Monteiro, Thang Doan, Breandan Considine, Tiago Falk, and Ioannis Mitliagkas. Multi-objective training of Generative Adversarial Networks with multiple discriminators. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, Proceedings of the 36<sup>th</sup> International Conference on Machine Learning, volume 97 of Proceedings of Machine Learning Research, pages 202–211, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL <http://proceedings.mlr.press/v97/albuquerque19a.html>.

Mohammed AlQuraishi. End-to-end differentiable learning of protein structure. bioRxiv, 2018. doi: 10.1101/265231. URL <https://www.biorxiv.org/content/early/2018/08/29/265231>.

Mario Alvarez-Picallo and C.-H. Luke Ong. Change actions: Models of generalised differentiation. volume abs/1902.05465, 2019. URL <https://arxiv.org/abs/1902.05465>.

Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. Fixing incremental computation: Derivatives of fixpoints, and the recursive semantics of Datalog. volume abs/1811.06069, 2018. URL <https://arxiv.org/abs/1811.06069>.

Nada Amin and Ross Tate. Java and Scala’s type systems are unsound: The existential crisis of null pointers. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, pages 838–848, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984004. URL <https://doi.acm.org/10.1145/2983990.2984004>.

Antolino Andrea and Luc Maisonobe. Automatic differentiation for propagation of orbit uncertainties on Orekit. 2016. URL <https://www.orekit.org/doc/Antolino-2016-automatic-diff-for-prop-of-orbit-uncertainties.pdf>.

Andrew W. Appel and Jens Palsberg. Modern Compiler Implementation in Java. Cambridge University Press, USA, 2nd edition, 2003. ISBN 052182060X.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein Generative Adversarial Networks. In Doina Precup and Yee Whye Teh, editors, Proceedings of the 34<sup>th</sup>

- International Conference on Machine Learning, volume 70 of *Proceedings of Machine Learning Research*, pages 214–223, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL <http://proceedings.mlr.press/v70/arjovsky17a.html>.
- John Backus. *Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs*, volume 21. ACM, New York, NY, USA, August 1978. doi: 10.1145/359576.359579. URL <https://doi.acm.org/10.1145/359576.359579>.
- Junjie Bai, Fang Lu, Ke Zhang, et al. ONNX: Open Neural Network Exchange, 2019. URL <https://github.com/ONNX/ONNX>.
- A. G. Baydin and B. A. Pearlmutter. Automatic differentiation of algorithms for machine learning. In *Proceedings of the AutoML Workshop at the International Conference on Machine Learning (ICML)*, Beijing, China, June 21–26, 2014, 2014. URL <https://arxiv.org/abs/1404.7456>.
- Atilim Güneş Baydin. Differentiable programming. URL <https://www.cs.nuim.ie/~gunes/files/Baydin-MSR-Slides-20160201.pdf>.
- Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767, 2015a. URL <https://arxiv.org/abs/1502.05767>.
- Atilim Gunes Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. DiffSharp: Automatic differentiation library. *CoRR*, abs/1511.07727, 2015b. URL <https://arxiv.org/abs/1511.07727>.
- Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.
- R. E. Bellman, H. Kagiwada, and R. E. Kalaba. Wengert’s numerical method for partial derivatives, orbit determination and quasilinearization. *Commun. ACM*, 8(4):231–232, April 1965. ISSN 0001-0782. doi: 10.1145/363831.364886. URL <https://doi.acm.org/10.1145/363831.364886>.
- Richard Bellman. *Dynamic Programming*. Princeton, NJ, USA, 1957. URL <https://www.gwern.net/docs/statistics/decision/1957-bellman-dynamicprogramming.pdf>.
- Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2):157–166, March 1994. ISSN 1045-9227. doi: 10.1109/72.279181. URL <https://doi.org/10.1109/72.279181>.

- James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, et al. Theano: a CPU and GPU math expression compiler. In Proceedings of the Python for scientific computing conference (SciPy), volume 4. Austin, TX, 2010. URL <http://deeplearning.net/software/theano/>.
- G. Berry and R. Sethi. From regular expressions to deterministic automata. Theor. Comput. Sci., 48(1):117–126, December 1986. ISSN 0304-3975. URL <https://dl.acm.org/citation.cfm?id=39528.39537>.
- Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In European Conference on Object-Oriented Programming, pages 257–281. Springer, 2014.
- Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. Journal of Machine Learning Research, 20(28):1–6, 2019. URL <http://jmlr.org/papers/v20/18-403.html>.
- Mathieu Blondel, Olivier Teboul, Quentin Berthet, and Josip Djolonga. Fast differentiable sorting and ranking, 2020. URL <https://arxiv.org/pdf/2002.08871.pdf>.
- Richard F Blute, J Robin B Cockett, and Robert AG Seely. Differential categories. Mathematical structures in computer science, 16(6):1049–1083, 2006.
- Richard F Blute, J Robin B Cockett, and Robert AG Seely. Cartesian differential categories. Theory and Applications of Categories, 22(23):622–672, 2009.
- Yang Bo. DeepLearning.scala: A simple library for creating complex neural networks. 2018. URL <https://github.com/ThoughtWorksInc/DeepLearning.scala>.
- Konstantinos Bousmalis, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor Sampedro, Kurt Konolige, Sergey Levine, and Vincent Vanhoucke. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. 2018. URL <https://arxiv.org/abs/1709.07857>.
- Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What’s decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, Verification, Model Checking, and Abstract Interpretation, pages 427–442, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-31622-0. URL <http://theory.stanford.edu/~arbrad/papers/arrays.pdf>.

- Mikio L Braun, Johannes Schaback, Matthias L Jugel, Nicolas Oury, et al. jBlas: Linear algebra for Java, 2011. URL <http://jblas.org/>.
- Olivier Breuleux and Bart van Merriënboer. Automatic differentiation in myia. 2017. URL <https://github.com/mila-udem/myia>.
- Aloïs Brunel, Damiano Mazza, and Michele Pagani. Backpropagation in the Simply Typed Lambda-calculus with Linear negation. 2020. URL <https://arxiv.org/abs/1909.13768>.
- Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964. ISSN 0004-5411. doi: 10.1145/321239.321249. URL <https://doi.acm.org/10.1145/321239.321249>.
- Roman V Buniy, Stephen DH Hsu, and Anthony Zee. Is Hilbert space discrete? *Physics Letters B*, 630(1-2):68–72, 2005. URL <https://doi.org/10.1016/j.physletb.2005.09.084>.
- Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing  $\lambda$ -calculi by static differentiation. In Proceedings of the 35<sup>th</sup> ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, pages 145–155, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594304. URL <https://doi.acm.org/10.1145/2594291.2594304>.
- Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. Artificial intelligence, 134(1-2):57–83, 2002. URL <https://core.ac.uk/download/pdf/82416379.pdf>.
- Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA ’89, pages 273–280, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0. doi: 10.1145/99370.99392. URL <https://doi.acm.org/10.1145/99370.99392>.
- Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of System F with subtyping. volume 109, pages 4–56, Duluth, MN, USA, February 1994. Academic Press, Inc. doi: 10.1006/inco.1994.1013. URL <https://dx.doi.org/10.1006/inco.1994.1013>.
- Pascal Caron, Jean-Marc Champarnaud, and Ludovic Mignot. Partial derivatives of an extended regular expression. pages 179–191, 2011.

Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1), 1 2017. doi: 10.18637/jss.v076.i01. URL <https://www.osti.gov/servlets/purl/1430202>.

Andrea Censi. A mathematical theory of co-design. *arXiv preprint arXiv:1512.08055*, 2015. URL <https://arxiv.org/pdf/1512.08055.pdf>.

