Rapport de stage de Master 2 :

Méthode multi-échelle avec apprentissage profond adaptatif pour les écoulements

JOUBERT Valentin
Master MACS 2023
Entreprise d'accueil : IMT Nord Europe
Tuteur entreprise : M. Modesar Shakoor / Tuteur université : M. Mehdi Badsi
Avril - Septembre 2023

Nantes Université

IMT Nord Europe
École Mines-Télécom
IMT-Université de Lille

Table des matières

Première partie

Introduction du sujet

1 Méthode multi-échelle

Dans le cadre d'un problème d'écoulement en milieu poreux, la résolution par un solveur éléments finis classique n'est pas très intéressant. En effet, la prise en compte de petits obstacles nécessiterait que les cellules du maillage soit plus fines que ces derniers et cela impliquerait un accroissement du temps de calcul non désirable. Afin de pallier ce problème, on utilise une méthode multi-échelle développée par le CERI MP de l'IMT Nord-Europe (Shakoor & Park, [1]). Pour l'exécution de cette méthode, on effectue deux résolutions de problèmes. La première à grande échelle sur un domaine noté $\Omega^M(x)$ prendra en compte le domaine en son ensemble avec ses conditions limites sans tenir compte des obstacles. Sur chaque point d'intégration du maillage de ce premier domaine, on créé un sous-domaine (noté $\Omega^m(x)$) comprenant un obstacle généré en fonction des caractéristiques du milieu poreux.

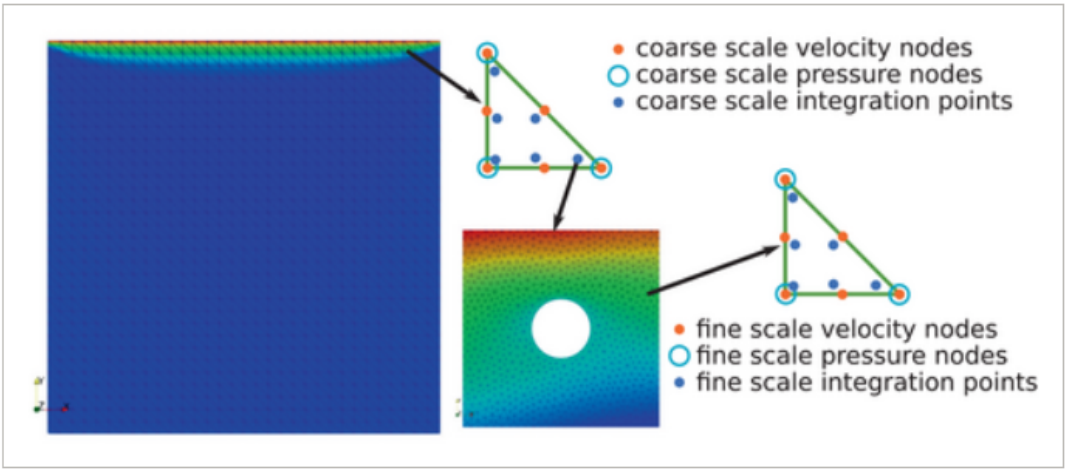

figure 1 : Schéma représentant la structure de la méthode multi-échelle (tiré de [1])

Le problème variationnel à résoudre sur l'échelle la plus grande est le suivant.
Soient des conditions de Neumann t_N^M sur Γ_N^M et conditions de Dirichlet v_D^M sur Γ_D^M

Trouver $v^M(x,t) \in V^M(t), p^M \in L^2(\Omega^M)$ tel que :

$$\begin{array}{l} \int_{\Omega^M}(f^M(x,t).\delta v^M(x) + \sigma^{M,dev}(x,t) : \nabla_x \delta v^M(x))dx \\ + \int_{\Omega^M}(-p^M(x).\nabla_x \delta v^M(x) - \delta p^M(x).\nabla_x v^M(x))dx \end{array} = \int_{\Gamma_N^M} t_N^M(x,t).\delta v^M(x)dx \quad (1)$$

$$\forall \delta v^M \in V^M(t)$$

$$\forall \delta p^M \in L^2(\Omega^M)$$

Avec l'espace

$$V^M(t) = \{w \in H^1(\Omega^M)^d, w(x) = v_D^M(x,t), \forall (x,t) \in \Gamma_D^M \times [0,T]\}$$

$$H^1(\Omega^M) = \{w \in L^2(\Omega^M), \nabla_x w \in L^2(\Omega^M)^d\}$$

Les fonctions $f^M(x,t)$ et $\sigma^{M,dev}(x,t)$ sont des fonctions dépendantes de $v^M(x,t)$ et $\nabla_x v^M(x,t)$ que l'on souhaite déterminer.
On introduit pour cela le deuxième problème sur la plus petite échelle $\Omega^m \subset \mathbb{R}^d$, avec des conditions de Dirichlet homogène autour des obstacles $\Gamma_O^m = 0$ avec $\Gamma_O^m \subset \partial\Omega^m$, qu'il faudra résoudre sur chaque point d'intégration. Ce problème est issu des mêmes équations que le problème à grande échelle avec des conditions pour la conservativité de notre système. On veut que la moyenne de la vitesse calculée sur le domaine le plus fin soit égale à la vitesse calculée sur le point d'intégration associé du plus grand domaine.

$$v^M(x,t) = \frac{1}{|\Omega^m|}\int_{\Omega^m} v^m(y,t)dy \quad (2)$$

$$V^m = \{w \in H^1(\Omega^m)^d, w(x) = 0, \forall (x) \in \Gamma_O^m\}$$

de même pour le gradient vitesse

$$\nabla_x v^M(x,t) = \frac{1}{|\Omega^m|}\int_{\Omega^m} \nabla_y v^m(y,t)dy \quad (3)$$

