Résolution du problème du labyrinthe par apprentissage par renforcement avec un réseau neuronal - Extraction d'un réseau de neurones des équations de Navier-Stokes

Nicolas,Yax

Département d'informatique ENS Cachan

August 21, 2019

Implémentation :
`https://github.com/Yaaxx/maze-resolution-with-pNS-neurons`

1 Introduction

Mon stage s'est déroulé dans l'équipe SequeL à l'INRIA Lille sous la direction de Philippe Preux. L'équipe SequeL s'intéresse à l'apprentissage séquentiel c'est à dire à une forme d'apprentissage où l'information arrive au fur et à mesure que l'agent explore son environnement. Elle travaille notamment sur l'apprentissage par renforcement et sur les problèmes de Bandits.
Mon stage est de 6 semaines et je dois travailler sur la résolution du problème du labyrinthe c'est à dire proposer un agent capable de résoudre n'importe quel labyrinthe donné sous la forme de tableau de 0 et de 1. Cet exercice assez classique en utilisant du Q-Learning tabulaire par exemple. Cependant cette méthode présente des inconvénient. En effet, comme nous allons le voir plus tard, le Q-Learning tabulaire utilise un tableau pour stocker des résultats. Cependant il existe parfois des liens entre les cases de ce tableau ce qui fait que comme un SUDOKU, à partir de quelques valeurs dans le tableau on peut en déduire le reste. Cette capacité à comprendre liens entre les cases du tableau permettrait à la fois une réduction du temps d'apprentissage car on n'aurait pas à entrer manuellement les données dans chaque case mais il y aurait aussi une factorisation de l'espace mémoire. Ainsi on résoudrait le problème à la fois plus efficacement en temps mais aussi de manière plus optimisée en espace. Pour implémenter une telle capacité de généralisation en pratique on peut utiliser un réseau de neurones qui est une structure capable de comprendre les liens entre les différents éléments du tableau afin de prédire les autres cases. Or les réseaux de neurones classiques sont actuellement mal adaptés pour résoudre ce problème car les informations sensées être stockées dans la table pour ce problème précis ne sont pas très continues (à cause des murs qui font une rupture) ce qui rend la table très difficile à remplacer par un réseau de neurones classique tout en conservant de bonnes performances. Voila le problème que je vais devoir résoudre. Je me suis dans un premier temps intéressé à l'amélioration en temps car de nos jours l'espace mémoire n'est pas vraiment un problème et devoir résoudre un labyrinthe tellement grand que les approches classiques ne fonctionnent plus me semble être un problème qui n'arrive que très peu en pratique. J'ai alors eu une idée assez originale en observant les champs de vecteurs que devait apprendre le réseau neuronal. En effet, ils ressemblaient à un écoulement d'eau dans le labyrinthe. J'ai donc travaillé sur les équations différentielles de mécanique de fluides (Équations de Navier-Stokes) et j'ai cherché à faire un réseau de neurones qui satisferait ces contraintes.
Ainsi nous allons tout d'abord voir le contexte scientifique et quelques définition puis nous allons suivre cette aventure des équations de Navier Stokes avec des réseaux neuronaux particuliers.

2 Contexte Scientifique

2.1 Apprentissage par renforcement

L'apprentissage par renforcement est une forme d'apprentissage qui pourrait être vu comme une simulation psychologique du fonctionnement comportemental des êtres vivants. L'agent va devoir maximiser une récompense qu'il va obtenir en effectuant, dans une certaine configuration, une certaine action qu'il aura choisi. Il peut avoir ainsi différentes récompenses, une pour chaque action possible dans la configuration dans laquelle il se trouve et devra maximiser la récompense obtenue (la plupart du temps, sur le long terme). Or l'agent ne connaît pas à l'avance les récompense associées aux différentes actions qui lui sont proposées. Il devra donc les découvrir au fil du temps. Il y a alors un choix à faire : explorer de nouvelles possibilités ou exploiter celles qui sont déjà connues. L'objectif de cet apprentissage est de trouver une bonne politique (fonction de choix, qui a une configuration donnée, donne l'action la plus intéressante) permettant de maximiser la récompense.

2.2 Q-Learning

Une des façons de faire de l'apprentissage par renforcement est d'utiliser la méthode de Q-Learning qui consiste en le fait, à partir d'une configuration donnée, de regarder pour chaque action possible les récompenses connues associées. En fonction des données connues et inconnues, on décide d'explorer une nouvelle possibilité ou d'en exploiter une autre. Suite à ce choix, l'agent reçoit de l'environnement la récompense réelle liée à son action qui est alors apprise par l'agent et sera désormais connue. On stocke usuellement ces résultats dans un tableau/dictionnaire et on parle alors de Q-Learning tabulaire.

2.3 Stratégies simples

Il existe plusieurs stratégies classiques pour réguler l'exploration et l'exploitation: le epsilon-greedy et le epsilon-first ont l'avantage d'être assez simples. Le concept du epsilon-greedy est de choisir l'exploration avec une probabilité epsilon et d'exploiter le reste du temps (c'est à dire qu'on va beaucoup plus exploiter que explorer). Enfin le epsilon-first consiste en le fait d'explorer pendant les N premiers coups puis d'exploiter le reste du temps. Il existe bien entendu beaucoup de variantes de ces 2 méthodes notamment en faisant varier epsilon en fonction du temps. L'idée étant, la plupart du temps, d'exploiter de plus en plus et donc de faire un mélange bien dosé des 2 concepts.

2.4 Réseau de neurones

Un réseau de neurones artificiel est fondamentalement un graphe orienté sans cycle sur lequel va s'effectuer un calcul par propagation de l'information. Voyons tout d'abord comment faire un calcul sur un graphe avant de voir ce qu'est un réseau de neurones. Tout d'abord, certains noeuds du graphe sont dits d'entrée et d'autres de sortie, les autres sont étiquetés par des opérations (pouvant être d'arité variable) 1.
Une fois les noeuds d'entrée remplis par des valeurs (souvent des réels), le calcul peut commencer : chaque noeud dont toutes les connections entrantes sont déjà calculées peut être calculé à son tour en effectuant l'opération du noeud sur les données entrantes. On peut donc effectuer un calcul par propagation des informations des noeuds d'entrée jusqu'à ce que les noeuds de sortie soient tous calculés. On peut alors calculer la fonction représentée par le graphe à partir d'un vecteur d'entrée 2.

Ce qu'on appelle *réseau neuronal* est un tel graphe mais constitué de motifs élémentaires appelés *neurones*. Pour les neurones classiques, on somme les entrées pondérées par des wi (appelés poids du réseau de neurones, ce sont des variables internes qui vont servir à l'apprentissage) puis on compose la fonction sigma qui est appelée *fonction d'activation* du neurone. On l'écrit souvent abusivement en écrivant la multiplication par les poids sur des arêtes pour alléger la notation 3.

La valeur de sortie est alors envoyé vers l'entrée d'autres neurones formant ainsi un réseau effectuant un calcul. Cette structure imite le fonctionnement des neurones biologiques qui, à partir de potentiels d'action arrivants, et qui sont plus ou moins bien transmis par les synapses, sont sommés puis si le potentiel résultant est suffisant, il se créé un nouveau potentiel d'action au niveau de la *zone gâchette* qui sera transmis aux neurones suivants. Ceci me permet donc d'introduire la fonction d'activation Heaviside qui n'est autre que $\mathbf{1}_{\mathbb{R}+}$ (si il y a un signal positif,

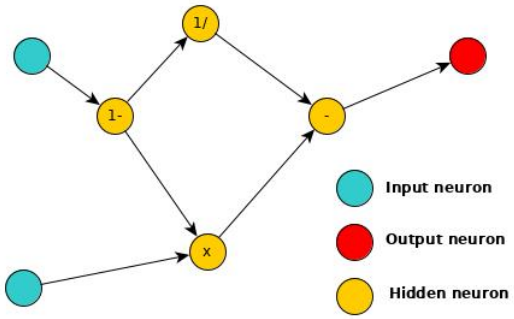


Figure 1: Calcul sur un graphe 1

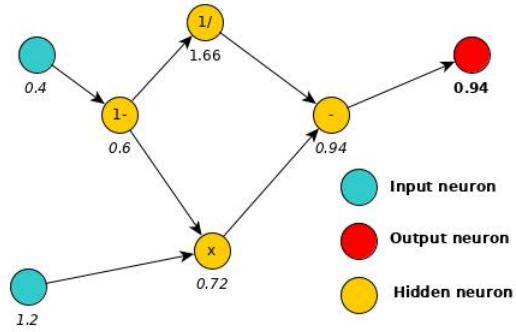


Figure 2: Calcul sur un graphe 2

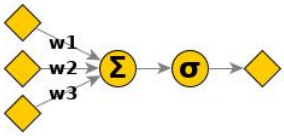


Figure 3: Calcul du neurone classique

un potentiel d'action est envoyé). Il existe beaucoup d'autres fonctions d'activation comme $relu(x) = max(0, x)$ qui est très utilisée. Il n'est pas facile de savoir quelle fonction d'activation utiliser. Ce choix se fait surtout par expérience et dépend de la formalisation du problème (si la sortie est bornée ou non par exemple).
Pour faire apprendre un réseau de neurones, la méthode la plus classique est de procéder par descente de gradient. Supposons que nous disposions d'un ensemble $(x_i, f(x_i))_{i\in I}$ que nous voulions faire apprendre à notre réseau de neurones. Il va falloir être capable de trouver une fonction dite *fonction de Loss* dépendante du vecteur de sortie du réseau de neurone que l'on nottera N dépendant de ses paramétres, qu'il va falloir minimiser et que cette minimisation permette l'apprentissage efficace de ces paramètres. Une fonction classique est alors la suivante (il en existe beaucoup d'autres):

$$L = \sum_{i\in I} (N(x_i) - f(x_i))^2$$

En effet, cette fonction peut sembler être un bon choix car si L vaut 0, alors on a bien appris tous les points voulus. Pour minimiser cette fonction , on va calculer le gradient de cette fonction de Loss par rapport à chacun des poids du réseau de neurones. On va alors utiliser une règle de mise à jour des poids comme celle-ci :

$$w_i := w_i - \lambda \frac{\partial L}{\partial wi}$$

Cette méthode permet de "suivre la courbure" de la fonction de Loss et d'adapter les poids de façon à progressivement descendre le gradient de cette fonction. Il faut savoir que le facteur $\lambda \in [0, 1]$ est appelé *facteur d'apprentissage*

et il permet d'affiner la précision de l'apprentissage : plus il est faible et plus l'ajustement des poids sera précis mais plus il sera lent. Bien entendu il arrive que notre fonction de Loss ait des minima locaux ce qui diminuera les performances de l'apprentissage. C'est pourquoi, encore une fois il faudra choisir sa fonction de Loss judicieusement tout comme les fonctions d'activation.
Pour calculer ce gradient, de nos jours on utilise une méthode de rétro-propagation dans le réseau de neurones afin de le calculer automatiquement. Il s'agit d'exprimer le gradient de L en fonction des dérivées partielles des valeurs de neurones de sortie. Ensuite on exprime les valeurs des neurones de sortie par rapport à celles des neurones précédents et on répète ce procédé récursivement jusqu'à ce que on arrive aux entrées (qui sont de dérivée nulle par rapport aux poids car les poids sont entre les neurones et qu'il n'y a plus de neurones les précédents).
Ceci est la théorie générale des réseaux de neurones. Il faut savoir que la grande majorité du temps les chercheurs préfèrent travailler sur des réseaux de neurones *par couches* 4 car cela permet d'avoir des formules de calcul de la sortie avec une expression matricielle ainsi que des formules de descente de gradient plus simples car le réseau est beaucoup plus homogène (ce ne sera pas notre cas car nous aurons besoin d'une version plus théorique des réseaux de neurones).