J-M Champarnaud, J-L Ponty, and Djelloul Ziadi. From regular expressions to finite automata. *International journal of computer mathematics*, 72(4):415–431, 1999. URL <https://doi.org/10.1080/00207169908804865>.

Kartik Chandra, Erik Meijer, Samantha Andow, Emilio Arroyo-Fang, Irene Dea, Johann George, Melissa Grueter, Basil Hosmer, Steffi Stumpos, Alanna Tempest, and Shannon Yang. Gradient descent: The ultimate optimizer, 2019. URL <https://arxiv.org/pdf/1909.13371.pdf>.

Émilie Charlier, Narad Rampersad, and Jeffrey Shallit. Enumeration and decidable properties of automatic sequences. *Lecture Notes in Computer Science*, page 165179, 2011. ISSN 1611-3349. doi: 10.1007/978-3-642-22321-1\_15. URL [https://dx.doi.org/10.1007/978-3-642-22321-1\\_15](https://dx.doi.org/10.1007/978-3-642-22321-1_15).

Robert Chatley, Alastair Donaldson, and Alan Mycroft. The next 7000 programming languages. In *Computing and Software Science*, pages 250–282. Springer, 2019. URL [https://link.springer.com/chapter/10.1007/978-3-319-91908-9\\_15](https://link.springer.com/chapter/10.1007/978-3-319-91908-9_15).

Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. Whats wrong with computational notebooks? pain points, needs, and design opportunities. URL [http://web.eecs.utk.edu/~azh/pubs/Chattopadhyay2020CHI\\_NotebookPainpoints.pdf](http://web.eecs.utk.edu/~azh/pubs/Chattopadhyay2020CHI_NotebookPainpoints.pdf).

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13<sup>th</sup> USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL <https://www.usenix.org/conference/osdi18/presentation/chen>.

Tongfei Chen. Typesafe abstractions for tensor operations (short paper). pages 45–50, 2017. doi: 10.1145/3136000.3136001. URL <https://doi.acm.org/10.1145/3136000.3136001>.

- Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Department of Computer Science, Hong Kong, 1998. URL <https://www.cse.ust.hk/~scc/publ/CS98-01-metamorphictesting.pdf>.
- Yan Chen, Joshua Dunfield, and Umut A. Acar. Type-directed automatic incrementalization. pages 299–310, 2012. doi: 10.1145/2254064.2254100. URL <https://doi.acm.org/10.1145/2254064.2254100>.
- Maxime Chevalier-Boisvert, Florian Golemo, Yanjun Cao, Bhairav Mehta, and Liam Paull. Duckietown environments for OpenAI Gym. <https://github.com/duckietown/gym-duckietown>, 2018.
- Bruce Christianson. A Leibniz notation for automatic differentiation. In Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther, editors, Recent Advances in Algorithmic Differentiation, volume 87 of Lecture Notes in Computational Science and Engineering, pages 1–9. Springer, Berlin, 2012. ISBN 978-3-540-68935-5. doi: 10.1007/978-3-642-30023-3\_1. URL <https://uhra.herts.ac.uk/bitstream/handle/2299/8933/904722.pdf>.
- Alonzo Church. The Calculi of Lambda-conversion. Annals of Mathematics Studies. Princeton University Press, 1941. ISBN 9780691083940. URL <https://books.google.ca/books?id=yWCYDwAAQBAJ>.
- Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. pages 268–279, 2000. doi: 10.1145/351240.351266. URL <https://doi.acm.org/10.1145/351240.351266>.
- James Clift and Daniel Murfet. Derivatives of Turing machines in Linear Logic. arXiv preprint arXiv:1805.11813, 2018. URL <https://arxiv.org/abs/1805.11813>.
- Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Idiap-RR Idiap-RR-46-2002, IDIAP, 2002.
- George F Corliss and Andreas Griewank. Operator overloading as an enabling technology for automatic differentiation. Technical report, Argonne National Laboratory, 1993. URL <http://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR93431.pdf>.
- Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Assessing the bus factor of Git repositories. In 2015 IEEE 22<sup>nd</sup> International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 499–503. IEEE, 2015. URL <https://hal>.

[inria.fr/hal-01257471/document](https://hal.inria.fr/hal-01257471/document).

Sébastien Crozet et al. nalgebra: a linear algebra library for Rust, 2019. URL <https://nalgebra.org>.

H.B. Curry and R. Feys. Combinatory Logic. Number v. 1 in Combinatory Logic. North-Holland Publishing Company, 1958. URL <https://books.google.ca/books?id=fEnuAAAAMAAJ>.

Marco Cuturi, Olivier Teboul, and Jean-Philippe Vert. Differentiable ranking and sorting using optimal transport. In Advances in Neural Information Processing Systems, pages 6858–6868, 2019. URL <https://papers.nips.cc/paper/8910-differentiable-ranking-and-sorting-using-optimal-transport.pdf>.

Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel nGraph: An intermediate representation, compiler, and executor for deep learning. CoRR, abs/1801.08058, 2018. URL <https://arxiv.org/abs/1801.08058>.

Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In Advances in Neural Information Processing Systems, pages 7178–7189, 2018.

Jonas Degrave, Michiel Hermans, Joni Dambre, and Francis Wyffels. A differentiable physics engine for deep learning in robotics. CoRR, abs/1611.01652, 2016. URL <https://arxiv.org/abs/1611.01652>.

T. J. Dekker. A floating-point technique for extending the available precision. Numer. Math., 18(3):224–242, June 1971. ISSN 0029-599X. doi: 10.1007/BF01397083. URL <https://dx.doi.org/10.1007/BF01397083>.

Tristan Deleu, Tobias Würfl, Mandana Samiei, Joseph Paul Cohen, and Yoshua Bengio. Torchmeta: A meta-learning library for PyTorch, 2019. URL <https://arxiv.org/pdf/1909.06576.pdf>.

Commons Math Developers. Apache Commons Math. Forest Hill, MD, USA: The Apache Software Foundation, 2012. URL <https://commons.apache.org/proper/commons-math/>.

Manfred Ramon Diaz Cabrera. Interactive and Uncertainty-aware Imitation Learning: Theory and Applications. PhD thesis, Concordia University, 2018. URL [https://spectrum.library.concordia.ca/984373/1/Diaz\\_MSc\\_F2018.pdf](https://spectrum.library.concordia.ca/984373/1/Diaz_MSc_F2018.pdf).

Edsger W Dijkstra. Answers to questions from students of software engineering. Circulated privately, 2000. URL <http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1305.PDF>.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O’Hearn. Scaling static analyses at Facebook. Commun. ACM, 62(8):62–70, July 2019. ISSN 0001-0782. doi: 10.1145/3338112. URL <https://doi.acm.org/10.1145/3338112>.

Sergey Dmitriev. Language oriented programming: The next programming paradigm. JetBrains onBoard, 1(2):1–13, 2004. URL <http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf>.

Chris Donahue, Julian McAuley, and Miller Puckette. Adversarial audio synthesis. In ICLR, 2019. URL <https://github.com/chrisdonahue/wavegan>.

Stuart E Dreyfus. Artificial neural networks, back propagation, and the Kelley-Bryson gradient procedure. Journal of guidance, control, and dynamics, 13(5):926–928, 1990. URL <https://arc.aiaa.org/doi/abs/10.2514/3.25422>.

Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Jianjun Zhao, and Yang Liu. Deepcruiser: Automated guided testing for stateful deep learning systems. CoRR, abs/1812.05339, 2018. URL <https://arxiv.org/abs/1812.05339>.

Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. arXiv preprint arXiv:1603.07285, 2016. URL <https://arxiv.org/abs/1603.07285>.

Paul S Dwyer, MS MacPhail, et al. Symbolic matrix derivatives. The annals of mathematical statistics, 19(4):517–534, 1948. URL <https://www.jstor.org/stable/2236019>.

A. Edalat and A. Lieutier. Domain theory and differential calculus (functions of one variable). In Proceedings 17<sup>th</sup> Annual IEEE Symposium on Logic in Computer Science, pages 277–286, July 2002. doi: 10.1109/LICS.2002.1029836.

Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. CoRR, abs/1606.01642, 2016. URL <https://arxiv.org/abs/1606.01642>.

Thomas Ehrhard and Laurent Regnier. The differential  $\lambda$ -calculus. Theor. Comput. Sci., 309(1):1–41, December 2003. ISSN 0304-3975. doi: 10.1016/S0304-3975(03)00392-X. URL [https://dx.doi.org/10.1016/S0304-3975\(03\)00392-X](https://dx.doi.org/10.1016/S0304-3975(03)00392-X).

- Conal Elliott. The simple essence of automatic differentiation. *Proc. ACM Program. Lang.*, 2(ICFP):70:1–70:29, July 2018. ISSN 2475-1421. doi: 10.1145/3236765. URL <https://doi.acm.org/10.1145/3236765>.
- Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003. URL <http://conal.net/papers/jfp-saig/>.
- Conal M. Elliott. Beautiful differentiation. In *Proceedings of the 14<sup>th</sup> ACM SIGPLAN International Conference on Functional Programming*, ICFP ’09, pages 191–202, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: 10.1145/1596550.1596579. URL <https://doi.acm.org/10.1145/1596550.1596579>.
- Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *2015 IEEE international conference on robotics and automation (ICRA)*, pages 4397–4404. IEEE, 2015. URL [https://www.researchgate.net/profile/Yuval\\_Tassa/publication/280944465\\_Simulation\\_tools\\_for\\_model-based\\_robots\\_Comparison\\_of\\_Bullet\\_Havok\\_MuJoCo\\_ODE\\_and\\_PhysX/links/55cdd4bb08aee19936f8078c.pdf](https://www.researchgate.net/profile/Yuval_Tassa/publication/280944465_Simulation_tools_for_model-based_robots_Comparison_of_Bullet_Havok_MuJoCo_ODE_and_PhysX/links/55cdd4bb08aee19936f8078c.pdf).
- Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA ’10, pages 307–309, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1. doi: 10.1145/1869542.1869625. URL <https://doi.acm.org/10.1145/1869542.1869625>.
- Michalis Famelis, Rick Salay, and Marsha Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 573–583. IEEE, 2012. URL <http://www.cs.toronto.edu/~famelis/icse12.pdf>.
- Michalis Famelis, Naama Ben-David, Alessio Di Sandro, Rick Salay, and Marsha Chechik. Mu-Mmint: An IDE for model uncertainty. In *Proceedings of the 37<sup>th</sup> International Conference on Software Engineering - Volume 2*, ICSE ’15, pages 697–700, Piscataway, NJ, USA, 2015. IEEE Press. URL <https://dl.acm.org/citation.cfm?id=2819009.2819141>.
- George Fink and Matt Bishop. Property-based testing: A new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, July 1997. ISSN 0163-5948. doi: 10.1145/263244.263267. URL <https://doi.acm.org/10.1145/263244.263267>.

Bryan Ford. Parsing Expression Grammars: A recognition-based syntactic foundation. In *Proceedings of the 31<sup>st</sup> ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. doi: 10.1145/964001.964011. URL <https://doi.acm.org/10.1145/964001.964011>.

M. Fowler. Fluent interface, 2005. URL <http://martinfowler.com/bliki/FluentInterface.html>.

Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in JavaScript. In *Proceedings of the 39<sup>th</sup> International Conference on Software Engineering*, ICSE '17, pages 758–769, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.75. URL <https://doi.org/10.1109/ICSE.2017.75>.

Lisa Gerhardt, Wahid Bhimji, Shane Canon, Markus Fasel, Doug Jacobsen, Mustafa Mustafa, Jeff Porter, and Vakho Tsulaia. Shifter: Containers for HPC. *Journal of Physics: Conference Series*, 898:082021, oct 2017. doi: 10.1088/1742-6596/898/8/082021. URL <https://doi.org/10.1088%2F1742-6596%2F898%2F8%2F082021>.

Carlo Ghezzi and Dino Mandrioli. Incremental parsing. *ACM Trans. Program. Lang. Syst.*, 1(1):58–70, January 1979. ISSN 0164-0925. doi: 10.1145/357062.357066. URL <https://doi.acm.org/10.1145/357062.357066>.

Yossi Gil and Tomer Levy. Formal Language Recognition with the Java Type Checker. 56:10:1–10:27, 2016. ISSN 1868-8969. doi: 10.4230/LIPIcs.ECOOP.2016.10. URL <http://drops.dagstuhl.de/opus/volltexte/2016/6104>.

VM Glushkov, VG Bodnarchuk, TA Grinchenko, AA Dorodnitsyna, VP Klimenko, AA Letichevskii, SB Pogrebinskii, AA Stognii, and Yu S Fishman. ANALITIK (algorithmic language for the description of computing processes using analytical transformations). *Cybernetics*, 7(3):513–552, 1971. URL <https://link.springer.com/content/pdf/10.1007/BF01070461.pdf>.

Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.

C. F. Goldfarb. A generalized approach to document markup. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 68–73, New York, NY, USA, 1981. ACM. ISBN 0-89791-050-8. doi: 10.1145/800209.806456. URL <https://doi.acm.org/10.1145/800209.806456>.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets. In Proceedings of the 27<sup>th</sup> International Conference on Neural Information Processing Systems - Volume 2, NIPS'14, pages 2672–2680, Cambridge, MA, USA, 2014. MIT Press. URL <https://dl.acm.org/citation.cfm?id=2969033.2969125>.

Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. Probabilistic programming with densities in SlicStan: Efficient, flexible, and deterministic. Proc. ACM Program. Lang., 3(POPL):35:1–35:30, January 2019. ISSN 2475-1421. doi: 10.1145/3290348. URL <https://doi.acm.org/10.1145/3290348>.

Andreas Griewank. Some bounds on the complexity of gradients, Jacobians, and Hessians. In Complexity in numerical optimization, pages 128–162. World Scientific, 1993. URL [http://ftp.mcs.anl.gov/pub/tech\\_reports/reports/P355.pdf](http://ftp.mcs.anl.gov/pub/tech_reports/reports/P355.pdf).

Andreas Griewank et al. On automatic differentiation. Mathematical Programming: recent developments and applications, 6(6):83–107, 1989.

P. R. Griffioen. Type inference for array programming with dimensioned vector spaces. In Proceedings of the 27<sup>th</sup> Symposium on the Implementation and Application of Functional Programming Languages, IFL '15, pages 4:1–4:12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4273-5. doi: 10.1145/2897336.2897341. URL <https://doi.acm.org/10.1145/2897336.2897341>.

Radu Grigore. Java generics are Turing Complete. pages 73–85, 2017. doi: 10.1145/3009837.3009871. URL <https://doi.acm.org/10.1145/3009837.3009871>.

Martin Guenther. Are serious things done with ROS in Python?, 2018. URL <https://discourse.ros.org/t/are-serious-things-done-with-ros-in-python/4359/6>.

Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 3–14. IEEE, 2017. URL [https://www.cs.mcgill.ca/~jguo/resources/papers/ICSE2017\\_JIN\\_Preprint.pdf](https://www.cs.mcgill.ca/~jguo/resources/papers/ICSE2017_JIN_Preprint.pdf).

Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, LFP '84, pages 293–298, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: 10.1145/800055.802046. URL <https://doi.acm.org/10.1145/800055.802046>.

- Jordan Henkel, Christian Bird, Shuvendu Lahiri, and Thomas Reps. Learning from, understanding, and supporting DevOps artifacts for Docker. In 42nd International Conference on Software Engineering (ICSE'20), May 2020. URL <https://www.microsoft.com/en-us/research/publication/learning-from-understanding-and-supporting-devops-artifacts-for-docker/>.
- Pieter Hintjens. ZeroMQ: messaging for many applications. O'Reilly Media, Inc., 2013.
- Robert Hirschfeld, Pascal Costanza, and Oscar Marius Nierstrasz. Context-oriented programming. Journal of Object Technology, 7(3):125–151, 2008.
- K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. Neural Netw., 2(5):359–366, July 1989. ISSN 0893-6080. doi: 10.1016/0893-6080(89)90020-8. URL [https://dx.doi.org/10.1016/0893-6080\(89\)90020-8](https://dx.doi.org/10.1016/0893-6080(89)90020-8).
- Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. DiffTaichi: Differentiable programming for physical simulation. CoRR, abs/1910.00935, 2019. URL <https://arxiv.org/abs/1910.00935>.
- Teijiro Isokawa, Tomoaki Kusakabe, Nobuyuki Matsui, and Ferdinand Peper. Quaternion neural network and its application. In International Conference on Knowledge-Based and Intelligent Information and Engineering Systems, pages 318–324. Springer, 2003. URL [https://link.springer.com/chapter/10.1007/978-3-540-45226-3\\_44](https://link.springer.com/chapter/10.1007/978-3-540-45226-3_44).
- Alekseĭ Grigorevich Ivakhnenko and Valentin Grigorévich Lapa. Cybernetic predicting devices. CCM Information Corporation, 1965. URL <https://books.google.ca/books?id=FhwVNQAAACAAJ>.
- Kenneth E Iverson. Math for the layman, 1999. URL [http://www.cs.trinity.edu/About/The\\_Courses/cs301/math-for-the-layman/](http://www.cs.trinity.edu/About/The_Courses/cs301/math-for-the-layman/).
- Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. volume abs/1712.05877, 2017. URL <https://arxiv.org/abs/1712.05877>.
- C. Barry Jay and Milan Sekanina. Shape checking of array programs. Technical report, In Computing: the Australasian Theory Seminar, Proceedings, 1997.
- Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for

fast feature embedding. In *Proceedings of the 22<sup>nd</sup> ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3063-3. doi: 10.1145/2647868.2654889. URL <https://doi.acm.org/10.1145/2647868.2654889>.

M. Kac. On some connections between probability theory and differential and integral equations. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 189–215, Berkeley, Calif., 1951. University of California Press. URL <https://projecteuclid.org/euclid.bsmsp/1200500229>.

W. Kahan. Further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40–, January 1965. ISSN 0001-0782. doi: 10.1145/363707.363723. URL <https://doi.acm.org/10.1145/363707.363723>.

Nidhi Kalra and Susan M Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 2016.

K. Kastner, J. F. Santos, Y. Bengio, and A. Courville. Representation mixing for tts synthesis. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5906–5910, May 2019. doi: 10.1109/ICASSP.2019.8682880.

Robert Kelly, Barak A. Pearlmutter, and Jeffrey Mark Siskind. Evolving the incremental  $\lambda$  calculus into a model of forward automatic differentiation (AD). *CoRR*, abs/1611.03429, 2016. URL <https://arxiv.org/abs/1611.03429>.

Andrew Kennedy. Dimension types. In *European Symposium on Programming*, pages 348–362. Springer, 1994.

Andrew Kennedy. *Types for Units-of-Measure: Theory and Practice*, pages 268–305. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-17685-2. doi: 10.1007/978-3-642-17685-2\_8. URL [https://doi.org/10.1007/978-3-642-17685-2\\_8](https://doi.org/10.1007/978-3-642-17685-2_8).

Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 21–40, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094814. URL <https://doi.acm.org/10.1145/1094811.1094814>.

- Andrew John Kennedy. Programming languages and dimensions. Technical report, University of Cambridge, Computer Laboratory, 1996. URL <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-391.pdf>.
- B. W. Kernighan and P. J. Plauger. Software tools. *SIGSOFT Softw. Eng. Notes*, 1(1): 1520, May 1976. ISSN 0163-5948. doi: 10.1145/1010726.1010728. URL <https://doi.org/10.1145/1010726.1010728>.
- Oleg Kiselyov. Number-parameterized types. *The Monad Reader*, 5:73–118, 2005.
- Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. April 2009. URL <https://www.microsoft.com/en-us/research/publication/fun-type-functions/>.
- Gerwin Klein, Steve Rowe, and Régis Décamps. JFlex—the fast scanner generator for Java. 2001. URL <http://www.jflex.de>.
- Max Kochurov, Colin Carroll, Thomas Wiecki, and Junpeng Lao. PyMC4: Exploiting coroutines for implementing a probabilistic programming framework. 2019. URL <https://openreview.net/pdf?id=rkgzj5Za8H>.
- Rainer Koppler. A systematic approach to fuzzy parsing. *Softw. Pract. Exper.*, 27(6): 637–649, June 1997. ISSN 0038-0644. doi: 10.1002/(SICI)1097-024X(199706)27:6<637::AID-SPE99>3.0.CO;2-3. URL [https://dx.doi.org/10.1002/\(SICI\)1097-024X\(199706\)27:6<637::AID-SPE99>3.0.CO;2-3](https://dx.doi.org/10.1002/(SICI)1097-024X(199706)27:6<637::AID-SPE99>3.0.CO;2-3).
- Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017. doi: 10.1371/journal.pone.0177459. URL <https://doi.org/10.1371/journal.pone.0177459>.
- Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*, 2019. URL <https://arxiv.org/pdf/1912.01412.pdf>.
- Leslie Lamport. A discussion with Leslie Lamport, August 2002. URL <https://www.microsoft.com/en-us/research/publication/discussion-leslie-lamport/>.
- Chris Lattner and Richard Wei. Swift for TensorFlow. 2018. URL <https://github.com/tensorflow/swift>.
- Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of Moore’s law, 2020.