De plus on introduit la notion de puissance virtuelle (Blanco et al, [7]) sur la grande

et petite échelle (respectivement notées P^M et P^m) avec

$$P^M(\delta V^M, \delta G^M, \delta p^M) = f^M(x,t).\delta V^M + \sigma^{M,dev} : \delta G^M - p^M tr(\delta G^M) \tag{4}$$

$$\forall (\delta V^M, \delta G^M, \delta p^M) \in \mathbb{R}^d \times \mathbb{R}^{d\times d} \times \mathbb{R}$$

et

$$\begin{aligned} &P^m(\delta V^M, \delta G^M, \delta v^m, \delta p^m, \delta\alpha, \delta\beta) = \\ &\frac{1}{|\Omega^m|}\int_{\Omega^m} \begin{bmatrix} \rho(\frac{\partial v^m}{\partial t(y,t)} + v^m(y,t).\nabla_y v^m(y,t)).\delta v^m(y) \\ +2\mu\nabla_y^{S,dev} v^m(y,t) : \nabla_y \delta v^m(y,t) \\ -p^m(y)\nabla_y.\delta v^m(y) - \delta p^m(y)\nabla_y.v^m(y,t) \end{bmatrix} dy \\ &-\delta\alpha.(\frac{1}{|\Omega^m|}\int_{\Omega^m} v^m(y,t)dy - v^M(x,t)) \\ &-\delta\beta : (\frac{1}{|\Omega^m|}\int_{\Omega^m} \nabla_y v^m(y,t)dy - \nabla_x v^M(x,t)) \\ &-\alpha.(\frac{1}{|\Omega^m|}\int_{\Omega^m} \delta v^m(y,t)dy - \delta V^M(x,t)) \\ &-\beta : (\frac{1}{|\Omega^m|}\int_{\Omega^m} \nabla_y \delta v^m(y,t)dy - \delta G^M(x,t)) \end{aligned} \tag{5}$$

$\forall (\delta V^M(x,t), \delta G^M(x,t)) \in \mathbb{R}\times\mathbb{R}^{d\times d}$, $\forall (\delta v^m, \delta p^m) \in V^m \times L^2(\Omega^m)$, $\forall (\delta\alpha, \delta\beta) \in \mathbb{R}\times\mathbb{R}^{d\times d}$

On impose aussi un équilibre entre ces deux valeurs. Finalement, le problème variationnel sur la plus petite échelle est le suivant.
Trouver $(v^m(.,t), p^m, \alpha, \beta) \in V^m(x) \times L^2(\Omega^m) \times \mathbb{R}^d \times \mathbb{R}^{d\times d}$
avec $(v^M(x,t), p^M) \in V^M(t) \times L^2(\Omega^M)$

$$\frac{1}{|\Omega^m|}\int_{\Omega^m} \begin{bmatrix} \rho(\frac{\partial v^m}{\partial t(y,t)} + v^m(y,t).\nabla_y v^m(y,t)).\delta v^m(y) \\ +2\mu\nabla_y^{S,dev} v^m(y,t) : \nabla_y \delta v^m(y,t) \\ -p^m(y)\nabla_y.\delta v^m(y) - \delta p^m(y)\nabla_y.v^m(y,t) \end{bmatrix} dy$$

$$-\delta\alpha.(\frac{1}{|\Omega^m|}\int_{\Omega^m} v^m(y,t)dy - v^M(x,t))$$

$$-\delta\beta : (\frac{1}{|\Omega^m|}\int_{\Omega^m} \nabla_y v^m(y,t)dy - \nabla_x v^M(x,t)) \tag{6}$$

$$-\alpha.(\frac{1}{|\Omega^m|}\int_{\Omega^m} \delta v^m(y,t)dy)$$

$$-\beta : (\frac{1}{|\Omega^m|}\int_{\Omega^m} \nabla_y \delta v^m(y,t)dy) = 0$$

$$\forall(\delta v^m, \delta p^m) \in V^m \times L^2(\Omega^m)$$

$$\forall(\delta\alpha, \delta\beta) \in \mathbb{R}^d \times \mathbb{R}^{d\times d}$$

Il est interessant de remarquer qu'on peut lier f^M, $\sigma^{M,dev}$ et α, β comme indiqué ci-dessous (preuve dans [1]) :

$$f^M = \alpha$$

et

$$\sigma^{M,dev} + p^M I = \beta$$

On peut donc obtenir f^M et $\sigma^{M,dev}$, nécessaires à la résolution sur la grande échelle, en résolvant le problème sur les petites. Toutefois, la résolution par éléments finis du problème variationnel à petite échelle (2) sur tout les points d'intégrations du domaine Ω^M et à chaque pas de temps est à proscrire puisqu'elle entraînerait une augmentation du temps de calcul non désirable.
C'est dans ce cadre que s'inscrit mon stage dont l'objectif est de créer un réseau de neurones capable de prédire α et β sans passer par la résolution éléments finis.

2 Réseau de neurones artificiel : définition

Un réseau de neurones artificiels est un système permettant, à partir de paramètres d'entrées, de prédire un résultat voulu.

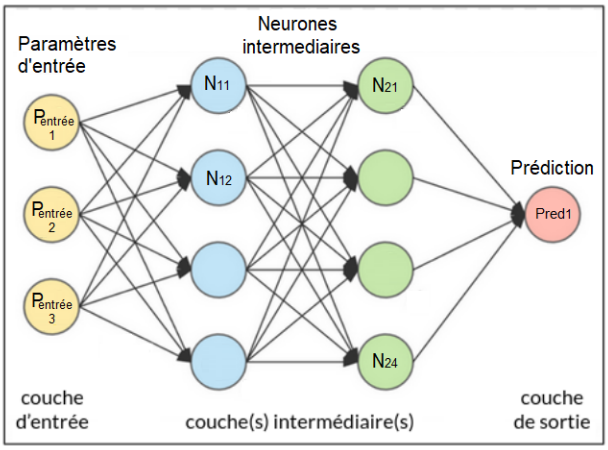

figure 2 : Structure d'un réseau de neurone artificiel

La couche d'entrée contient les paramètres nécessaires à la prédiction. Chaque neurone est le résultat d'une fonction d'activation prenant en antécédent la somme d'un réel (nommé biais) et d'une combinaison linéaire des neurones de la couche précédente. La fonction d'activation est la même pour tout les neurones d'une même couche, seul la combinaison linéaire et le biais changent. Cette méthode est itérée jusqu'à la couche de sortie contenant autant de neurones que l'on a de paramètres à prédire.