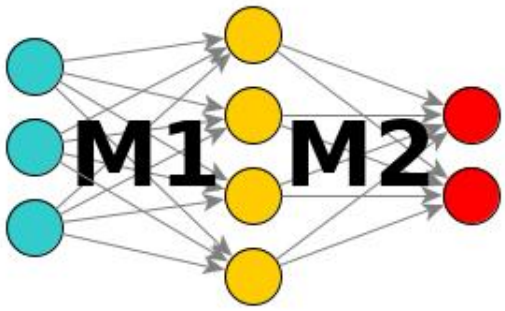


Figure 4: Réseau de neurones à 3 couches et 2 matrices de poids M1 et M2

3 Sujet de Recherche, démarche et résultats

La méthode développée ci-dessous fonctionne très bien sur le problème précis du labyrinthe. Cependant il faut comprendre que, en machine learning, on s'interesse à des méthods générales d'apprentissages donc le vrai résultat de ce rapport est la méthode appliquée au problème du labyrinthe et non la résolution du labyrinthe par l'agent. Je vais donc développer dans ces différentes parties les étapes successives du raisonnement ayant permis à partir d'un lot d'équations, de développer un réseau neuronal ayant le comportement voulu.

3.1 Motivations

Les approches classiques de résolution du problème du labyrinthe par renforcement avec un réseau de neurones consomment beaucoup plus de place mémoire qu'une approche tabulaire et prennent bien plus de temps pour résoudre le labyrinthe et apprendre une solution [5]. C'est donc le but de mon stage que de trouver une nouvelle approche afin de réussir à mieux utiliser les réseaux de neurones pour résoudre ce problème. Après avoir programmé une résolution du labyrinthe par Q-Learning tabulaire, il restait à trouver une forme de réseau de neurones (ce qu'on appelle *topologie* du réseau de neurones) qui serait capable de remplacer la table. D'après mon maître de stage, il serait plus simple de trouver une topologie capable d'apprendre les actions à effectuer directement plutôt que les récompenses associées à chaque action. Il s'agirait donc de partir sur une variante du Q-Learning dont le concept serait, au lieu de stocker une récompense dans la table, de stocker directement les actions les plus rentables pour l'agent. Ceci serait plus facile à apprendre pour un réseau de neurones. Or si on sait que les réseaux de neurones classiques ne sont pas bons pour résoudre ce problème, je propose d'aborder le problème un peu différement. En effet, le fait que les réseaux de neurones classiques soient si mauvais à cette tâche n'est pas étonnant. Ces approximateurs de fonctions ne peuvent approximer facilement que des fonctions

très continues et dès qu'il faut traiter d'un problème peu continu, le nombre de neurones nécessaires explose. Or ce qui fait la puissance des réseaux de neurones est le fait qu'ils soient composés de petits éléments que sont les neurones et que ces neurones vont interagir entre eux dans une structure pour résoudre des problèmes complexes. On peut alors penser que cette difficulté à résoudre ce problème de labyrinthe pourrait venir des composants élémentaires des réseaux de neurones classiques qui sont optimisés pour apprendre des fonctions continues et que en les remplaçant par d'autres mieux optimisés pour les environnements qui ont la forme d'un labyrinthe, on pourrait avoir de meilleurs résultats.
D'un autre coté, en étudiant les tables que le réseau de neurone devait apprendre, j'ai fait le rapprochement avec l'écoulement d'un fluide de l'entrée vers la sortie 5.

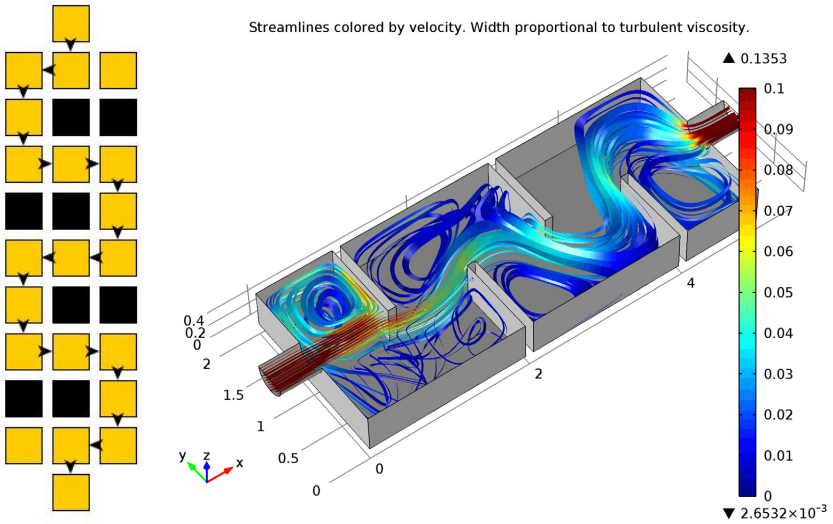


Figure 5: Solution d'un labyrinthe (à gauche) et sa résolution par mécanique des fluides (à droite)

Je me suis alors intéressé aux équations de Navier-Stokes qui régissent la mécanique des fluides et je me suis demandé comment il était possible de concevoir un neurone capable d'apprendre efficacement des solutions de ces équations. Puis en me souvenant du cours d'apprentissage d'algorithmique 2 de L3, je me suis dit : "Pourquoi seulement pouvoir approximer facilement des solutions et ne pas plutôt pouvoir apprendre **uniquement** des solutions ?". Les solutions des équations de Navier-Stokes seraient alors notre ensemble dit d'hypothèses (ensemble des fonctions dans lequel on va chercher notre fonction solution). Il s'agit donc d'une problématique de Deep Learning (faire apprendre efficacement beaucoup de données à un réseau de neurones).

3.2 Les équations de Navier-Stokes, les contraintes sur les réseaux de neurones

Les équations de Navier Stokes sur le champs de vitesse sont les suivantes : [1]

$$div\,\overrightarrow{v} = \overrightarrow{0} \tag{1}$$

$$\rho(\frac{\partial \overrightarrow{v}}{\partial t} + (\overrightarrow{v}.\overrightarrow{\nabla}).\overrightarrow{v}) = -\overrightarrow{\nabla}p + \mu\nabla^2\overrightarrow{v} + \overrightarrow{f} \tag{2}$$

Pour le moment, personne n'a trouvé de solution générale de ces équations différentielles. C'est pourquoi je compte résoudre ces équations numériquement avec un maillage simple : mon labyrinthe est constitué de cases, il s'agirait de prendre un point au centre de chaque case. De plus, ces équations ont une inconnue supplémentaire : le champs de pression p. Il va être utile pour coder les murs et la sortie. Il s'agira de mettre une plus forte pression dans les murs (environ 100 fois plus que dans les couloirs) ainsi que une pression très très faible à la sortie (environ 10^5 fois moins que dans les couloirs) afin que le flux converge vers cette case. Maintenant que le labyrinthe est modélisé, on va supprimer les termes inutiles des équations : je n'ai pas besoin d'un facteur temporel car les fluctuations temporelles ne m'intéressent pas. On va aussi oublier le terme de viscosité (plutôt utile près des bords mais je ne m'intéresse que au centre des cases) ainsi que le terme de force car on a codé les

murs dans le champs de pression. Voici les nouvelles équations de Navier-Stokes que je vais utiliser :

$$div\,\overrightarrow{v} = \overrightarrow{0} \tag{3}$$

$$\rho(\overrightarrow{v}.\overrightarrow{\nabla}).\overrightarrow{v} = -\overrightarrow{\nabla}p \tag{4}$$

A partir de ces équations, il va être possible de récupérer quelques résultats théoriques assez intéressants.
Tout d'abord, on ne sait pas actuellement résoudre ces équations explicitement. Il n'est donc pas question d'essayer même de manière détournée de les résoudre pendant mon stage. Il va donc être important de baliser le terrain avant d'essayer de faire quoi que ce soit afin d'espérer trouver quelque chose dans mon court séjour au laboratoire. Je vais donc aller beaucoup plus loin dans ces hypothèses et même supposer qu'elles n'ont pas de solution trouvables avec les outils mathématiques actuels. Ainsi comme la plupart des réseaux de neurones classiques ont une sortie calculable explicitement à partir d'un graphe, si j'arrivais à résoudre mon problème général, alors je pourrais résoudre Navier-Stokes ce qui est exclu. Avec cette simple réduction je sais déjà que je ne pourrais pas résoudre mon problème avec un réseau de neurones 100% explicite. On va donc être obligé d'avoir des noeuds qui vont devoir exécuter des opérations non triviales comme par exemple résoudre des équations dont on ne connaît pas de solution exacte. On pourrait alors se dire "On va soigner le mal par le mal" et essayer de faire un réseau de neurones dont tous les noeuds seraient des résolutions approchées de sous-équations différentielles et que la sortie finale soit une solution approchée des équations de Navier-Stokes. Cette méthode m'a paru assez originale mais n'a rien donné dans mes premiers essais (c'est une piste qui peut être très intéressante à suivre mais qui dépasse complètement mes connaissances en équations différentielles actuellement).
De plus, si je fais un réseau de neurones, j'ai besoin de le faire apprendre (sinon je ne pourrais jamais le faire remplacer ma table). La méthode d'apprentissage la plus utilisée est la descente de gradient. Je dois donc être capable de dériver partiellement mes fonctions dans mon réseau de manière explicite si possible (il faut être précis quand on fait des descentes de gradient). Ces fonctions sur mes noeuds doivent donc être de dérivée explicite mais qu'on ne puisse pas les intégrer explicitement sinon on résoudrait Navier-Stokes. Ce jeu de contraintes était suffisamment complexe pour que je pense à une autre solution.