- Sören Laue. On the equivalence of forward mode automatic differentiation and symbolic differentiation. *CoRR*, abs/1904.02990, 2019. URL <https://arxiv.org/abs/1904.02990>.
- Sören Laue, Matthias Mitterreiter, and Joachim Giesen. Computing higher order derivatives of matrix and tensor expressions. In *Advances in Neural Information Processing Systems*, pages 2750–2759, 2018. URL <http://papers.nips.cc/paper/7540-computing-higher-order-derivatives-of-matrix-and-tensor-expressions.pdf>.
- Sören Laue, Matthias Mitterreiter, and Joachim Giesen. A simple and efficient tensor calculus. In *Conference on Artificial Intelligence (AAAI)*, 2020. URL [https://theinf2.informatik.uni-jena.de/theinf2\\_multimedia/Publications/tensorCalculus.pdf](https://theinf2.informatik.uni-jena.de/theinf2_multimedia/Publications/tensorCalculus.pdf).
- Gary T Leavens and Todd D Millstein. Multiple dispatch as dispatch on tuples. *ACM SIGPLAN Notices*, 33(10):374–387, 1998. URL <http://web.cs.ucla.edu/~todd/research/oopsla98.pdf>.
- Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015. ISSN 0028-0836. doi: 10.1038/nature14539.
- Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph.*, 37(4):139:1–139:13, July 2018. ISSN 0730-0301. doi: 10.1145/3197517.3201383. URL <https://doi.acm.org/10.1145/3197517.3201383>.
- J. C. R. Licklider. Man-computer symbiosis. *IEEE Ann. Hist. Comput.*, 14(1):24–, January 1992. ISSN 1058-6180. URL <https://dl.acm.org/citation.cfm?id=612400.612433>.
- David Lieb, Andrew Lookingbill, and Sebastian Thrun. Adaptive road following using self-supervised learning and reverse optical flow. In *Robotics: science and systems*, pages 273–280, 2005.
- Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, Jun 1976. ISSN 1572-9125. doi: 10.1007/BF01931367. URL <https://doi.org/10.1007/BF01931367>.
- Barbara Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN Not.*, 23(5):17–34, January 1987. ISSN 0362-1340. doi: 10.1145/62139.62141. URL <https://doi.acm.org/10.1145/62139.62141>.
- Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. *CoRR*, abs/1806.09055, 2018. URL <https://arxiv.org/abs/1806.09055>.

P Martin Lof et al. An intuitionistic theory of types. In Predicative part colloquium, pages 73–118, 1973.

Matthew M Loper and Michael J Black. OpenDR: An approximate differentiable renderer. In European Conference on Computer Vision, pages 154–169. Springer, 2014.

Daniel Lowd and Christopher Meek. Adversarial learning. In Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, KDD '05, pages 641–647, New York, NY, USA, 2005. ACM. ISBN 1-59593-135-X. doi: 10.1145/1081870.1081950. URL <https://doi.acm.org/10.1145/1081870.1081950>.

Sitao Luan, Mingde Zhao, Xiao-Wen Chang, and Doina Precup. Break the ceiling: Stronger multi-scale deep graph convolutional networks. In Advances in Neural Information Processing Systems, pages 10943–10953, 2019.

David R. MacIver. Hypothesis, 2018. URL <https://github.com/HypothesisWorks/hypothesis>.

Dougal Maclaurin. Modeling, Inference and Optimization with Composable Differentiable Procedures. PhD thesis, Harvard University, April 2016. URL <https://dash.harvard.edu/handle/1/33493599>.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in NumPy. In ICML 2015 AutoML Workshop, 2015. URL <https://github.com/HIPS/autograd>.

J.R. Magnus and H. Neudecker. Matrix differential calculus with applications in statistics and econometrics. Wiley series in probability and mathematical statistics. Wiley, 1988. ISBN 0471915165. Pagination: xvii, 393.

Dhruv C. Makwana and Neelakantan R. Krishnaswami. NumLin: Linear Types for Linear Algebra. In Alastair F. Donaldson, editor, 33<sup>rd</sup> European Conference on Object-Oriented Programming (ECOOP 2019), volume 134 of Leibniz International Proceedings in Informatics (LIPIcs), pages 14:1–14:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-111-5. doi: 10.4230/LIPIcs.ECOOP.2019.14. URL <http://drops.dagstuhl.de/opus/volltexte/2019/10806>.

Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. Docker ecosystem–vulnerability analysis. Computer Communications, 122:30–43, 2018.

Joaquim RRA Martins, Peter Sturdza, and Juan J Alonso. The complex-step derivative approximation. *ACM Transactions on Mathematical Software (TOMS)*, 29(3):245–262, 2003.

Conor McBride. The derivative of a regular type is its type of one-hole contexts. 2001.

Conor McBride. Clowns to the left of me, jokers to the right (pearl): Dissecting data structures. In *Proceedings of the 35<sup>th</sup> Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’08, pages 287–295, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328474. URL <https://doi.acm.org/10.1145/1328438.1328474>.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960. URL <https://doi.org/10.1145/367177.367199>.

Bhairav Mehta, Manfred Diaz, Florian Golemo, Christopher J. Pal, and Liam Paull. Active domain randomization, 2019. URL <https://arxiv.org/abs/1904.04762>.

Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’06, pages 706–706, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142552. URL <https://doi.acm.org/10.1145/1142473.1142552>.

Paul B Menage. Adding generic process containers to the Linux kernel. In *Proceedings of the Linux Symposium*, volume 2, pages 45–57, 2007.

Charith Mendis, Cambridge Yang, Yewen Pu, Dr.Saman Amarasinghe, and Michael Carbin. Compiler auto-vectorization with imitation learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 14598–14609. Curran Associates, Inc., 2019. URL <http://papers.nips.cc/paper/9604-compiler-auto-vectorization-with-imitation-learning.pdf>.

Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239), March 2014. ISSN 1075-3583. URL <https://dl.acm.org/citation.cfm?id=2600239.2600241>.

Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: A functional pearl. In Proceedings of the 16<sup>th</sup> ACM SIGPLAN International Conference on Functional Programming, ICFP ’11, pages 189–195, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034801. URL <https://doi.acm.org/10.1145/2034773.2034801>.

Ioannis Mitliagkas, Constantine Caramanis, and Prateek Jain. Memory limited, streaming PCA. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, Advances in Neural Information Processing Systems 26, pages 2886–2894. Curran Associates, Inc., 2013. URL <http://papers.nips.cc/paper/5035-memory-limited-streaming-pca.pdf>.

Aaron Moss. Derivatives of Parsing Expression Grammars. In Proceedings 15<sup>th</sup> International Conference on Automata and Formal Languages, AFL 2017, Debrecen, Hungary, September 4-6, 2017., pages 180–194, 2017. doi: 10.4204/EPTCS.252.18. URL <https://doi.org/10.4204/EPTCS.252.18>.

Maurice Naftalin and Philip Wadler. Java generics and collections. O’Reilly Media, 2007.

Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. Silver-chain: A fluent API generator. In Proceedings of the 16<sup>th</sup> ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, pages 199–211, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5524-7. doi: 10.1145/3136040.3136041. URL <https://doi.acm.org/10.1145/3136040.3136041>.

Peter Naur. Programming as theory building. Microprocessing and microprogramming, 15(5):253–261, 1985.

Yuri Nesterov. Gradient methods for minimizing composite functions. Mathematical Programming, 140(1):125–161, Aug 2013. ISSN 1436-4646. doi: 10.1007/s10107-012-0629-5. URL <https://doi.org/10.1007/s10107-012-0629-5>.

Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. DyNet: The dynamic neural network toolkit. arXiv preprint arXiv:1701.03980, 2017.

Virginia Niculescu. A design proposal for an object oriented algebraic library. *Studia Universitatis Babes-Bolyai, Informatica*, 48(1):89–100, 2003.

Virginia Niculescu. On using generics for implementing algebraic structures. *Studia Universitatis Babes-Bolyai, Informatica*, 56(4), 2011.

Alexander Nozik. Kotlin language for science and kMath library. *AIP Conference Proceedings*, 2163(1):040004, 2019. doi: 10.1063/1.5130103. URL <https://aip.scitation.org/doi/abs/10.1063/1.5130103>.

Yu Nureki. JAutoDiff: A pure Java library for automatic differentiation, 2012. URL <https://github.com/uniker9/JAutoDiff>.

Takeshi Ogita, Siegfried M. Rump, and Shin’ichi Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, June 2005. ISSN 1064-8275. doi: 10.1137/030601818. URL <https://dx.doi.org/10.1137/030601818>.