On entraîne notre réseau avec des données pré-calculées dont on connaît le résultat attendu en fonction des paramètres d'entrée. Cet entraînement est un problème d'optimisation qui consiste à minimiser l'erreur entre la sortie effective du réseau de neurone et le résultat pré-calculé en jouant sur les poids des combinaison linéaires de notre système et les biais. Dans le cadre du stage, il faudra modifier la structure d'un réseau de neurone existant prédisant seulement la vitesse. Pour déterminer comment modifier la structure final de notre réseau de neurone, il faut savoir quels sont les paramètres nécessaires à la résolution par la méthode multi-échelle.

Deuxième partie

Structure de notre réseau de neurones

1 Paramètres de notre réseau de neurones

Dans notre cas, le réseau de neurones servira notre résolution multi-échelle en prédisant la solution des problèmes sur l'échelle la plus fine. Pour construire un réseau de neurones adapté, il faut déterminer les paramètres nécessaires à la résolution de nos systèmes et ceux qui seront utiles aux résolutions ultérieurs.

Les paramètres nécessaires à la résolution de ce problème sont :

1. $v^M(x,t)$ et $\nabla_x v^M(x,t)$ la solution et son gradient calculés sur le point d'intégration associé de l'échelle la plus grossière et servant à la conservativité du système entre les deux échelles. Ces paramètres sont de dimension respective d et d^2 (6 au total pour nous en 2D).
2. dt un réel correspondant au pas de temps entre chaque itération.
3. $v^m(:,t-dt)$;... ;$v^m(:,t-o\times dt)$ les solutions du problème aux temps précédents sur la plus petite échelle pour le calcul de $\frac{\partial v^m}{\partial t(y,t)}$. Le nombre de solutions dépendra du choix de l'ordre o de la dérivée en temps. Cela augmentera le nombre de paramètres de $o \times N_n \times d$, avec N_n le nombre de noeuds sur le plus petit domaine.

Nous aurons donc une couche d'entrée avec le nombre de neurones suivant.

$$N_e = o \times N_n \times d + d + d^2 + 1 \tag{7}$$

Les paramètres nécessaires aux résolutions ultérieures des problèmes sur les différentes échelles sont :

1. $v^m(:,t)$ pour la même raison et avec la même dimension que vu précédemment.

2. P^m la puissance, un scalaire nécessaire pour la résolution du problème à plus grande échelle. Cette puissance nous donnera $f^M(x,t)$ et $\sigma^{M,dev}(x,t)$ puisque (d'après (5)) nous pouvons les obtenir via la différentiation de la puissance virtuelle en fonction $v^M(x,t)$ et $\nabla_x v^M(x,t)$. De plus, il nous faudra aussi les dérivées seconde de v_p pour l'utilisation de l'algorithme de Newton-Raphson [1]

Avec ces paramètres, on aura donc une couche de sortie avec le nombre de neurones suivant.

$$N_s = N_n \times d + 1 \tag{8}$$

Pour le reste du réseau de neurones il est plus compliqué de choisir le nombre de couches/neurones. Il est communément admis dans la littérature qu'un réseau avec deux couches intermédiaires peut prédire la grande majorité des problèmes sans faire exploser sa complexité qui engendrerait un temps d'entraînement beaucoup plus long. Pour ce qui est du nombre de neurones par couche, un choix trop élevé augmenterait la complexité du réseau plus que nécessaire, mais un choix trop bas nous donnerait un réseau trop peu précis. On peut tout de même déterminer un nombre de neurones sur les couches intermédiaires à ne pas dépasser (Khosravi et al, [3]) en fonction de la quantité de données disponibles pour l'entraînement de notre réseau.
La minimisation du nombre de neurones est primordial afin de réduire le temps de calcul. Si on ne peut réduire directement autant qu'on le souhaite le nombre de neurones dans les couches intermédiaires sans dégrader notre résultat, il existe d'autres moyens de le faire.

2 Utilisation d'un auto-encodeur

Un auto-encodeur est un réseau de neurone qui essaye de reproduire ses paramètres d'entrée en sortie après être passé dans des couches intermédiaires possédant moins de neurones que les couches aux extrémités. Cela nous permet d'encoder (respectivement décoder) des données en isolant les opérations effectuées avant (respectivement après) la couche intermédiaire. Dans notre cas il est intéressant d'en utiliser un puisque notre réseau de neurones peut potentiellement, si le maillage sur la plus petite échelle est très fin, prendre un très grand nombre de paramètres d'entrée.

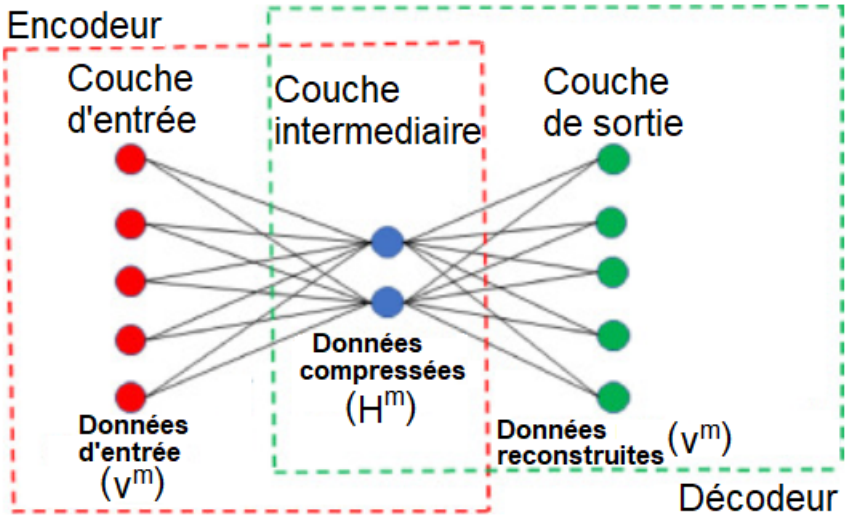

figure 3 : Schéma représentant la structure d'un auto-encodeur

En encodant v^m au temps initial et en entraînant notre réseau de neurone avec les solutions encodées (notées H^m) plutôt qu'avec les valeurs en tous points des solutions précédentes, nous réduirons drastiquement le nombre de neurones présents notre réseau. Pour cela, nous utiliserons un auto-encodeur implémenté avant mon arrivée.