3.3 Navier-Stokes par un réseau de neurones, contourner les contraintes

Si je ne peux pas apprendre par descente de gradient, il va falloir trouver une autre façon d'apprendre. De plus, les équations différentielles sont des équations de continuité locale et mes dérivées sont selon X et Y. L'idée la plus logique serait alors de faire un plan de neurones ce qui permettrait de faire plus facilement ces dérivées partielles. De plus, pourquoi ne pas faire dans un premier temps une "table de neurones" ? On n'y gagnerait pas en place par rapport au Q-Learning tabulaire mais on pourrait utiliser la puissance d'induction des réseaux de neurones pour apprendre les liens qui lient les éléments de la table et donc pour avoir un pouvoir prédictif bien supérieur à la méthode tabulaire. On aurait alors un gros gain en temps. L'intérêt de cette table de neurones serait aussi d'avoir une méthode d'apprentissage efficace car on n'aurait qu'à donner au neurone la valeur qu'on veut qu'il apprenne (ce ne seraient plus les connections entre neurones qui auraient des poids mais les neurones eux-mêmes) puis il se chargera de reconstruire une solution de Navier-Stokes en modifiant légèrement les neurones aux alentours en profitant de cette structure de plan pour faire ses calculs efficacement. Ceci semble donc une bonne approche qui contourne les contraintes théoriques précédentes.

3.4 Résoudre les équations de Navier-Stokes de manière séquentielle

Maintenant que l'on a une piste de topologie, il m'a semblé important de vérifier que cette solution était bien en accord avec le sujet posé et qu'elle y répondrait bien. Vu que l'on a un agent qui va se déplacer dans le labyrinthe, il est important que cet apprentissage puisse se faire de manière progressive. On va devoir par conséquent trouver une manière de résoudre Navier-Stokes dans ces conditions. Le souci est que, quand on arrive dans un cul de sac, les équations de Navier-Stokes (notamment la divergence) disent que toute la composante doit avoir une vitesse de 0 (eau stagnante car elle n'est pas dans le courant qui relie l'entrée à la sortie). Or si on se permet de faire une propagation de cette information dans les neurones pour la faire remonter et changer ceux de toute l'aile, ce ne serait pas un coût énorme par rapport à un réseau de neurones classique qui nécessite à chaque apprentissage de rétro-propager de l'information à travers tous ses neurones. Cependant on peut faire mieux : dans tous les cas l'agent devra ressortir de l'impasse donc autant corriger les valeurs de vitesse au fur et à mesure qu'il remonte. De plus dans le cas de croisements, on peut avoir des problèmes 6.

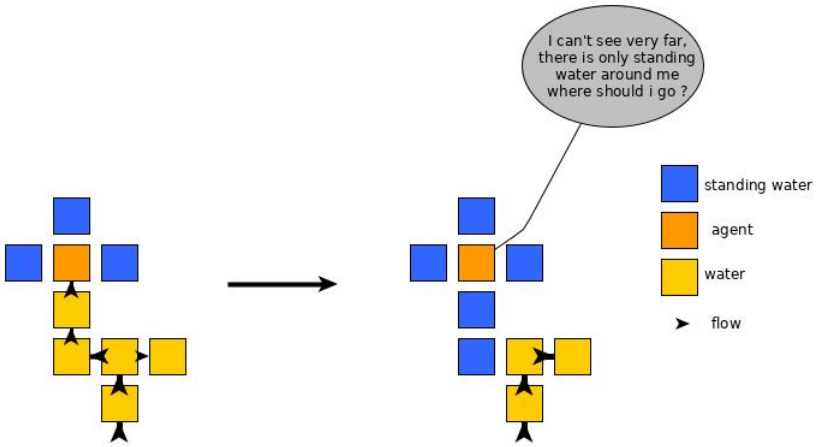


Figure 6: Problème pour sortir des impasses : croisement

Or si on résolvait de manière séquentielle, on pourrait faire en sorte que l'agent suive le courant au fur et à mesure et le remonte (c'est une optimisation dont je reparlerai plus tard). Il y a cependant un problème potentiel à résoudre la divergence en cas de cycle de manière séquentielle 7.

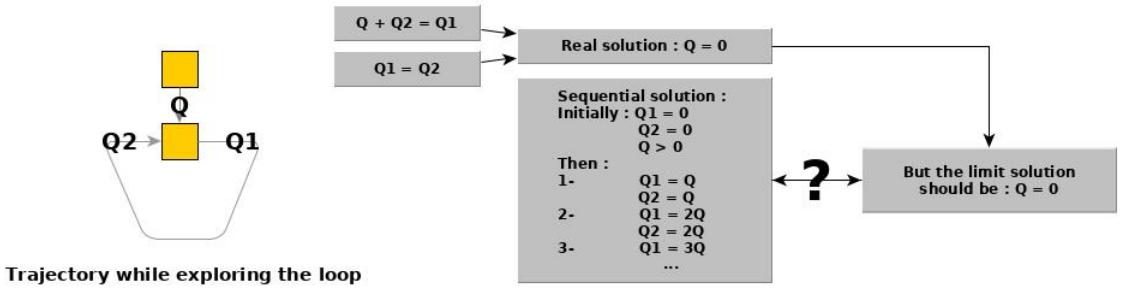


Figure 7: Divergence dans les cycles

En effet, en fonction de notre façon de résoudre les équations, les valeurs de vitesse pourraient s'emballer et l'agent pourrait se retrouver bloqué dans la boucle. Il faudra donc bien y réfléchir dans la suite. Enfin, la dernière équation n'est clairement pas triviale donc je vais me contenter de la discrétiser et de voir ce que cela va donner séquentiellement. Elle contient toute la complexité des équations de Navier-Stokes et est alors la raison de la plupart des problèmes depuis le début, il faut donc la manipuler avec précaution.

3.5 Vers des équations plus discrètes

Après avoir fait une première implémentation d'une résolution de Navier-Stokes, je me suis rendu compte que ces équations se sont vraiment résolubles numériquement que dans des environnement très continus. En effet, une simple imprécision peut avoir des effets énormes sur la suite de la résolution. J'ai alors pensé à essayer d'augmenter la résolution de mon environnement afin d'avoir des solutions moins approchées et plus continues, puis de faire une moyenne sur chaque case mais cela restait assez compliqué notamment vu le nombre d'inconnues dans les équations. Je suis parti des équations de Navier-Stokes dont les solutions étaient notre espace de recherche. Or la résolution de ces équations n'est pas facile sur un espace discret. Ainsi il va donc falloir changer notre façon de caractériser cet espace de solutions. Il est parfois difficile d'avoir un espace de recherche qui soit aussi petit que possible (juste l'ensemble des solutions du problème serait parfait mais souvent difficile à caractériser et à résoudre). On a alors 2 possibilités : une transformation "safe" ou une transformation "unsafe" 8. On pourrait

bien entendu penser à une transformation hybride entre les 2 mais on pourra faire mieux ici.

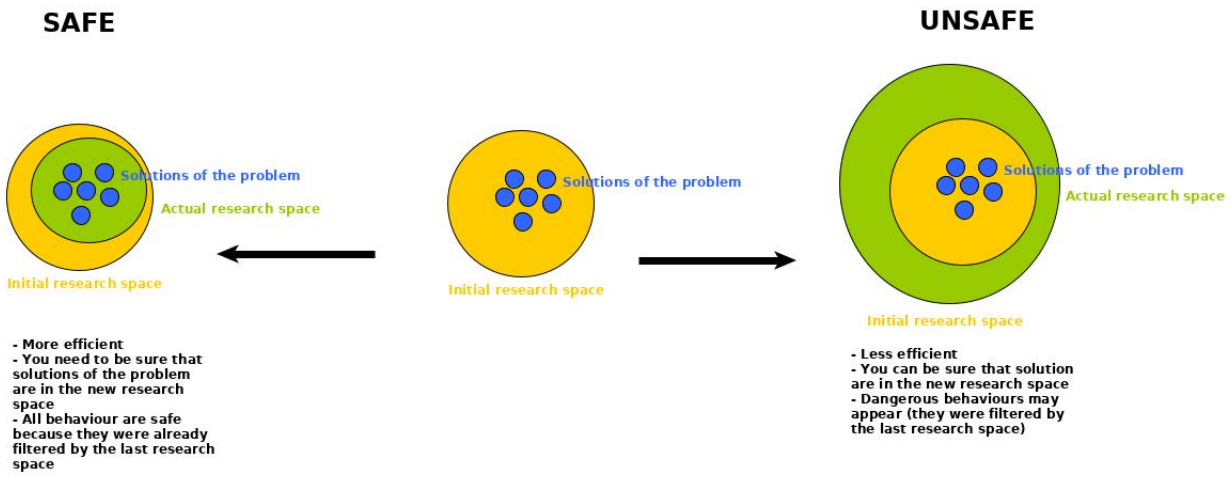


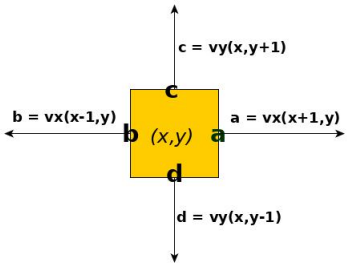
Figure 8: Transformer des équations

L'idéal est donc de faire une transformation "safe". Pour cela on va faire des équations qui suivent la logique de la mécanique des fluides et on va considérer chacun des cas possibles et envisager les différents comportements autorisés par la mécanique des fluides. On en déduit ces équations suite à plusieures factorisations que je n'ai pas la place d'expliciter ces équations :

$$div(\overrightarrow{v}) = \begin{cases} \alpha & \text{à l'entrée} \\ -\alpha & \text{à la sortie} \\ 0 & \text{sur le reste des cases} \end{cases} \tag{5}$$

$$\overrightarrow{v} = \overrightarrow{ave}\,\overrightarrow{v} + \lambda\overrightarrow{v_{learn}} + \overrightarrow{dev} \tag{6}$$

avec, pour la notation suivante :



les expressions suivantes :

$$\overrightarrow{v} = \begin{pmatrix} vx \\ vy \end{pmatrix}$$

$$\overrightarrow{ave}\,\overrightarrow{v} = \begin{pmatrix} \frac{a+b}{2} \\ \frac{c+d}{2} \end{pmatrix}$$

$$\overrightarrow{dev} = \mu.sv(vx.vy).([\mathbf{1}_{\mathbb{R}^*_-}(c.vy) + \mathbf{1}_{\mathbb{R}^*_-}(d.vy)].\frac{\|vy\|}{2} - [\mathbf{1}_{\mathbb{R}^*_-}(a.vx) + \mathbf{1}_{\mathbb{R}^*_-}(b.vx)].\frac{\|vx\|}{2}).\begin{pmatrix}1\\-1\end{pmatrix}$$

$$\mu = \mathbf{1}_{\mathbb{R}_+}(-1 + \sum_{\substack{(i,j)\in[-1,0,1]\\ i.j=0}} P(x+i, y+j))$$

$$P(x,y) = \begin{cases}1 & \text{si il y a un obstacle en x y}\\ 0 & \text{sinon}\end{cases}$$

$$sv(x) = \begin{cases}sgn(x) & sgn(x) \neq 0\\ random(\{-1,1\}) & sinon\end{cases}$$