Christopher Olah. Neural networks, types, and functional programming. 2015. URL <https://colah.github.io/posts/2015-09-NN-Types-FP>.

John-Paul Ore, Sebastian Elbaum, Carrick Detweiler, and Lambros Karkazis. Assessing the type annotation burden. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 190–201, 2018. URL <https://jpwco.com/pdf/ase18main-p15-p-bcc79e2-37685-final.pdf>.

Gerardo Pardo-Castellote. OMG Data-Distribution Service: Architectural overview. In *23<sup>rd</sup> International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pages 200–206. IEEE, 2003.

Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL <http://papers.neurips.cc/paper/>

[9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf](#).

Liam Paull, Howard Li, and Liuchen Chang. A novel domestic electric water heater model for a multi-objective demand side management program. *Electric Power Systems Research*, 80(12):1446–1451, 2010.

Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Yajun Fang, et al. Duckietown: an open, inexpensive and flexible platform for autonomy education and research. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1497–1504. IEEE, 2017.

David J Pearce and James Noble. Implementing a language with flow-sensitive and structural typing on the jvm. *Electronic Notes in Theoretical Computer Science*, 279(1):47–59, 2011.

Barak A Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008a.

Barak A. Pearlmutter and Jeffrey Mark Siskind. Using programming language theory to make automatic differentiation sound and efficient. pages 79–90, 2008b. ISSN 1439-7358. doi: 10.1007/978-3-540-68942-3\_8. URL <http://www.bcl.hamilton.ie/~barak/papers/sound-efficient-ad2008.pdf>.

Vaclav Pech, Alex Shatalin, and Markus Voelter. JetBrains MPS as a tool for extending Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ ’13*, pages 165–168, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2111-2. doi: 10.1145/2500828.2500846. URL <https://doi.acm.org/10.1145/2500828.2500846>.

Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26<sup>th</sup> Symposium on Operating Systems Principles, SOSP ’17*, pages 1–18, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132785. URL <https://doi.acm.org/10.1145/3132747.3132785>.

Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014.

- Association for Computational Linguistics. doi: 10.3115/v1/D14-1162. URL <https://www.aclweb.org/anthology/D14-1162>.
- Kaare Brandt Petersen et al. The Matrix Cookbook. 2012. URL <https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf>.
- Leonardo Piñeyro, Alberto Pardo, and Marcos Viera. Structure verification of deep neural networks at compilation time using dependent types. In Proceedings of the XXIII Brazilian Symposium on Programming Languages, SBLP 2019, pages 46–53, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-7638-9. doi: 10.1145/3355378.3355379. URL <https://doi.acm.org/10.1145/3355378.3355379>.
- Joelle Pineau, Geoffrey Gordon, and Sebastian Thrun. Policy-contingent abstraction for robust robot control. In Proceedings of Conference on Uncertainty in Artificial Intelligence (UAI), pages 477 – 484, August 2003. URL <https://arxiv.org/pdf/1212.2495.pdf>.
- Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d’Alché Buc, Emily Fox, and Hugo Larochelle. Improving reproducibility in machine learning research (a report from the neurips 2019 reproducibility program), 2020. URL <https://arxiv.org/pdf/2003.12206.pdf>.
- Gordon Plotkin. Some principles of differential programming languages. POPL, 2018.
- Gill A Pratt. Is a Cambrian explosion coming for robotics? Journal of Economic Perspectives, 29(3):51–60, 2015. URL [https://www.aeaweb.org/full\\_issue.php?doi=10.1257/jep.29.3#page=53](https://www.aeaweb.org/full_issue.php?doi=10.1257/jep.29.3#page=53).
- Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In ICRA workshop on open source software, volume 3, page 5. Kobe, Japan, 2009.
- Mirco Ravanelli, Dmitriy Serdyuk, and Yoshua Bengio. Twin regularization for online speech recognition. Interspeech 2018, Sep 2018. doi: 10.21437/interspeech.2018-1407. URL <http://dx.doi.org/10.21437/Interspeech.2018-1407>.
- Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in GitHub. Commun. ACM, 60(10): 91–100, September 2017. ISSN 0001-0782. doi: 10.1145/3126905. URL <https://doi.acm.org/10.1145/3126905>.

Norman A. Rink. Modeling of languages for tensor manipulation. *CoRR*, abs/1801.08771, 2018. URL <https://arxiv.org/abs/1801.08771>.

Mikael Rittri. Dimension inference under polymorphic recursion. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 147–159, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7. doi: 10.1145/224164.224197. URL <https://doi.acm.org/10.1145/224164.224197>.

Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new IR for machine learning frameworks. In *Proceedings of the 2<sup>nd</sup> ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 58–68, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5834-7. doi: 10.1145/3211346.3211348. URL <https://doi.acm.org/10.1145/3211346.3211348>.

Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1. doi: 10.1145/1868294.1868314. URL <https://doi.acm.org/10.1145/1868294.1868314>.

Frank Rosenblatt. The Perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. URL <https://www.ling.upenn.edu/courses/cogs501/Rosenblatt1958.pdf>.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018. URL <https://arxiv.org/abs/1805.00907>.

W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9<sup>th</sup> International Conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. ISBN 0-89791-216-0. URL <https://dl.acm.org/citation.cfm?id=41765.41801>.

Claudio Ruch, Sebastian Hörl, and Emilio Frazzoli. AMoDeus, a simulation-based testbed for autonomous mobility-on-demand systems. *2018 21<sup>st</sup> International Conference on Intelligent Transportation Systems (ITSC)*, pages 3639–3644, 2018. URL [https://www](https://www.https://www).

[amodeus.science/](http://amodeus.science/).

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. pages 696–699, 1988. URL <https://dl.acm.org/citation.cfm?id=65669.104451>.

Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. URL <http://web.mit.edu/Saltzer/www/publications/rfc/csr-rfc-060.pdf>.

Stephen Samuel and Leonardo Colman Lopes. KotlinTest, 2018. URL <https://github.com/kotlintest/kotlintest>.

Moses Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische annalen*, 92(3):305–316, 1924.

F. F. Sellers, M. Y. Hsiao, and L. W. Bearnsen. Analyzing errors with the boolean difference. *IEEE Trans. Comput.*, 17(7):676–683, July 1968. ISSN 0018-9340. doi: 10.1109/TC.1968.227417. URL <https://doi.org/10.1109/TC.1968.227417>.

Claude E Shannon. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950. URL <https://www.tandfonline.com/doi/abs/10.1080/14786445008521796>.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation*, 21(4):361–376, Dec 2008. ISSN 1573-0557. doi: 10.1007/s10990-008-9037-1. URL <https://doi.org/10.1007/s10990-008-9037-1>.

Michael Sperber, R Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised<sup>6</sup> report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(S1):1–301, 2009. URL <http://www.r6rs.org/final/html/r6rs/r6rs.html>.

Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the numeric tower. In *International Symposium on Practical Aspects of Declarative Languages*, pages 289–303. Springer, 2012.

Bernd Steinbach and Christian Posthoff. Boolean differential calculus. *Synthesis Lectures on Digital Circuits and Systems*, 12(1):1–215, 2017.

Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In Proceedings of the 28<sup>th</sup> International Conference on Machine Learning (ICML-11), pages 609–616, 2011. URL <https://stanford-ppl.github.io/website/papers/icml11-sujeeth.pdf>.

Gerald J. Sussman and Guy L. Steele, Jr. Scheme: An interpreter for Extended Lambda Calculus. Technical report, Cambridge, MA, USA, 1975.