Nous ferons donc nos prédictions avec les données compressées à chaque pas de temps puis décompresserons la solution au temps final. Notre algorithme aura la structure suivante :

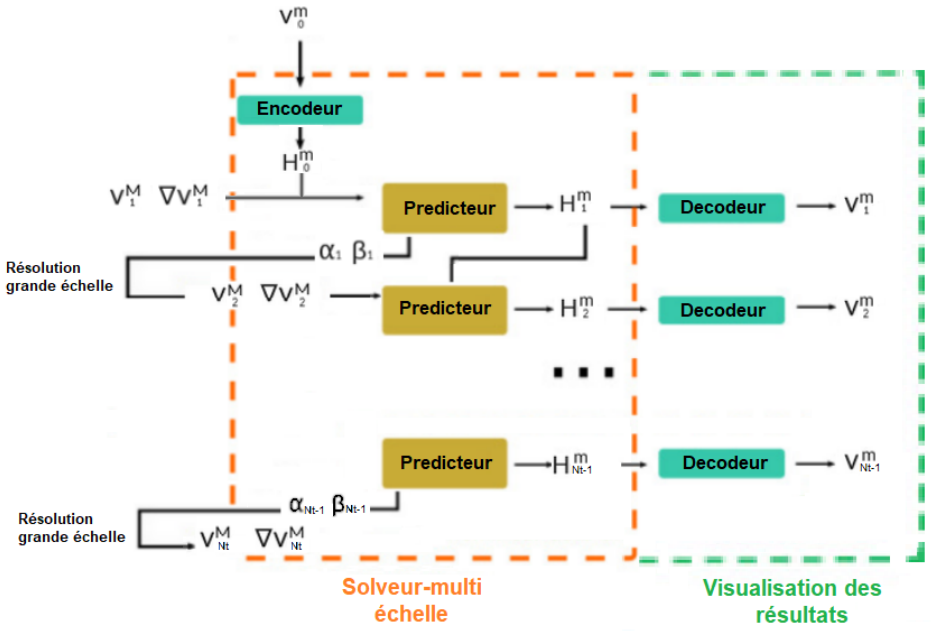

figure 4 : Schéma représentant la structure de l'algorithme

Nous allons donc, pour chaque prédiction avec le réseau de neurone, récupérer les paramètres V^M et ∇V^M du point d'intégration associé sur la grande échelle ainsi que les solutions encodées H^M prédites précédemment puis prédire α, β et le nouveau champs de vitesse encodé. La décompression des solutions encodées n'est pas nécessaire à chaque pas de temps puisque le réseau sera entraîner à prédire en recevant en entrée les solutions encodées. Toutefois, l'utilisation du décodeur reste obligatoire pour la visualisation des résultats. Il faudra donc s'assurer que la compression de la solution au temps initiale n'engendre pas une trop grande erreur qui risquerait de se répercuter sur la qualité de prédiction de notre réseau.
Une fois la structure et les paramètres du réseau de neurones de prédiction et de l'auto-encodeur fixés, on les construit dans un notebook Jupyter à l'aide de la bibliothèque TensorFlow de Google, implémentée en Python. Le code élément finis qui effectue la résolution du problème sur la plus grande échelle est en C. Il faut donc trouver un moyen de faire communiquer ces deux codes afin de pouvoir résoudre notre problème multi-échelle.

Troisième partie

Intégration du réseau de neurones dans le solveur multi-échelle

1 Communication des résultats des différentes échelles

Il existe plusieurs manières de faire communiquer un code Python et un code C. Dans notre cas, l'utilisation d'un réseau de neurones de TensorFlow nous a premièrement amené à considérer l'utilisation de l'API TensorFlow pour le C. Son utilisation aurait permis de charger le réseau de neurones sauvegardé depuis Python directement dans le code C pour effectuer les prédictions nécessaires à la résolution du problème multi-échelle. Malheureusement, le manque de documentation nous à rapidement fait changer notre choix.

Notre deuxième choix à été l'utilisation de l'API Python/C. Puisque nous ne pouvions pas importer le réseau de neurones en C, nous avons pensé à lancer la prédiction dans un script Python depuis le C et de récupérer les variables via un module commun. Cette méthode fut plus concluante que la première puisque nous avons réussi à effectuer des prédictions. Cependant, c'est le debuggage qui pose problème avec cette API. Il est difficile de suivre la gestion de la mémoire avec Valgrind puisque les transitions Python/C sont inaccessibles.

Il existe un autre outil développé par TensorFlow, il s'agit de l'outil de service TensorFlow. Il permet d'utiliser un réseau de neurone enregistré sans l'importer en C. cet outil est un serveur qu'on peut exécuter sur une autre machine que celle effectuant le calcul éléments finis. Le code éléments finis va ensuite interroger le serveur et se voir renvoyer les prédictions, et ce à travers une interface CURL. Cette dernière est accessible en C avec la librairie libcurl, et ce avec très peu d'effort de programmation. C'est ce troisième choix que nous avons décidé de garder. Maintenant que nous avons

un réseau de neurones et un moyen de communiquer son résultat à notre solveur éléments finis de la grande échelle. Il faut nous assurer que notre réseau ayant été entraîné sur des cas potentiellement très différents des cas qui vont être rencontrés lors des prochaines simulations puisse prédire correctement les solutions.

2 Différence cas d'entraînement/cas concret

Lors de l'entraînement de notre réseau de neurones, nous avions des cas où toutes les entrées ainsi que les sorties étaient normalisées pour une utilisation optimale de notre réseau. Il faudra donc faire en sorte que pour les autres cas que ceux d'entraînement une normalisation soit faite. De plus, dans l'objectif de réduire la taille de notre réseau, nous n'avons pas sorti $f^M(x,t)$ et $\sigma^{M,dev}(x,t)$ mais la puissance $P^M(x,t)$. Or, comme vu précédemment, ce n'est pas la puissance qui nous intéresse pour les résolutions mais ses différentiations. Pour pouvoir faire la distinction entre les cas d'entraînement et les cas que notre réseaux va rencontrer une fois enregistré, nous avons un booléen qui, en fonction du cadre dans lequel le réseau de neurone est utilisé, déterminera s'il y aura une normalisation des entrées, une dénormalisation des sorties et des différentiations sur la puissance prédite. Ces différentiations seront effectuées avec les fonctions fournies par la bibliothèque TensorFlow.