Le terme de divergence indique simplement une conservation de la matière. Enfin la deuxième équation se compose en 3 termes : un terme de continuité:

$$\overrightarrow{ave}\overrightarrow{v}$$

un terme d'apprentissage forcé:

$$\lambda\overrightarrow{v_{learn}}$$

et un terme d'exploration :

$$\overrightarrow{dev}$$

Le terme de continuité sert à essayer de conserver une continuité de vitesse entre les cases. Cependant ce terme n'est pas parfait et n'assure pas une conservation de la matière sans failles ce qui posera des problèmes pratiques plus tard (on pourrait ajouter un terme de correction assez facilement mais je voulais rester sur quelque chose de relativement simple). Le terme d'apprentissage forcé permet d'augmenter la valeur de vitesse dans un sens ou dans l'autre en fonction de ce que l'on veut faire. Il est donc plutôt là pour inciter le réseau de neurones à considérer une solution particulière allant plutôt dans une direction à partir d'un point. Enfin le dernier terme permet de faire en sorte que le réseau neuronal explore par lui même (voir chapitre suivant), ce qui rend justement ce facteur d'apprentissage anecdotique. Le plus gros du travail sera fait par le pouvoir de prédiction du réseau neuronal et ses suggestions d'exploration. On a donc intégré une partie de l'apprentissage par renforcement au réseau neuronal qui saura lui-même quand il a besoin d'explorer. Il s'avère que cet ensemble de contraintes que l'on va résoudre est exactement l'ensemble des solutions du problème du "labyrinth" (terme anglais) c'est à dire l'ensemble des labyrinthes n'ayant qu'une solution. Nous verrons cependant par la suite que nous avons perdu quelques points bleus lors de la résolution de "maze" c'est à dire quand il y a plusieurs solutions au labyrinthe. On a donc fait une transformation "safe" pour le "labyrinth problem" optimale (on a exactement les solutions) mais une transformation hybride qui contient des comportements dangereux pour la résolution (comme par exemple se retrouver bloqué dans un cycle).

3.6 Résolution des équations de "pseudo Navier Stokes"

Maintenant que nous avons des équations satisfaisantes, il va falloir essayer de les résoudre afin de tirer un réseau neuronal de cette résolution. Le principe de l'apprentissage séquentiel est que nous n'allons regarder que les cases adjacentes à notre agent et résoudre ces équations sur ces cases. Pour qu'elles soient résolues de manière plus pertinente, vu que l'agent est en mouvement, on considère qu'il arrive d'une case adjacente et que l'on veut résoudre les équations sur la nouvelle case sur laquelle il arrive. On suppose donc que la valeur de la case depuis laquelle l'agent vient est correcte. Elle sera donc considérée comme étant une constante, ce qui permettra de véritablement propager de l'information dans le labyrinthe. De plus on suppose que la vitesse dans les murs est de 0 (donc constante), ce que l'on peut écrire en modifiant la première équation de la forme (mais elle est un peu moins digeste) :

$$\sum_{i\in[-1,1]} vx(x+i,y)(1-P(x+i,y)) + vy(x,y+i)(1-P(x,y+i)) = \begin{cases}\alpha & \text{à l'entrée}\\ -\alpha & \text{à la sortie}\\ 0 & \text{sur le reste des cases}\end{cases}$$

On pourrait aussi garder l'ancienne équation et ajouter celle ci, beaucoup plus digeste mais moins explicite à résoudre :

$$P\overrightarrow{v} = \overrightarrow{0}$$

Ainsi pour résoudre la divergence, si on est dans un couloir, on a 1 mur de chaque coté, et on connait la case de derrière, donc il ne reste qu'une inconnue et on peut résoudre les équations sans ambiguïté. On remarque que si on est dans un cul de sac, on n'a pas d'inconnue et seulement dans ce cas on se permet de modifier la case de derrière (qui vaudra 0 par conséquent). Dans les cas de croisements, il existe plusieurs solutions aux équations de Navier-Stokes et il en est de même pour nos équations de "pseudo Navier Stokes". Il va donc falloir faire un choix pour les résoudre. Pour ce faire, on va faire le choix d'augmenter les courants déjà existants proportionnellement à leur force (un courant faible sera peu affecté tandis qu'un courant fort variera beaucoup plus). On va donc utiliser ces formules :

$$v\alpha(x+a, y+b) \mathrel{+}= \begin{cases} \frac{s.\Delta div.\|v\alpha(x+a,y+b)\|}{somme} & \text{si somme} \neq 0 \rightarrow \text{terme d'exploitation et de renforcement} \\ \frac{s.\Delta div}{nb\_couloirs} & \text{si somme} = 0 \rightarrow \text{terme d'exploration équitable} \end{cases}$$

avec a et b dans [-1,0,1] tels que a.b = 0 et α pouvant signifier x ou y

$$s = a + b \rightarrow \text{désigne le signe de variation de la grandeur}$$

$$\Delta div = -div(\overrightarrow{v})(x,y) + \begin{cases} \alpha & \text{(x,y) est l'entrée} \\ -\alpha & \text{(x,y) est la sortie} \\ 0 & \text{sinon} \end{cases}$$

$$somme = \sum_{i\in[-1,1]} \|vx(x+i,y)\|(1-P(x+i,y)) + \|vy(x,y+i)\|(1-P(x,y+i))$$

$$nb\_couloirs = \sum_{\substack{(i,j)\in[-1,0,1] \\ i.j=0}} P(x+i,y+j)$$

Avec ces formules on trouve les nouvelles valeurs des neurones proches en faisant varier a et b (on ne doit pas essayer de résoudre la divergence dans un mur (si (x+a,y+b) est un mur), c'est écrit dans les équations).
Maintenant que l'on connaît les valeurs des neurones proches, on va pouvoir trouver la valeur du neurone en (x,y). Pour cela il suffit simplement d'utiliser la formule donnée précédemment. Attention cependant, les valeurs de vitesse pour les cases adjacentes qui apparaissent dans le terme de déviation ($\overrightarrow{dev}$) sont celles **avant** la correction de la divergence (sinon ça marche beaucoup moins bien). Le code est sur le github et vous permettra de voir comment cela fonctionne.

3.7 D'une résolution séquentielle sans propagation vers un réseau neuronal

Le sujet du stage reste de résoudre le problème avec un réseau de neurones. Il va donc falloir s'y ramener mais tout le travail effectué jusqu'à maintenant a été fait pour rendre cette tâche plus facile. En effet, on a une résolution locale des équations (c'est pourquoi les équations différentielles peuvent être intéressantes pour faire des réseaux de neurones en apprentissage séquentiel). Ceci permet de résoudre l'équation sur chaque case en n'interagissant que avec ses voisins. Il faut tout de même que ces cases soient accédées tour à tour selon une trajectoire car les équations ont été faites pour un apprentissage séquentiel. On peut donc imaginer un plan avec des motifs élémentaires qui organiseraient leur voisinage immédiat en communiquant entre eux tout en respectant le procédé décrit ci-dessus qui ne donne que des solutions des équations de "pseudo Navier Stokes". J'ai donc construit le graphe de ces calculs afin de dessiner des machines élémentaires qui seront nos neurones.

(cf Annexes)

Contrairement aux neurones classiques, ces neurones sont composés de plusieurs *organites* c'est à dire qu'ils ne font pas que de modifier leur valeur puis de la transmettre aux autres mais il peuvent aussi servir d'intermédiaires et prendre des données aux neurones voisins, faire un calcul dessus et leur renvoyer le résultat du calcul sans modifier leur propre valeur.

Nous avons donc bien réussi à faire un réseau de neurones qui ne peut donc apprendre que des solutions des équations de "pseudo Navier Stokes".

3.8 Lien avec l'apprentissage par renforcement

Maintenant que nous avons résolu notre problème de Deep-Learning, il va falloir raccorder notre solution à notre algorithme d'apprentissage par renforcement. On peut faire apprendre des valeurs de vitesse au réseau de neurones mais cela sert plus à orienter le flux à travers le labyrinthe qu'à réellement se souvenir de la valeur exacte des vitesses donnée. Ceci vient du fait que le réseau de neurones à déjà un terme d'exploration incorporé et que ce terme est déjà régulé par le réseau lui-même. Il va donc falloir un peu modifier l'algorithme d'apprentissage par renforcement si on veut avoir de bons résultats. En effet, il va falloir tenir compte de ces capacités d'anticipation et d'exploration du réseau de neurones qui le rendent bien meilleur que la plupart des réseaux de neurones actuels qui se contentent uniquement de mémoriser. Ces compétences vont se manifester par une proposition de trajectoire immédiate (juste à coté de la case sur laquelle est l'agent) que l'agent pourra suivre ou non. La seule chose que l'agent a à faire est de suivre ce que le réseau de neurone lui indique de faire. Il est possible si souhaité d'ajouter une influence sur cette exploration / exploitation en utilisant le vecteur d'apprentissage forcé afin d'influencer les choix du réseau de neurones. Attention cependant à ne pas donner d'informations erronées (comme foncer dans un mur ou revenir en arriere) avec une norme trop importante sinon cela pourrait créer un bug au niveau de la logique de résolution du réseau de neurones qu'il devra essayer de fixer lui-même. C'est pourquoi il est peu recommandé d'utiliser ce terme d'apprentissage forcé.
Il reste à déterminer comment, à partir du réseau neuronal, choisir quelle direction prendre. La façon dont on va choisir la prochaine case sur laquelle aller à partir du réseau de neurones va dépendre non seulement de la case actuelle mais aussi des cases connexes : on détermine un score pour chaque axe : x et y :

$$scorex = \begin{cases} 0 & sgn(x) = 0 \\ vx(x,y).vx(x+sgn(vx),y) & sinon \end{cases}$$

$$scorey = \begin{cases} 0 & sgn(y) = 0 \\ vx(x,y).vx(x,y+sgn(vy)) & sinon \end{cases}$$