Norihisa Suzuki and David Jefferson. Verification decidability of Presburger array programs. J. ACM, 27(1):191–205, January 1980. ISSN 0004-5411. doi: 10.1145/322169.322185. URL <https://doi.acm.org/10.1145/322169.322185>.

AD Talantsev. On the analysis and synthesis of certain electrical circuits by means of special logical operators. Avt. i Telem., 20(7):898–907, 1959. URL <http://www.mathnet.ru/links/bafa0b55349439b7d01a805198178a20/at12783.pdf>.

Ross Tate. Mixed-site variance. In FOOL ’13: Informal Proceedings of the 20th International Workshop on Foundations of Object-Oriented Languages, 2013. URL <http://www.cs.cornell.edu/~ross/publications/mixedsite/>.

H. C. A. Tavante, Benedito Donizeti Bonatto, and Maurilio Pereira Coutinho. Open source implementations of electromagnetic transient algorithms. 2018 13<sup>th</sup> IEEE International Conference on Industry Applications (INDUSCON), pages 825–828, 2018. URL <https://github.com/hannelita/PyTHTA>.

Eclipse Deeplearning4j Development Team. DL4J: Deep Learning for Java. 2016a. URL <https://github.com/eclipse/deeplearning4j>.

Eclipse Deeplearning4j Development Team. ND4J: Fast, scientific and numerical computing for the JVM. 2016b. URL <https://github.com/eclipse/deeplearning4j>.

A. Thayse and M. Davio. Boolean differential calculus and its application to switching theory. IEEE Trans. Comput., 22(4):409–420, April 1973. ISSN 0018-9340. doi: 10.1109/T-C.1973.223729. URL <https://dx.doi.org/10.1109/T-C.1973.223729>.

Andre Thayse. Boolean calculus of differences, volume 101 of ser-LNCS. 1981. ISBN 0-387-10286-8 (paperback). URL <https://www.springer.com/gp/book/9783540102861>.

Sebastian Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In Proceedings 2000 ICRA. Millennium Conference. IEEE

International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065), volume 1, pages 306–312. IEEE, 2000.

Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE ’18, pages 303–314, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180220. URL <https://doi.acm.org/10.1145/3180155.3180220>.

Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. URL [http://learningsys.org/papers/LearningSys\\_2015\\_paper\\_33.pdf](http://learningsys.org/papers/LearningSys_2015_paper_33.pdf).

Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C Pocock, Stephen Green, and Guy L Steele. Augur: Data-parallel probabilistic modeling. In *Advances in Neural Information Processing Systems*, pages 2600–2608, 2014. URL <https://papers.nips.cc/paper/5531-augur-data-parallel-probabilistic-modeling.pdf>.

Christos Tsirigotis, Xavier Bouthillier, François Corneau-Tremblay, Peter Henderson, Reyhane Askari, Samuel Lavoie-Marchildon, Tristan Deleu, Dendi Suhubdy, Michael Noukhovitch, Frédéric Bastien, et al. Oríon: Experiment version control for efficient hyperparameter optimization. 2018. URL <https://openreview.net/pdf?id=r1xkNLPixX>.

Ryan Turner. A model explanation system. In *2016 IEEE 26<sup>th</sup> International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6. IEEE, 2016.

Ryan Turner and Brady Neal. How well does your sampler really work? *arXiv preprint arXiv:1712.06006*, 2017. URL <https://arxiv.org/pdf/1712.06006.pdf>.

Ryan Turner, Jane Hung, Eric Frank, Yunus Saatchi, and Jason Yosinski. Metropolis-Hastings Generative Adversarial Networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36<sup>th</sup> International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6345–6353, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL <http://proceedings.mlr.press/v97/turner19a.html>.

- Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.
- Bart van Merriënboer. Sequence-to-sequence learning for machine translation and automatic differentiation for machine learning software tools. PhD thesis, Université de Montréal, September 2018. URL <http://hdl.handle.net/1866/21743>.
- Bart van Merriënboer, Dan Moldovan, and Alexander Wiltschko. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In *Advances in Neural Information Processing Systems 31*, pages 6256–6265, 2018. URL <https://arxiv.org/abs/1711.02712>.
- Bart van Merriënboernboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. Automatic differentiation in ML: where we are and where we should be going. *CoRR*, abs/1810.11530, 2018. URL <https://arxiv.org/abs/1810.11530>.
- Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor Comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018. URL <https://arxiv.org/abs/1802.04730>.
- Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. *Software Language Engineering, SLE*, 16(3), 2010. URL <https://pdfs.semanticscholar.org/5adb/c633179cd7d32e1c1840225b2890ddeb32a5.pdf>.
- Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering*, pages 41–61, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11245-9. URL <https://gsd.uwaterloo.ca/sites/default/files/2014-sle-projectional.pdf>.
- Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, PLDI '97*, pages 31–43, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. URL <https://doi.acm.org/10.1145/258915.258920>.
- Timothy A. Wagner. Practical Algorithms for Incremental Software Development Environments. PhD thesis, EECS Department, University of California, Berkeley, Mar

1998. URL <http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/5885.html>.
- Fei Wang, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. In *Advances in Neural Information Processing Systems 31*, pages 10180–10191, 2018a. URL <https://www.cs.purdue.edu/homes/rompf/papers/wang-nips18.pdf>.
- Fei Wang, Xilun Wu, Gregory M. Essertel, James M. Decker, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *CoRR*, abs/1803.10228, 2018b. URL <https://arxiv.org/abs/1803.10228>.
- Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. GraphGAN: Graph representation learning with Generative Adversarial Nets. 2018c. URL <https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16611>.
- Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Proceedings of the 32<sup>nd</sup> International Conference on Neural Information Processing Systems, NIPS’18*, pages 7686–7695, 2018d. URL <https://dl.acm.org/citation.cfm?id=3327757.3327866>.
- Xie Wang, Huaijin Wang, Zhendong Su, et al. Global optimization of numerical programs via prioritized stochastic algebraic transformations. In *Proceedings of the 41<sup>st</sup> International Conference on Software Engineering, ICSE ’19*, pages 1131–1141, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE.2019.00116. URL <https://doi.org/10.1109/ICSE.2019.00116>.
- Richard Wei, Vikram S. Adve, and Lane Schwartz. DLVM: A modern compiler infrastructure for deep learning systems. *CoRR*, abs/1711.03016, 2017. URL <https://arxiv.org/abs/1711.03016>.
- R. E. Wengert. A simple automatic derivative evaluation program. *Commun. ACM*, 7(8):463–464, August 1964. ISSN 0001-0782. doi: 10.1145/355586.364791. URL <https://doi.acm.org/10.1145/355586.364791>.
- Paul J Werbos et al. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.
- Ruffin White and Henrik Christensen. ROS and Docker. In *Robot Operating System (ROS)*, pages 285–307. Springer, 2017.

Norbert Wiener. Some moral and technical consequences of automation. *Science*, 131(3410):1355–1358, 1960. ISSN 0036-8075. doi: 10.1126/science.131.3410.1355. URL <https://science.scienmag.org/content/131/3410/1355>.

Virginia Vassilevska Williams. Multiplying matrices in  $\mathcal{O}(n^{2.373})$  time. 2014. URL <https://people.csail.mit.edu/virgi/matrixmult-f.pdf>.

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Trans. Evol. Comp*, 1(1):67–82, April 1997. ISSN 1089-778X. doi: 10.1109/4235.585893. URL <https://doi.org/10.1109/4235.585893>.

Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. *CoRR*, abs/1802.04680, 2018. URL <https://arxiv.org/abs/1802.04680>.

Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI ’98, pages 249–257, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277732. URL <https://doi.acm.org/10.1145/277650.277732>.

Anqi Xu. *Efficient Collaboration with Trust-seeking Robots*. PhD thesis, McGill University Libraries, 2017. URL <http://www.cim.mcgill.ca/~anqixu/thesis/thesis.pdf>.

Wojciech Zaremba. *Learning Algorithms from Data*. PhD thesis, New York University, 2016. URL [https://cs.nyu.edu/media/publications/zaremba\\_wojciech.pdf](https://cs.nyu.edu/media/publications/zaremba_wojciech.pdf).

Wojciech Zaremba, Karol Kurach, and Rob Fergus. Learning to discover efficient mathematical identities. In *Proceedings of the 27<sup>th</sup> International Conference on Neural Information Processing Systems - Volume 1*, NIPS’14, pages 1278–1286, Cambridge, MA, USA, 2014. MIT Press. URL <https://dl.acm.org/citation.cfm?id=2968826.2968969>.

Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187(1-2):147–165, November 1997. ISSN 0304-3975. doi: 10.1016/S0304-3975(97)00062-5. URL [https://dx.doi.org/10.1016/S0304-3975\(97\)00062-5](https://dx.doi.org/10.1016/S0304-3975(97)00062-5).

Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, page 11, 2020. ISSN 2326-3881. doi: 10.1109/tse.2019.2962027. URL <http://dx.doi.org/10.1109/tse.2019.2962027>.

Zhi Quan Zhou and Liqun Sun. Metamorphic testing of driverless cars. *Commun. ACM*, 62(3):61–67, February 2019. ISSN 0001-0782. doi: 10.1145/3241979. URL <https://doi.acm.org/10.1145/3241979>.

Julian G. Zilly, Jacopo Tani, Breandan Considine, Bhairav Mehta, Andrea F. Daniele, Manfred Diaz, Gianmarco Bernasconi, Claudio Ruch, Jan Hakenberg, Florian Golemo, A. Kirsten Bowser, Matthew R. Walter, Ruslan Hristov, Sunil Mallya, Emilio Frazzoli, Andrea Censi, and Liam Paull. The AI driving olympics at NeurIPS 2018. *CoRR*, abs/1903.02503, 2019. URL <https://arxiv.org/abs/1903.02503>.

# Chapter 7

---

## Type-safe differentiable programming

### 7.1. Grammaire

Vous trouverez ci-dessous une grammaire BNF à peu près complète pour Kotlin $\nabla$  :

```
<type> ::= Double | Float | Int | BigInteger | BigDouble  
<nat> ::= 1 | ... | 99  
<output> ::= Fun<type>Real> | VFun<type>Real,<nat>> | MFun<type>Real,<nat>,<nat>>  
<int> ::= 0 | <nat><int>  
<float> ::= <int>.<int>  
<num> ::= <type>(<int>) | <type>(<float>)  
<var> ::= x | y | z | ONE | ZERO | E | Var()  
<signOp> ::= + | -  
<binOp> ::= <signOp> | * | / | pow  
<trigOp> ::= sin | cos | tan | asin | acos | atan | asinh | acosh | atanh  
<unaryOp> ::= <signOp> | <trigOp> | sqrt | log | ln | exp  
<exp> ::= <var> | <num> | <unaryOp><exp> | <var><binOp><exp> | (<exp>)  
<expList> ::= <exp> | <exp>,<expList>  
<linOp> ::= <signOp> | * | dot
```

```

⟨vec⟩ ::= Vec(⟨expList⟩) | Vec⟨nat⟩(⟨expList⟩)

⟨vecExp⟩ ::= ⟨vec⟩ | ⟨signOp⟩⟨vecExp⟩ | ⟨exp⟩*⟨vecExp⟩ | ⟨vec⟩⟨linOp⟩⟨vecExp⟩ |
             ⟨vecExp⟩.norm(⟨int⟩)

⟨mat⟩ ::= Mat⟨nat⟩x⟨nat⟩(⟨expList⟩)

⟨matExp⟩ ::= ⟨mat⟩ | ⟨signOp⟩⟨matExp⟩ | ⟨exp⟩⟨linOp⟩⟨matExp⟩ |
             ⟨vecExp⟩⟨linOp⟩⟨matExp⟩ | ⟨mat⟩⟨linOp⟩⟨matExp⟩

⟨anyExp⟩ ::= ⟨exp⟩ | ⟨vecExp⟩ | ⟨matExp⟩ | ⟨derivative⟩ | ⟨invocation⟩

⟨bindings⟩ ::= ⟨exp⟩ to ⟨exp⟩ | ⟨exp⟩ to ⟨exp⟩, ⟨bindings⟩

⟨invocation⟩ ::= ⟨anyExp⟩(⟨bindings⟩)

⟨derivative⟩ ::= d(⟨anyExp⟩) / d(⟨exp⟩) | ⟨anyExp⟩.d(⟨exp⟩) | ⟨anyExp⟩.d(⟨expList⟩)

⟨gradient⟩ ::= ⟨exp⟩.grad()

```

# Chapter 8

---

## Testing intelligent systems

### 8.1. Linear regression

Rappeler l'équation matricielle de la régression linéaire (LR), où  $\mathbf{X} : \mathbb{R}^{m \times n}$  et  $\boldsymbol{\Theta} : \mathbb{R}^{n \times 1}$  :

$$\hat{\mathbf{f}}(\mathbf{X}; \boldsymbol{\Theta}) = \mathbf{X}\boldsymbol{\Theta} \quad (8.1.1)$$

Imaginez que l'on nous donne l'ensemble de données suivant :

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_m \end{bmatrix} = \begin{pmatrix} 1 & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ 1 & \dots & x_{mn} \end{pmatrix}, \mathbf{Y} = \begin{bmatrix} y_2 \\ \vdots \\ y_m \end{bmatrix} \quad (8.1.2)$$

Notre objectif en matière de moindres carrés ordinaires (OLS) LR est de minimiser la perte, ou l'erreur entre les données et la prédition du modèle :

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}; \boldsymbol{\Theta}) = \|\mathbf{Y} - \hat{\mathbf{f}}(\mathbf{X}; \boldsymbol{\Theta})\|^2 \quad (8.1.3)$$

$$\boldsymbol{\Theta}^* = \underset{\boldsymbol{\Theta}}{\operatorname{argmin}} \mathcal{L}(\mathbf{X}, \mathbf{Y}; \boldsymbol{\Theta}) \quad (8.1.4)$$

#### 8.1.1. Méthode des différences finies

Tout d'abord, nous considérons le cas scalaire, où  $\hat{\mathbf{f}}(\mathbf{X}; \boldsymbol{\Theta}) = \hat{f}(x; \theta_2, \theta_1) = \theta_2 x + \theta_1$ .

Puisque  $\mathbf{X}, \mathbf{Y}$  sont considérés comme fixes, nous pouvons réécrire  $\mathcal{L}(\mathbf{X}, \mathbf{Y}; \boldsymbol{\Theta})$  aussi simplement :

$$\mathcal{L}(\boldsymbol{\Theta}) = \mathcal{L}(\theta_2, \theta_1) = \frac{1}{m} \sum_{i=1}^m (y_i - (\theta_2 x_i + \theta_1))^2 \quad (8.1.5)$$

Pour trouver le minimiseur de  $\mathcal{L}(\boldsymbol{\Theta})$ , nous avons besoin de  $\nabla_{\boldsymbol{\Theta}} \mathcal{L} = [\frac{\partial \mathcal{L}}{\partial \theta_2},$