Même en prenant en compte ces différences entre les prédictions d'entraînement et les prédictions effectuées dans le cadre de la résolution multi-échelle. Nous ne pouvons pas être certains que tous les cas qui vont être traités par notre réseau de neurones auront une solution prédite avec une erreur d'approximation raisonnable. Il faut anticiper le fait que le réseau risque d'être confronté à des cas dont il n'a jamais été entraîné à prédire convenablement la solution. Nous allons nous pencher sur les façons d'améliorer ce réseau de neurones afin d'éviter ces prédictions hasardeuses qui se répercuterons sur le résultat de la simulation.

Quatrième partie

Réseau de neurones actif

Mesurer l'incertitude d'un réseau de neurone revient à quantifier la confiance du réseau envers sa prédiction. Cette incertitude peut être issue de plusieurs facteurs et peut donc être séparée en deux groupes :

- L'incertitude aléatoire liée à la variabilité naturelle des données. En effet, si les paramètres d'entrée sont issus de mesures, il se peut qu'une précision trop faible ou un bruitage dans le recueil des données puisse affecter la prédiction.

- L'incertitude épistémique liée à la variabilité engendrée par le réseau. Le choix des paramètres qui vont être entrés dans le réseau ainsi que le choix de la structure (nombre de couches, choix de la fonction d'activation) peuvent aussi impacter la prédiction.

C'est cette deuxième incertitude que l'on va chercher à caractériser.
Pour ce faire, nous allons générer de la variance dans nos prédictions. Nous ferons ensuite plusieurs tirages consécutifs pour de même données d'entrée et nous analyserons les différents résultats prédis. Une variance faible dans les prédictions indiquera une forte confiance du réseau sur la qualité du résultat. A l'inverse, une forte variance indiquera une incertitude du réseau sur la prédiction de ce cas en particulier. Pour générer cette variance nous allons introduire des dropouts dans notre réseau. Un dropout permet, avec une probabilité p définie au préalable, d'annuler un neurone.

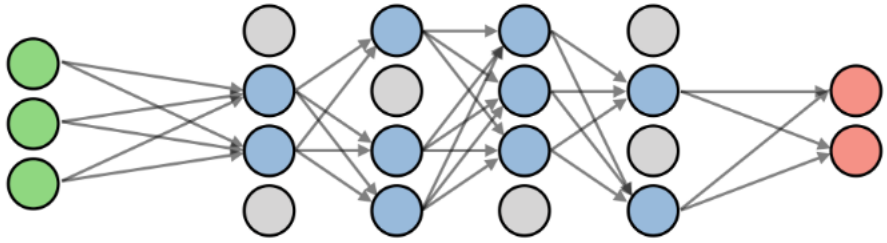

figure 5 : Schéma représentant l'utilisation de dropout.

Cette méthode introduit de la variance par le fait qu'à chaque itération d'une prédiction, les neurones ne seront pas issus d'une combinaison linéaire des mêmes neurones de la couche précédentes. En utilisant cette technique au cours de l'entraînement, cela permet aussi d'avoir une meilleure répartition des poids entre les neurones et donc une assurance que tout les paramètres sur le couche initiale auront un impact sur la prédiction. L'utilisation de dropouts nous intéresse surtout pour sa première propriété puisque nous allons pouvoir étudier la variance des résultats après plusieurs prédictions d'un même cas, nous aurons alors un aperçu de la confiance du réseau sur la prédiction. Une fois que nous aurons déterminé des cas avec une grande incertitude de prédiction, nous pourrons améliorer notre réseau en ajoutant ces cas aux données d'entraînement. Cette méthode permettra d'effectuer un entraînement plus efficace avec un minimum de cas et donc de gagner du temps à la fois sur la génération de données et sur le temps d'entraînement.

Cinquième partie

Résultats

Nous allons nous concentrer sur un maillage de la petite échelle en 2D composé de 3172 points d'intégrations. Notre auto-encodeur sera entraîné à la compression/décompression du champs de vitesse sur ce maillage avec une couche intermédiaire composée de seulement 8 neurones. On choisi d'effectuer une approximation d'ordre 2 pour la dérivée en temps. Notre réseau de neurone de prédiction prendra donc 23 paramètres d'entrée pour une prédiction de 9 paramètres de sortie (7,8) Nous aurons accès à un jeu de données composée de 1024 problèmes pré-calculés avec un solveur éléments-finis sur 101 pas de temps de 0.1 seconde. Cela représente $1024 \times 101 = 103424$ prédictions pour notre réseau qui fonctionne pour un seul pas de temps à la fois.

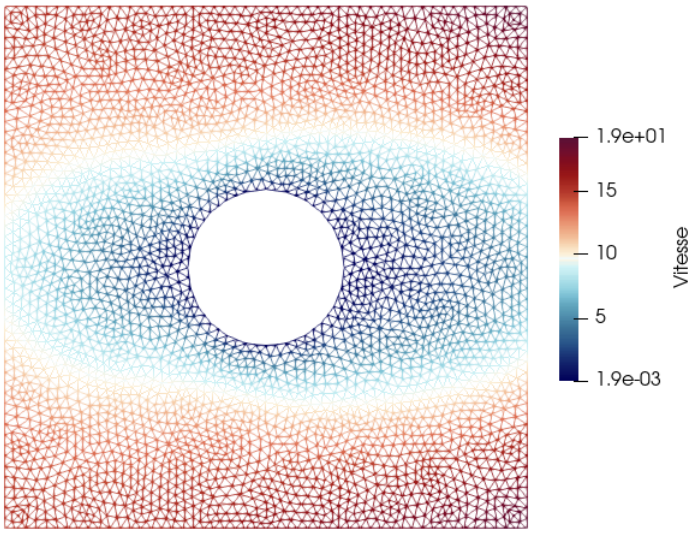

figure 6 : Exemple de champs de vitesse pré-calculée par un solveur éléments-finis sur notre maillage

On répartis en trois groupes distincts avec les répartitions et objectifs suivants :

1. 60% des cas sont dédiés à l'entraînement des réseaux de neurones.