Une fois ces grandeurs calculées, on prend la plus grande, ce qui permet de choisir l'axe du mouvement, puis il reste à choisir le sens : on choisit selon le signe de la grandeur choisie. Voici un algorithme qui résume tout cela et qui traite des cas particuliers:

Algorithm 1 Select movement

```
VX ← vx(x, y)
VY ← vx(x, y)
scorex ← vx(x + sgn(VX), y) * VX
scorey ← vx(x, y + sgn(VY)) * VY
if abs(scorex) > abs(scorey) then
  if sgn(scorex) == 1 then
    move(right)
  else
    move(left)
  end if
else
  if abs(scorey) == abs(scorex) and scorex == 0 then
    r_move ← choose_a_random_possible_move()
    move(r_move)
  else
    if sgn(scorey) == 1 then
      move(top)
    else
      move(bottom)
    end if
  end if
end if
```

Le cas où scorex = scorey = 0 arrive quand l'agent a trouvé une impasse et essaie donc de remonter le cours d'eau pour retrouver du courant. C'est ici que l'on va pouvoir aider le réseau de neurones car il a raison de dire

que les vitesses sont nulles mais il ne faut pas que l'agent se perde en cas de croisement (cf partie 3.4). Il suffit donc, avant de résoudre la divergence, de chercher une case de vitesse non nulle à proximité et de s'y déplacer si la valeur de vitesse de la case actuelle est 0. Cette opération se fait en temps et espace constant donc je me suis permis de l'implémenter sur le github. La convergence est ainsi plus rapide.

3.9 Limites techniques de l'implémentation

L'implémentation d'un tel algorithme pose plusieurs problèmes technique dont un majeur qui n'affecte pas le fait de trouver une solution mais qui gène en cas d'exploration de structures très arborescentes dans le labyrinthe. En effet, il y a une faille dans les équations que j'ai trouvées (j'en avais déjà parlé dans le chapitre consacré à leur introduction). La conservation de la matière n'est pas respectée parfaitement par la seconde équation. Si rien n'entrait en conflit avec cela, tout fonctionnerait toujours bien (j'avais prévu l'éventualité en concevant les équations, ce n'était pas grave pour la résolution). Or régulièrement, quand l'agent arrive dans un cul de sac, il doit mettre les valeurs de ce couloir à 0. Ce qui fait qu'il calcule le terme vu précédemment pour résoudre l'équation de divergence. Cependant ce terme qui doit être l'opposé de la valeur actuelle sur la case a été calculé différemment de la valeur actuellement sur la case. Il en résulte un problème bien connu des flottants en informatique : on n'obtient pas parfaitement 0. Or l'algorithme fait une très grande différence entre les valeurs valant 0 (pas de courant) et celles ne valant pas 0 ce qui fait qu'il va falloir "cut" ces valeurs trop faibles pour les mettre à 0. J'ai finalement réussi à distinguer les valeurs qu'il faut mettre à 0 de celles qu'il ne faut pas. Donc ce problème ne se pose plus.

3.10 Résultats et comparaison au Q-Learning classique

J'ai comparé la méthode de résolution par les neurones de Navier-Stokes au Q-Learning tabulaire et les résultats sont assez impressionants (cf Annexes). Pour des soucis d'équité, j'ai implémenté quelques améliorations à l'algorithme de Q-Learning : que l'on arrive sur une case du labyrinthe depuis le haut, la gauche, la droite ou le bas n'a aucune importance au niveau de la récompense associée. Ainsi au lieu d'associer une récompense à un couple (état, action), je n'ai compté que l'état dedans. De plus, j'ai empéché l'agent de faire demi-tour dans le labyrinthe (j'ai gardé en mémoire sa derniere position et il ne pouvait plus y aller en explorant à moins qu'il n'ait pas d'autres alternatives). Pour faire du Q-Learning tabulaire on utilise la loi de mise à jour de la table suivante (Q étant le tableau des récompenses, r la récompense réelle et move(s,d) la fonction qui a une position s et une direction d associe la nouvelle position s' suite au mouvement):

$$Q[s] \mathrel{+}= \alpha(r + \gamma.max_d Q[move(s,d)] - Q[s])$$

avec pour mes calculs : $\alpha = 0.3$ et γ=0.92. Une récompense de 1 à l'arrivée et de -0.001 sur toutes les autres cases (afin qu'il trouve le meilleur chemin). L'implémentation est sur le github. Les courbes analysent les performances des 2 apprentissages sur 3 labyrinthes 25x25 (cases) différents générés aléatoirement selon l'algorithme de Wilson. Chaque courbe représente 150 tests d'apprentissages depuis 0 sur 500 résolutions complètes de labyrinthes (une résolution est appelée un *episode*) et montre le nombre de pas dans le labyrinthe pour le résoudre (en échelle logarithmique). Chaque courbe est affichée avec un paramétre d'opacité de 0.01 ce qui donne une figure de densité afin de voir la fiabilité de chaque méthode. L'algorithme de Navier-Stokes fonctionne également très bien sur des labyrinthes plus grands : j'ai essayé 100x100 mais je ne peux pas tester sur beaucoup plus grand car l'algorithme de Wilson a une complexité exponentielle. Les 25 000 labyrinthes générés par cet algorithme en prévision du stage sont disponibles sur le github (je n'ai pas pu tous les tester faute de temps et de puissance de calcul, je ne me suis donc interessé qu'aux premiers). On peut noter quelques pics au niveau de la courbe verte après le premier. Ceux-ci sont liés au problème des flottants qui fait que l'agent se perd dans des zones d'eau stagnante.

3.11 Conséquences de ces travaux

Ces travaux ont permis de créer un réseau de neurones fait spécifiquement pour les besoins du problème à partir de contraintes sous formes d'équations. Cette méthode diffère de beaucoup des autres méthodes similaires dans le sens où, on n'utilise pas la fonction de loss des réseaux de neurones classiques pour approcher des solutions [3] mais on est certain à 100% que les contraintes sont respectées. Ces contraintes offrent au réseau de neurones un pouvoir prédictif très puissant lui permettant de résoudre le labyrinthe de manière bien plus efficace que un Q-Learning tabulaire ou un Deep Q-Learning (avec un réseau de neurones). On pourrait alors imaginer appliquer ce procédé à d'autres problèmes sur lesquels on pourrait faire de plus grandes avancées, comme par exemple les

échecs. En effet, le plus difficile ces temps-ci en apprentissage par renforcement, c'est de traiter des problèmes avec beaucoup d'actions possibles. Il est alors difficile pour l'agent de les tester toutes rapidement ce qui fait que la phase d'exploration est très longue. En ajoutant des contraintes pour guider l'agent, on pourrait drastiquement réduire le nombre d'actions nécessaires pour résoudre un problème. Par exemple, actuellement, AlphaZero, l'IA championne du monde aux échecs, shogi et Go utilise 2 fois plus d'output que le nombre de cases de l'échiquier (soit 128 pour les échecs). Or aux échecs, on peut coder un mouvement en 4 coordonnées entre 1 et 8. Il s'agirait donc de faire un réseau sous contraintes avec 4 output et avec pour contraintes : les outputs sont dans [1,8], les 2 premières sorties codent la case de départ du mouvement donc une pièce du joueur IA doit s'y trouver, les 2 deuxièmes sorties codent la case d'arrivée du mouvement donc il faut que ce mouvement soit valide (pas de pièce alliée à l'arrivée et il faut que le déplacement soit possible avec la pièce choisie). Bien entendu écrire formellement ces contraintes et trouver un réseau qui les satisfait ne sera pas une chose triviale mais c'est une poursuite possible de ce travail sur les labyrinthes et Navier-Stokes en généralisant la méthode pour trouver des réseaux de neurones. De plus il n'est pas certain que ces contraintes balisant l'espace de recherche accélèrent réellement l'apprentissage. En effet, on a déjà travaillé sur certains problèmes que l'on a réussi à transformer de façon à ne pas pouvoir apprendre de solution inefficace (on ne considère que les coups possibles aux echecs et au Go par exemple). Ainsi est-ce que faire un réseau de neurone qui, à une situation donnée, ne peut donner directement que des coups possibles accélèrera vraiment l'apprentissage [4]? Ceci reste une question importante du domaine de recherches qui se dessine. En effet pour accélérer un apprentissage il existe 3 façon de faire : 1- faire les calculs plus vite afin de simuler plus de parties d'echecs 2- trouver des stratégies d'exploration et d'exploitation plus efficaces afin de tirer le maximum de chaque partie pour survoler plus intelligemment l'ensemble des possibles (nécessite d'avoir une très bonne compréhension en tant qu'humain de l'environnement) 3- Garantir des performances (empécher l'apparition de comportements dangereux par exemple).
La méthode présentée ici repose sur un couplage des 2 dernières méthode c'est à dire qu'on connait assez bien le problème du labyrinthe en tant qu'humain et donc on peut créer un cerveau artificiel capable de traiter des données fournies au fur et à mesure de l'exploration plus efficacement. Or ces calculs sont plus complexes et sont donc un peu plus lent que le Q-Learning tabulaire classique. Ainsi le gain en temps n'est pas si enorme comparé au Q-Learning classique pour des petits labyrinthe. L'interêt est bien plus grand sur de gros labyrinthes.
Une autre possibilité extrêmement utile vient du fait que les roboticiens rejettent en bloc l'utilisation du deep learning car ils ont peur que leur robot puisse faire du mal à des humains. En effet le deep learning n'est jamais fiable à 100%. Cependant avec cette méthode, on pourrait développer des réseaux de neurones fiables cette fois-ci à 100% et faits spécifiquement pour une tâche (donc à apprentissage très rapide ce qui résout un autre problème de l'utilisation du deep learning en robotique : il risque d'y avoir des dégats rapidement si l'apprentissage est trop long ou pas assez efficace). Bien entendu, il reste toujours une partie très difficile : écrire formellement les contraintes permettant d'être certain que le robot ne fasse pas de mal à un humain (et il faudra en extraire un réseau neuronal).

4 Conclusion

Au final, on a bien réussi à extraire un réseau neuronal des équations de pseudo Navier-Stokes. Ce réseau de neurones fait même plus que ce qu'on attendait de lui car il est capable lui-même d'explorer et de réguler son exploration. Il résout donc la plupart des labyrinthes de manière optimale et est bien meilleur que le Q-Learning classique en terme de nombre de pas dans le labyrinthe. On peut visualiser ce procédé comme la création d'un cerveau artificiel (c'est à dire de neurones et d'une structure pour ces neurones) qui permet, par sa structure, un apprentissage beaucoup plus naturel de la tâche donnée. Le problème étant qu'actuellement nous travaillons toujours sur un même support (l'ordinateur). Or un tel traitement des données n'est pas toujours optimal sur ce support ce qui fait que l'on peut plus ou moins efficacement créer de tels cerveaux artificiels. Le mieux serait d'avoir un support physique adapté à ce traitement de l'information (cf nano-neurones [2]).