2. 20% des cas sont dédiés à la validation de l'entraînement des réseaux de neurones.
3. 20% des cas sont dédiés aux test post-entraînement.

Les poids des neurones et les biais des réseaux sont donc optimisés sur le jeu d'entraînement jusqu'à avoir des résultats acceptables sur le jeu de validation. Ce qui est acceptable est définit avant l'entraînement. Dans notre cas, on arrête l'entraînement lorsque la prédiction du jeu de validation ne s'est pas améliorée pendant 30 itérations d'entraînement successives. Nous observons ensuite le comportement des réseaux sur le jeu test n'ayant jamais servi lors de l'entraînement ou de la validation.

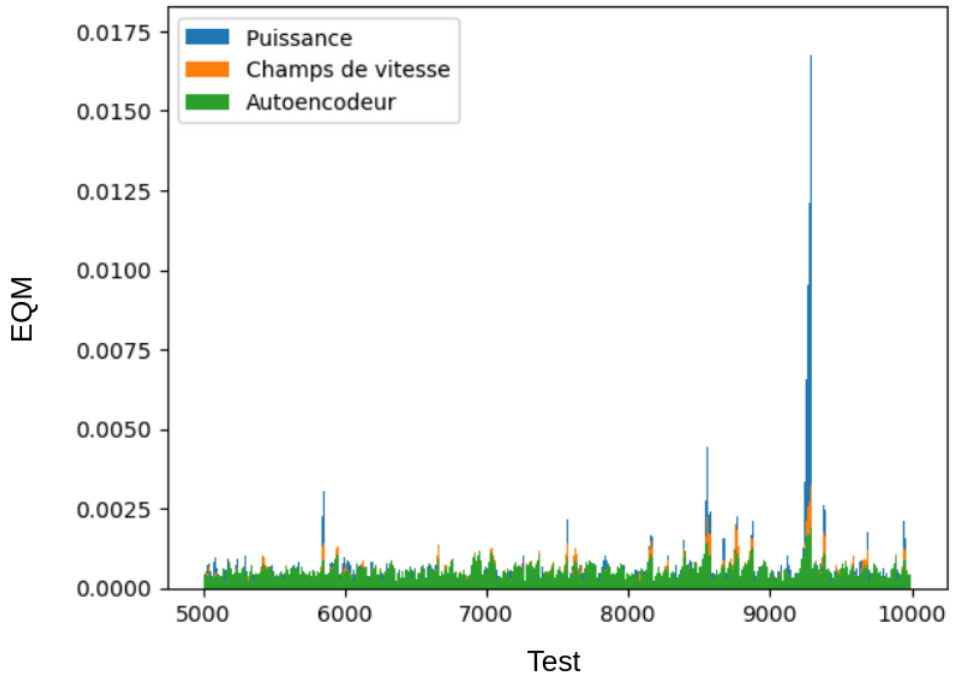

figure 6 : Erreurs quadratiques moyennes (EQM) relatives sur tout les cas test entre :

1. la puissance prédite et celle obtenue par la solution éléments-finis,
2. le champs de vitesse prédis et celui calculé par éléments-finis,
3. le champs de vitesse calculé par éléments-finis et la sortie après sa compression/décompression.

L'erreur quadratique moyenne relative est calculée avec les solutions dénormalisées comme ci-dessous :

$$\text{i}^{eme}\ \text{erreur} = \sqrt{\frac{\sum_{x=0}^{3172}(v^m_{i,entree} - v^m_{i,sortie})^2}{\sum_{j=0}^{N}\sum_{x=0}^{3172}(v^m_{j,entree})^2}}$$

On remarque que pour certains cas, l'erreur explose. Cela correspond à une génération automatique de paramètres d'entrée qui a pris des valeurs éloignées de celles utilisées pour l'entraînement de notre réseau.
En effet, la génération des cas étant aléatoires, il se peut que certains cas soient vraiment singuliers. Pour vérifier cela, on regarde s'il existe des cas avec une puissance supérieure à la puissance moyenne + 10 fois la variance. Pour nos cas, la puissance moyenne est de

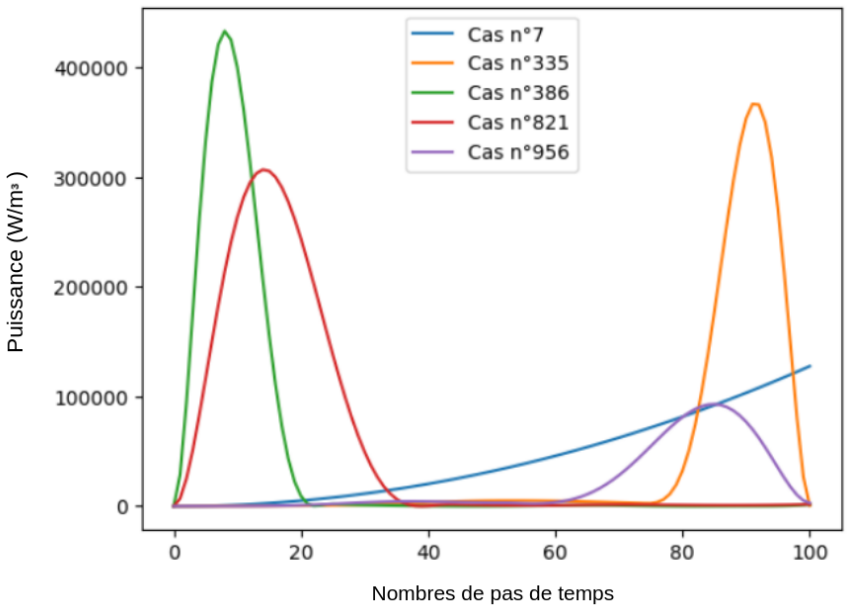

figure 7 : Cas avec une grande puissance à prédire.

On compare ces cas avec les prédictions effectuées par le réseau de neurone.

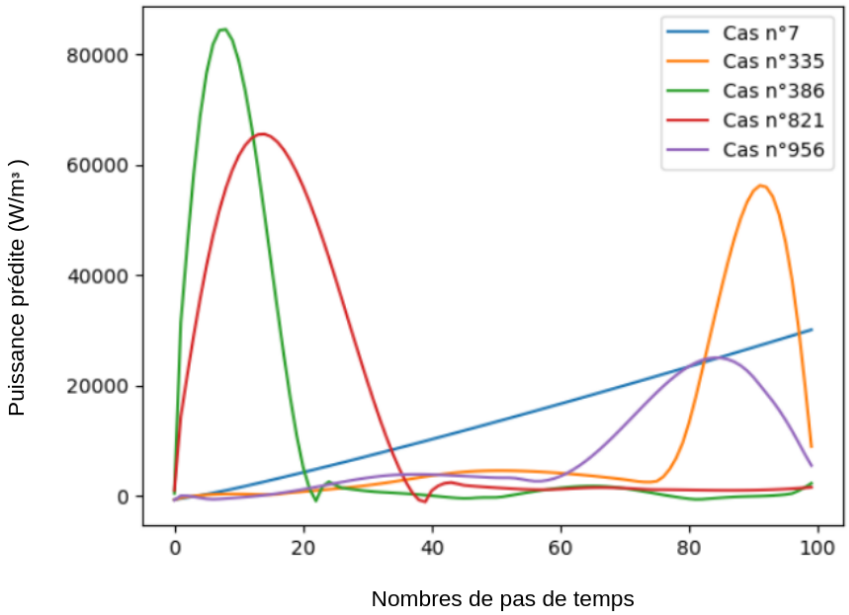

figure 8 : Prédiction des cas avec une grande puissance.

Même si les prédictions ont la bonne forme, les puissances prédites sont environ 5 fois trop petites. La majorité de ces cas étant déjà compris dans les cas d'entraînements, il faut trouver un moyen de ne pas prendre en compte ces cas particuliers. Afin d'avoir une utilisation optimale de notre réseau de neurone, on va mesurer la confiance de notre réseau vis-à-vis de sa prédiction afin de ne pas l'utiliser dans des cas peu ou non entraînés. Pour ce faire, on calcule la variance des résultats après l'utilisation de dropouts. On détermine premièrement le nombre d'itérations nécessaires pour avoir une valeur représentative de la variance d'un résultat prédit.

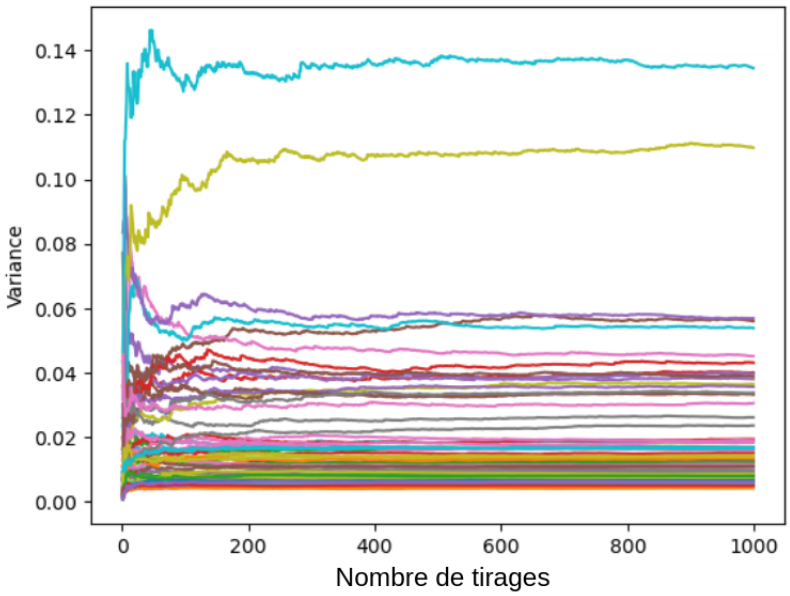

figure 9 : Graphes représentant l'évolution de la variance des résultats normalisés en fonction du nombre d'itérations

Nous pouvons remarquer que la variance des résultats se stabilise aux alentours de 200 itérations. Toutefois, nous allons fixer le nombre d'itérations à 100 pour effectuer pour le calcul de variance puisque cela suffit à déterminer quelles prédictions engendrent de grandes variances et lesquelles ne posent pas de problèmes d'incertitudes trop élevé. Maintenant que nous savons combien d'itérations faire pour obtenir la variance d'une prédiction et donc quantifier la certitude du réseau sur cette dernière, nous pouvons vérifier la corrélation entre la variance et la précision des résultats pour des cas tests générés de la même façon que les cas utilisés pour l'entraînement.