5 Ouverture

Je pense que le plus important dans ce travail n'est pas le fait de pouvoir mieux résoudre le problème du labyrinthe. Le véritable intérêt et ce qui est vraiment innovant dans cette méthode c'est surtout d'avoir été capable d'extraire un réseau de neurones d'un lot de contraintes. Ce concept inédit permettrait de vraiment améliorer les possibilités du machine learning par la suite en permettant de travailler sur des modèles avec une convergence infiniment

plus rapide, avec moins de données et avec une meilleure robustesse (on a éliminé de nombreuses de fonctions suite à la prise en compte des contraintes). Or même si on donne des contraintes et que l'on en fait un réseau de neurones, si elles ne balisent pas assez fortement le terrain, ce ne sera pas une grande avancée. Cependant si on arrivait à mieux développer ces techniques et si on arrivait à avoir une meilleure compréhension de cette forme d'apprentissage, alors on pourrait avoir de bien meilleurs résultats que les méthodes classiques. On pourrait alors par exemple, battre Google au niveau de la reconnaissance d'images car la quantité de données ne serait plus un problème.
L'idéal serait d'avoir une fonction qui, étant donné un lot de contraintes, soit capable de produire un réseau de neurones particulier, spécifiquement fait pour résoudre le problème lié au lot de contraintes donné et qui renvoie NONE s'il n'en existe pas. Or ce genre de problème est typiquement non calculable (dépend de la formulation) et il y a donc du travail pour essayer de délimiter ce qui est calculable de ce qui ne l'est pas afin de voir si c'est une bonne direction à suivre pour des travaux futurs.

References

[1] Louapre David. La mystérieuse équation de navier-stokes. https://sciencetonnante.wordpress.com/2014/03/03/la-mysterieuse-equation-de-navier-stokes/.

[2] Leroux Hugo. Demain, un ordinateur inspiré de notre cerveau? page 1, jan 2018. https://lejournal.cnrs.fr/articles/demain-un-ordinateur-inspire-de-notre-cerveau.

[3] Magill Martin, Z. Qureshi Faisal, and W. de Haan Hendrick. Neural networks trained to solve differentiale-quations learn general representations. *32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.*, page 1, 2018.

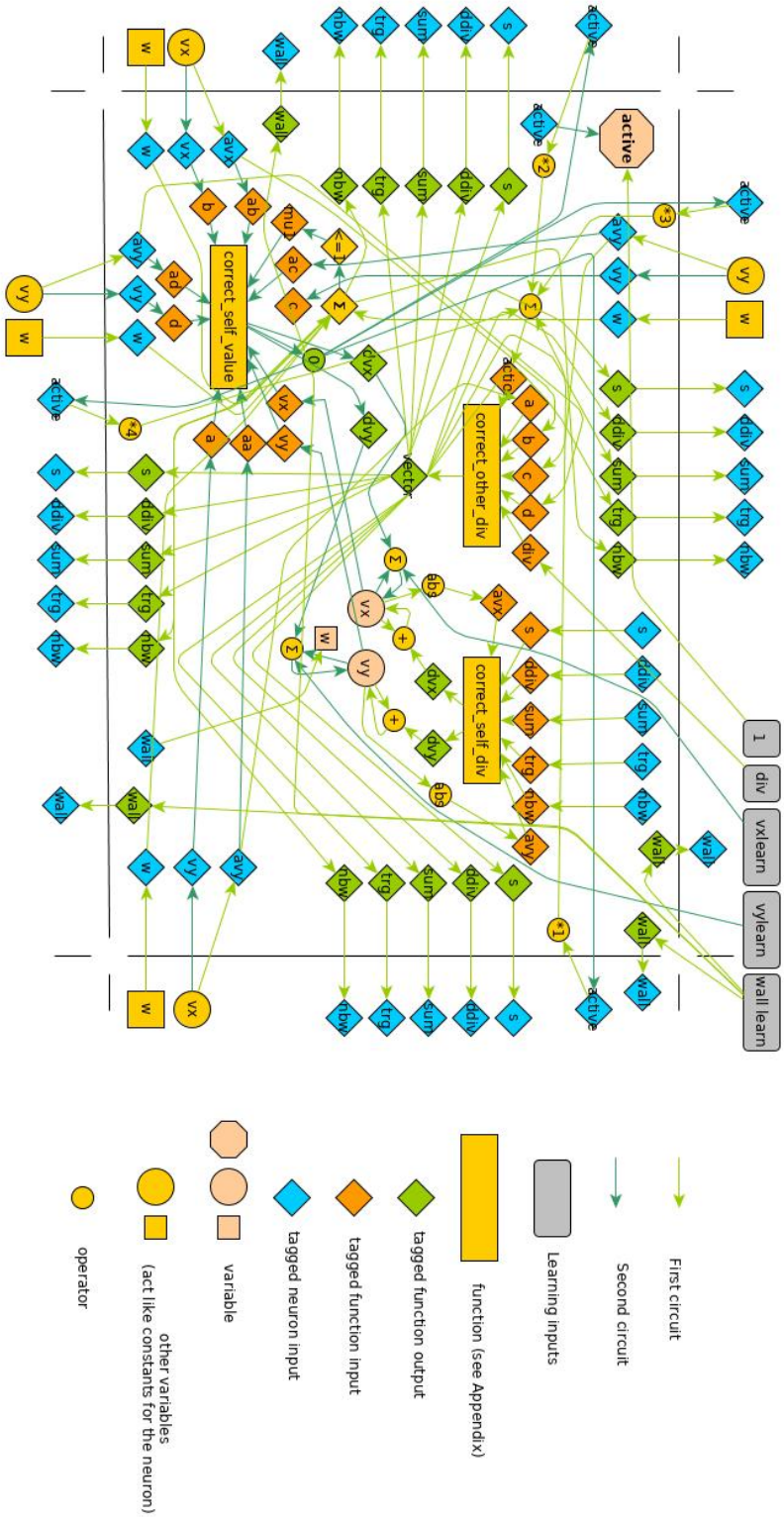
[4] Marquez-Neila Pablo, Salzmann Mathieu, and Fua Pascal. Imposing hard constraints on deep networks: Promises and limitations. page 1, jun 2017.

[5] Zafrany Samy. Deep reinforcement learning for maze solving. https://www.samyzaf.com/ML/rl/qmaze.html.

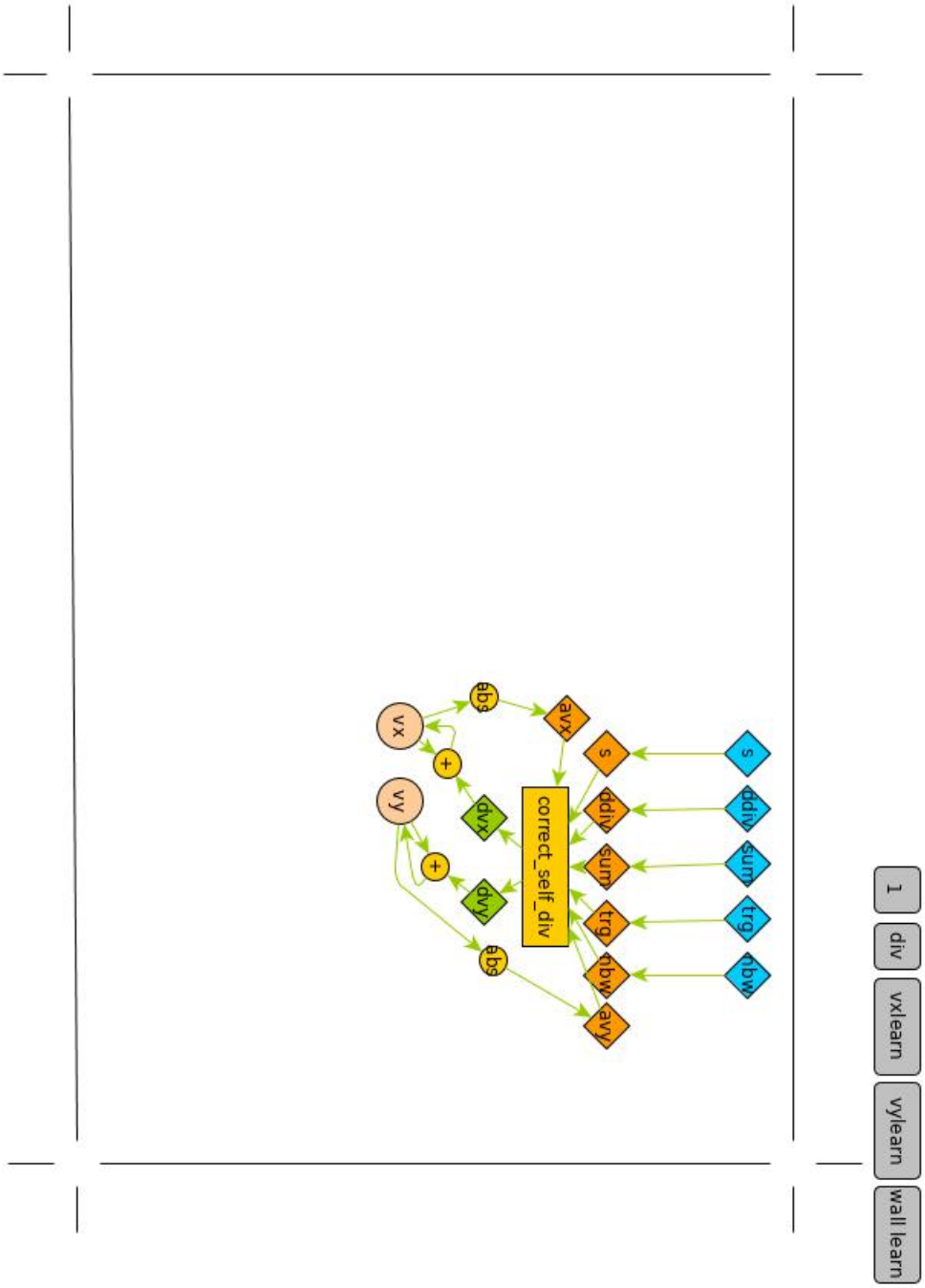
6 Annexes

6.1 pNSNeuron

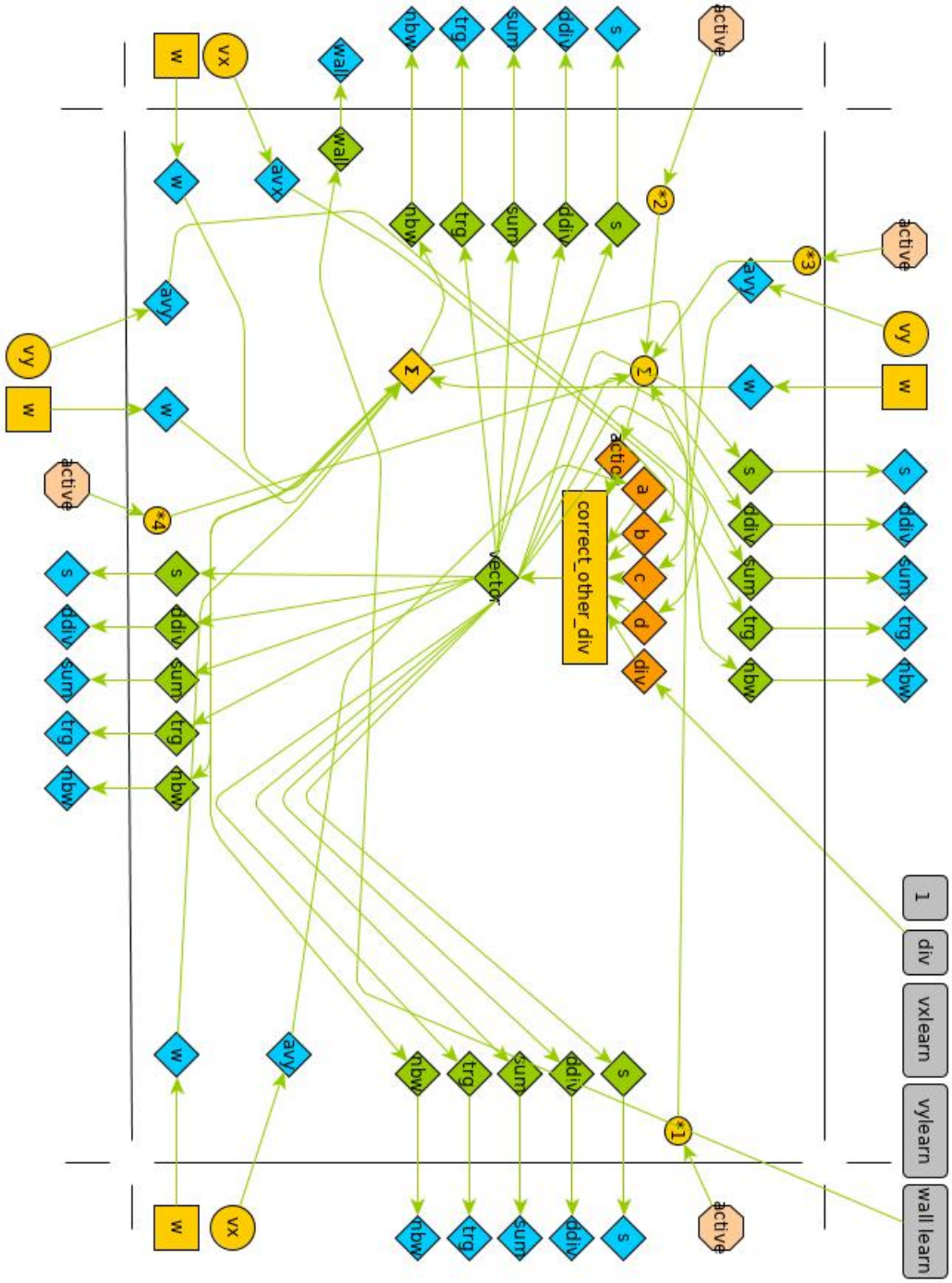
A - pNSNeuron global -



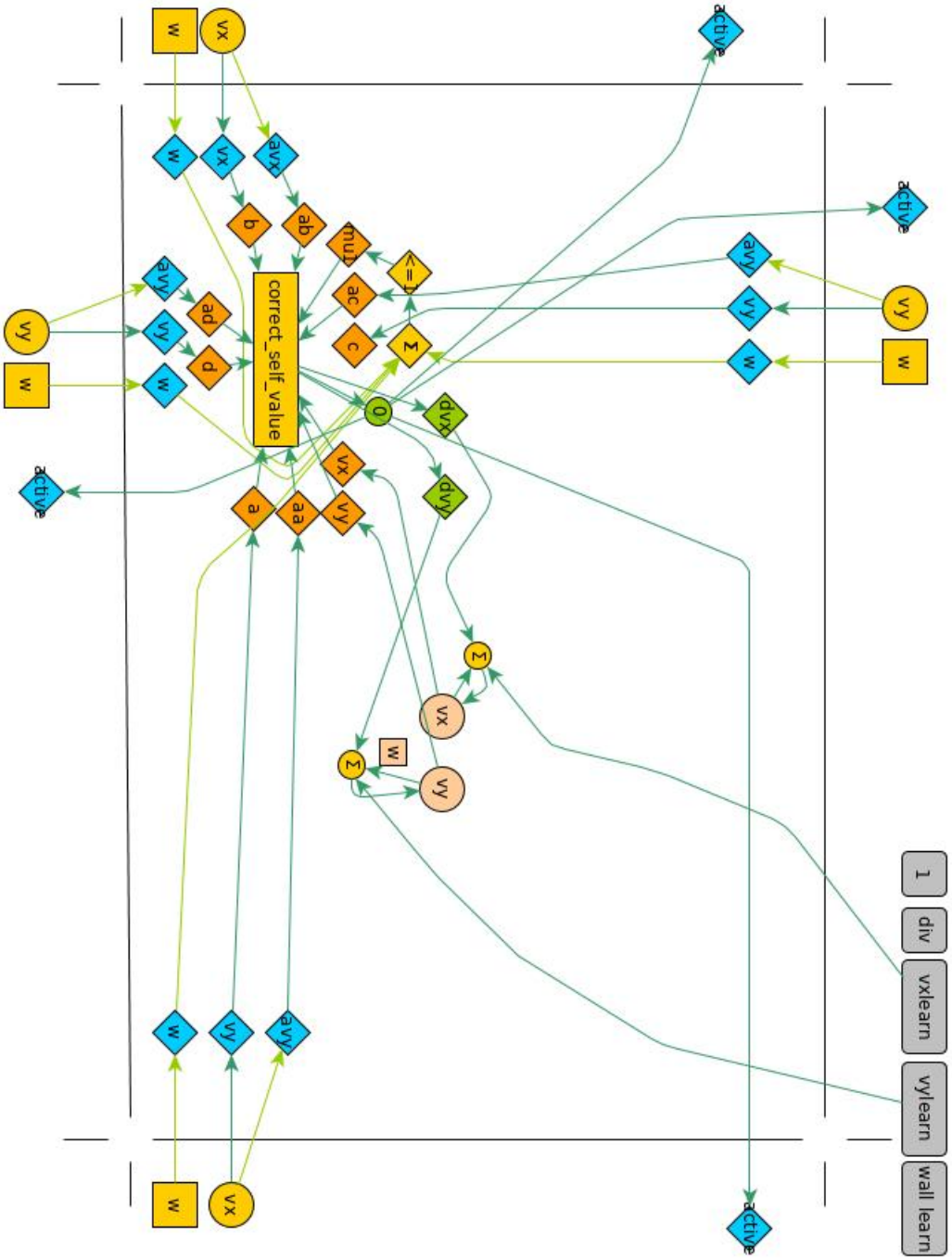
B - Correct Self Div Organelle -



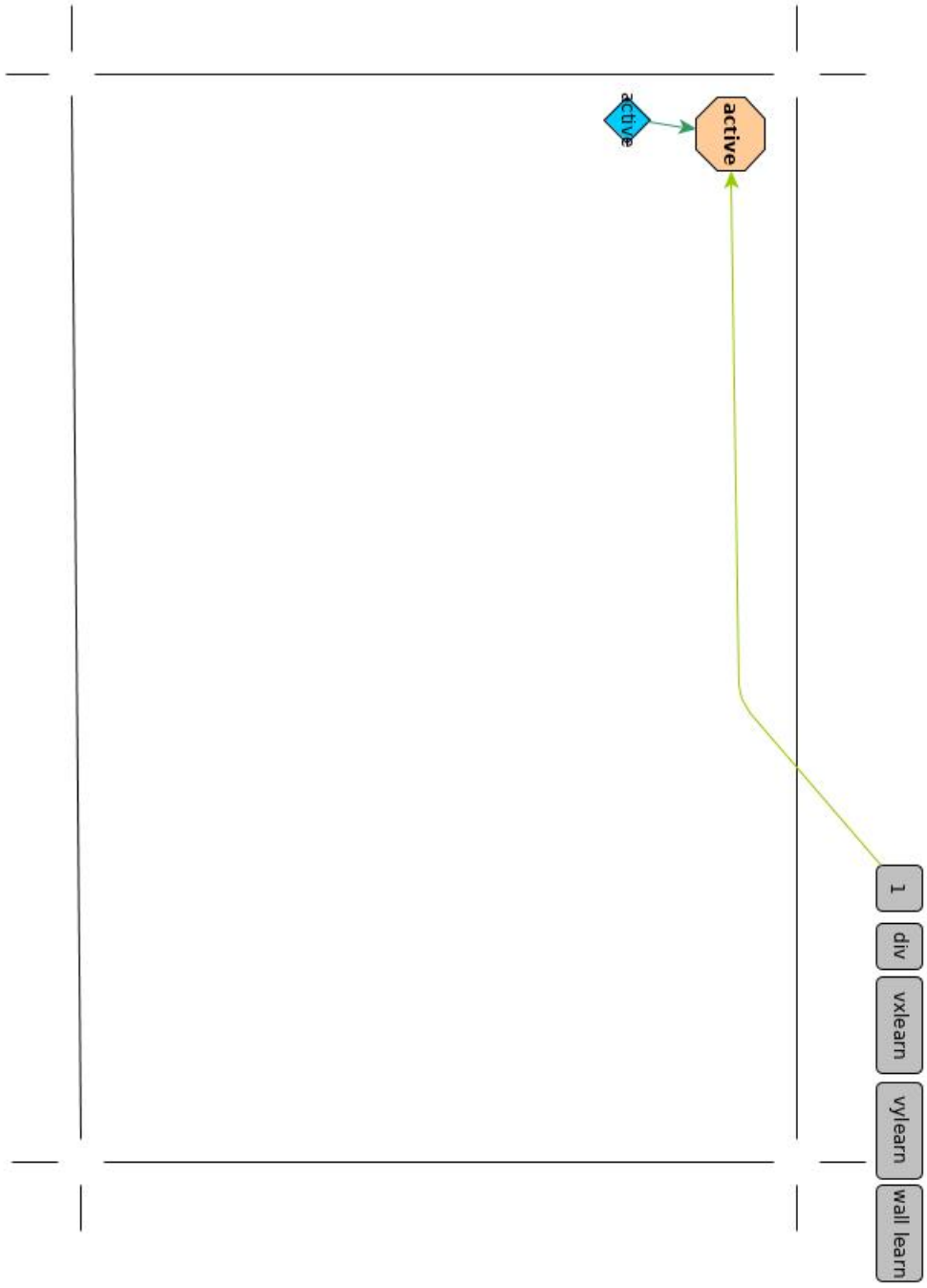
C - Correct Other Div Organelle -



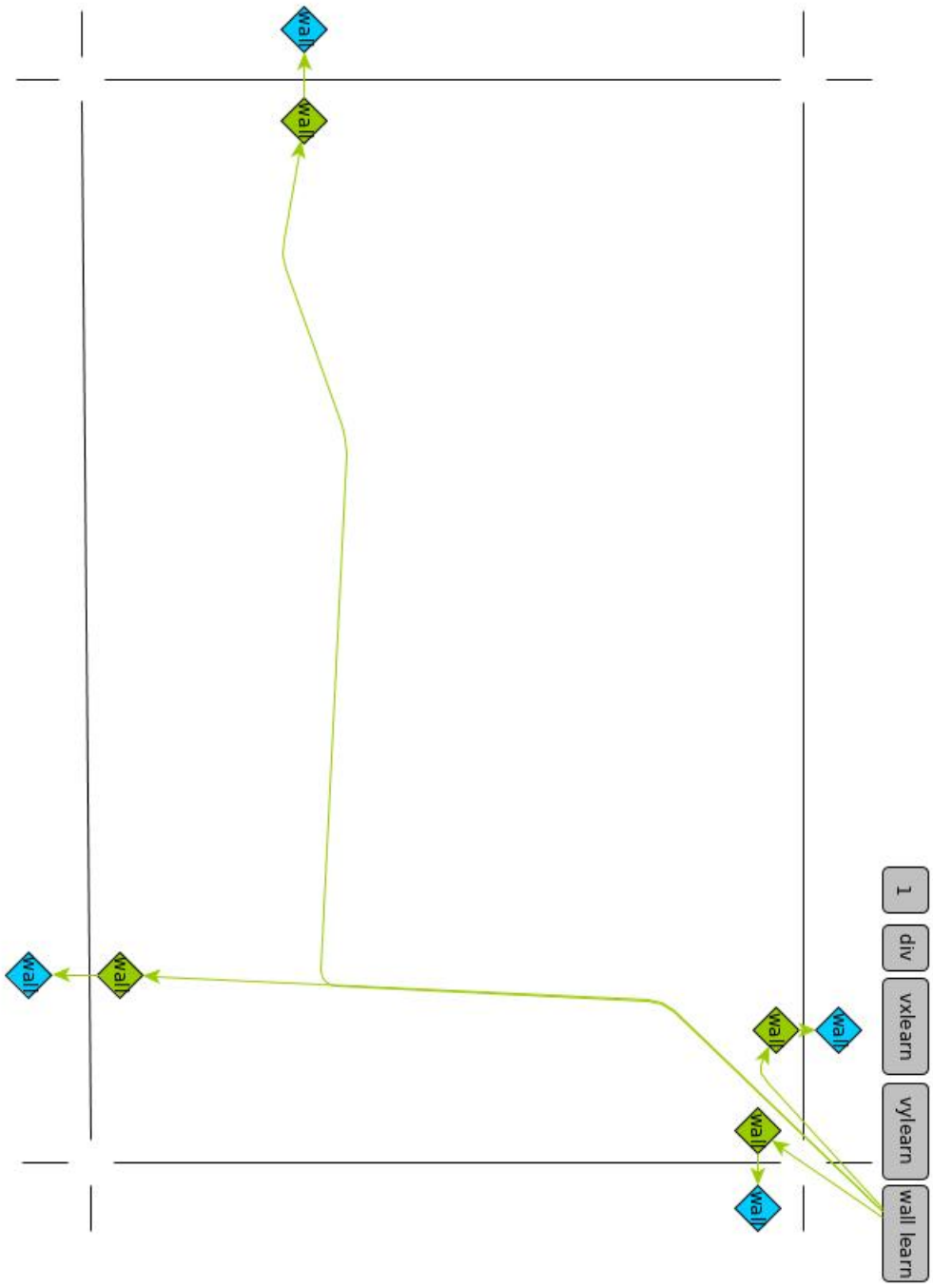
D - Correct Self Value Organelle -



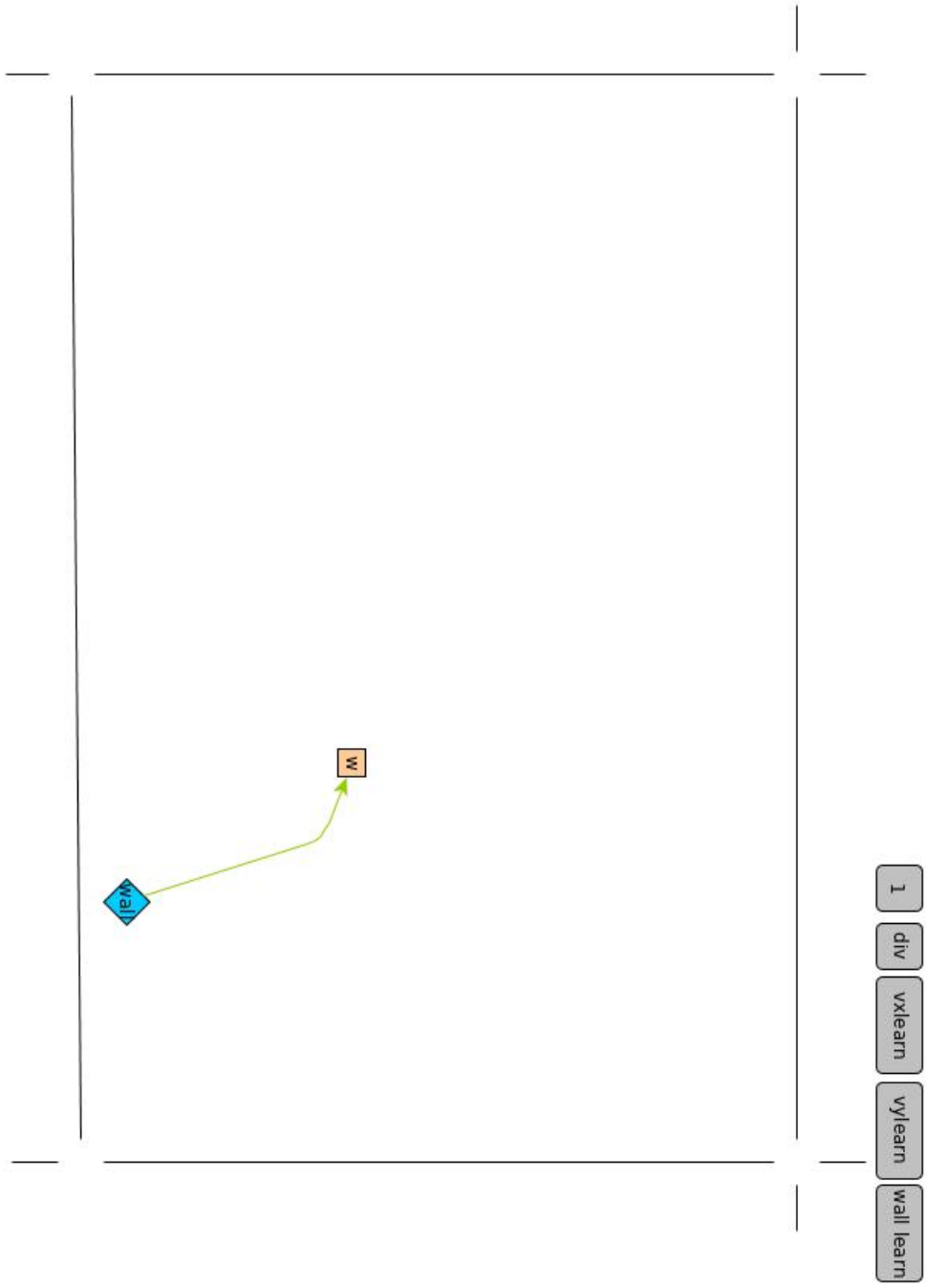
E - Active Organelle -



F - Correct Other Wall Organelle -



G - Correct Self Wall Organelle -



6.2 Functions

Algorithm 2 correct_self_div

Require: s,ddiv,sum,nbw,trg,avx,avy
  if $sum == 0$ **then**
    if $trg == 1$ **then**
      dvx ← s*ddiv/nbw
      dvy ← 0
    else
      dvx ← 0
      dvy ← s*ddiv/nbw
    end if
  else
    if $trg == 1$ **then**
      dvx ← s*ddiv*avx/sum
      dvy ← 0
    else
      dvx ← 0