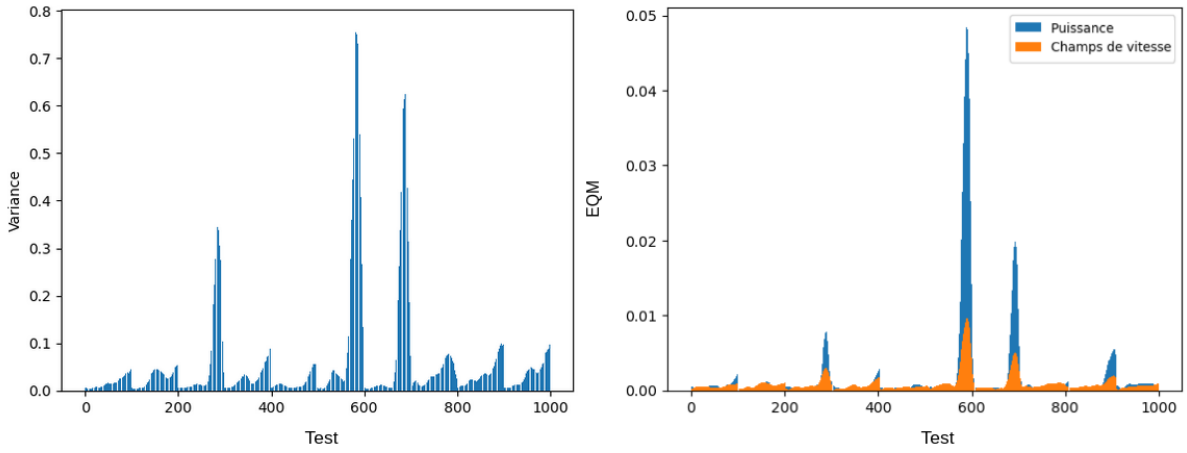

figure 10 : Comparaison entre variance et erreur de prédiction

La figure 10 montre bien la corrélation entre l'incertitude de résultat, que l'on peut obtenir sans avoir les données calculées au préalable par un solveur éléments finis, et l'erreur d'approximation. Il semble donc possible de détecter les cas que le réseau prédit avec trop peu de précision. Tous les résultats précédents on été obtenus sur un premier jeu de données généré avec des conditions initiales choisies aléatoirement. Nous allons désormais appliquer le réseau à un cas concret résolu par le solveur élément finis.

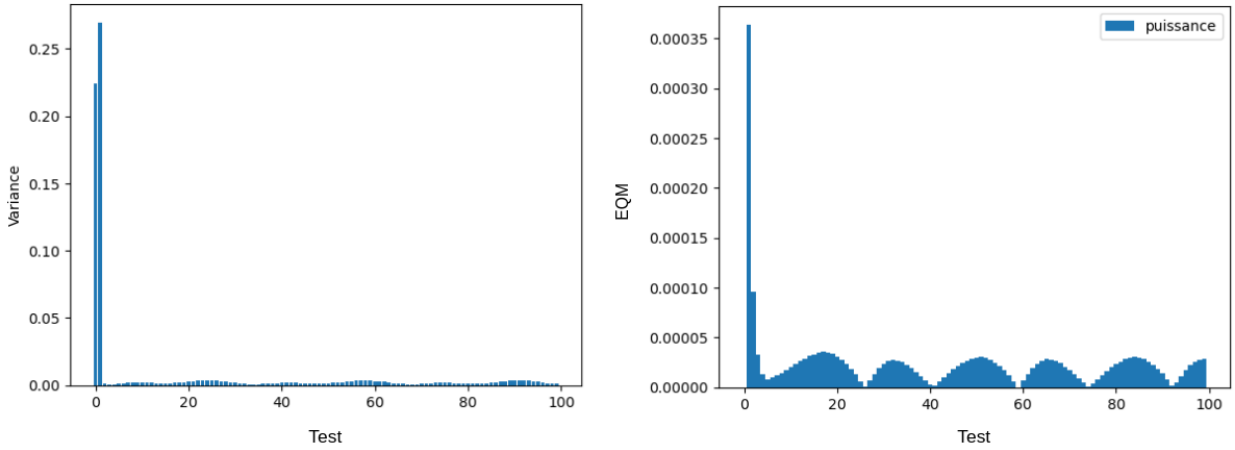

figure 11 : Comparaison entre variance et erreur de prédiction de la puissance sur une deuxième jeu de données

Nous pouvons remarquer que l'erreur de prédiction de la puissance est plus faible pour ce second jeu de données. Cela peut être expliqué par le fait que la puissance prend des valeurs moins grandes dans ce cas concret que dans les cas précédents générés aléatoirement. Nous pouvons aussi remarquer que l'incertitude semble encore être corrélée à l'erreur de prédiction ce qui est bon signe pour une utilisation du réseau sur des cas variés.

Sixième partie

Conclusion

Durant ce stage, nous sommes partis d'un autoencoeur et d'un réseau de neurone ne prédisant que la vitesse. Nous avons ajouté la prédiction de la puissance et sa différentiation, la communication avec le code C et l'estimation des incertitudes. Il reste tout de même des d'éléments à perfectionner avant de pouvoir avoir un solveur multi-échelle utilisable. Notamment la capacité de prédire des cas plus variés en améliorant l'apprentissage actif. En effet, la méthode de l'incertitude n'est pas miraculeuse car des cas fortement similaires engendrant une grande incertitude risquent d'être ajoutés plusieurs fois aux données d'entraînement. Il faudra donc trouver un moyen de pallier cela (l'utilisation du partitionnement des données peut être envisagé).

J'ai personnellement beaucoup appris au cours de ce stage avec de nouvelles notions notamment sur les réseaux de neurones, leurs fonctionnements et implémentations, ainsi qu'un nouvel environnement de travail dans un bureau de recherche très différent des autres cadres de travail que j'ai pu connaître auparavant. J'aimerai pour la suite me faire une expérience du travail dans le privé afin de pouvoir faire un choix de carrière professionnel plus éclairé.

Je remercie M. Modesar Shakoor de m'avoir donné l'opportunité d'effectuer ce stage et d'acquérir cette expérience, ainsi que les personne m'ayant accompagnés durant ces deux années de master, aussi bien enseignants que collègues de promotion ou de stage.

Références

[1] Modesar Shakoor and Chung Hae Park, Computational homogenization of unsteady flows with obstacles, 2022

[2] Dmytro Vasiukov, José Mennesson, Krushna Shinde, Modesar Shakoor and Vincent Itier , Dimensionality reduction through convolutional autoencoders for fracture patterns prediction, 2023

[3] Abbas Khosravi, Saeid Nahavandi and Doug Creighton,A prediction interval-based approach to determine optimal structures of neural network metamodels, 2010

[4] Yarin Gal,Uncertainty in Deep Learning, 2016

[5] Ozan Sener and Silvio Savarese,Active learning for convolutional neural network : a core-set approach, 2018

[6] Lior Rokach and Oded Maimon, Clustering Methods, 2005

[7] P. J. Blanco, P. J. S´anchez, E. A. de Souza Neto, and R. A. Feijoo, Variational Foundations and Generalized Unified Theory of RVE-Based Multiscale Models, 2016.