      dvy ← s*ddiv*avy/sum
    end if
  end if
  return dvx,dvy

Algorithm 3 correct_other_div

Require: a,b,c,d,active
  ddiv ← a-b+c-d
  s ← a+b
  sum ← 0
  if *active* ≠ 0 **then**
    sum ← sum + abs(a)
  end if
  if *active* ≠ 1 **then**
    sum ← sum + abs(b)
  end if
  if *active* ≠ 2 **then**
    sum ← sum + abs(c)
  end if
  if *active* ≠ 3 **then**
    sum ← sum + abs(d)
  end if
  nbw ← wa+wb+wc+wd
  top,bottom ← (s,ddiv,sum,1)
  left,right ← (s,ddiv,sum,0)
  vector ← {"top":top,"left":left,"bottom":bottom,"right":right}
  return vector

Algorithm 4 correct_self_value

Require: a,b,c,d,svx,svy,aa,ab,ac,ad,vx,vy
  avex ← (a+b)/2
  avey ← (c+d)/2
  sv ← sv(vx)*sv(vy)
  dev ← mu*sv*(((ac*vy<0)+(ad*vy<0))*abs(vy)/2-((aa*vx<0)+(ab*vx<0))*abs(vx)/2)
  dvx ← avex+dev
  dvy ← avey-dev
  return dvx,dvy

6.3 Résultats