Illumination indirecte en temps réel avec des modèles d'apprentissage

Emmanuel Bengio

Il existe de nombreuses approches temps-réel pour simuler l'illumination globale dans une scène avec des sources de lumière dynamiques. Nous explorons ici une technique qui précalcule une fonction d'illumation indirecte, sous forme d'un modèle qui apprends une fonction non-linéaire. On utilise ici la capacité des modèles d'apprentissage à tirer de la cohésion dans des sous-espaces de l'espace d'entrée pour modéliser une fonction de radiance.

1 Introduction

L'illumination globale semble un phénomène intéressant à pouvoir reproduire en temps réel, puisque l'illumination directe est facile à calculer, mais reproduit avec peu de fidélité la réalité. L'illumination indirecte permet grandement d'augmenter la fidélité de l'image, puisqu'elle joue généralement un rôle assez important à la radiance d'un point donné.

L'illumination indirecte en un point est difficile à calculer puisque qu'elle doit prendre en compte l'intégrale des rayons incidents qui l'affectent, en plus de prendre en compte les effets de rebonds et d'occlusion, le tout affecté par des BRDF qui peuvent être complexes.

Il existe différentes techniques en temps-réel qui ont des degrés de réalisme plus ou moins élevés: l'utilisation intelligente de points de lumière virtuels [6], l'utilisation d'échantillons épars pour approximer les rebonds de rayon sur GPU [7], ou encore l'utilisation de *precomputed radiance transfer* [9].

Malheureusement ces méthodes ont quelques désavantages. En particulier, il est parfois difficile de calculer les effets hautes-fréquences, commes les caustiques. Un autre problème est la taille des données nécessaires pour les solutions de précalcul.

Ce travail essaie partiellement de reproduire et d'améliorer des résultats obtenus à partir de réseaux de neurones artificiels entraînés pour précalculer une fonction de régression de radiance (*radiance regression function*) [8].

Un des avantages d'utiliser cette approche est que ces modèles sont capables, s'ils sont bien entraînés, de capturer la structure sous-jacente de leur espace d'entrée. En fait ce sont des approximateurs universels [2, 3], c'est à dire qu'il peuvent potentiellement modéliser n'importe quelle fonction continue dans l'hypercube.

La structure des réseaux de neurones leur permet d'approximer une fonction dont les sous-espaces d'entrées sont discontinus et dont la sortie est discontinue par endroits (tant qu'il s'agit d'un nombre fini de "morceaux" continus). Cela rends les réseaux de neurones adaptés à un problème directement lié à la géométrie et l'illumination, puisqu'ils peuvent par exemple gérer la discontinuité de la radiance entre deux objets très proches l'un de l'autre.

Il existe d'autres méthodes d'apprentissage plus ou moins reliées à cette dernière qui sont testées ici, et qui sont tout aussi robustes aux discontinuités, comme les arbres de décisions et les mixtures d'experts [1]. En fait ces méthodes pourraient permettre d'aller plus loin dans la réduction de taille des données à précalculer (du modèle) autant que dans le temps d'entraînement nécessaire, la précision du modèle étant presque entièrement limitée par la quantité de sous-modèles qu'on est prêt à entraîner et stocker.

2 Travaux reliés

Méthodes interactives

La méthode de points de lumière virtuels (VPL) [5] consiste à tirer des rayons depuis les sources de lumière de

manière uniforme, mais éparse, dans la scène, et à la première intersection de ces rayons créer des lumières (*point lights*) virtuelles. On peut améliorer cette technique pour des lumières dynamiques en éliminant partiellement les VPL qui ne sont plus directement visibles depuis leur lumière d'origine [6], et relancer les rayons quand on a en trop éliminé. À partir des VPL on peut créer des shadow maps plus complexes qui simulent relativement bien l'illumination globale à un rebond, ce qui est efficace mais peut difficilement modéliser les effets hautes-fréquences.

Méthodes de précalcul

La méthode de *precomputed radiance transfer* [9] réussit à précalculer pour chaque objet à sa surface une fonction de transfert de radiance, encodée en harmoniques sphériques. Cette fonction décrit la relation de l'objet avec l'environnement d'un point de vue du shading, ce qui fait que la radiance incidente n'a pas besoin d'être précalculée, vu qu'elle peut l'être en temps réel par un GPU.

Au *run-time*, la radiance incidente est passée à travers la fonction de transfert, qui est convoluée avec la BRDF de la surface en question et évaluée en fonction de la direction de vue actuelle pour produire une couleur.

Bien que cette méthode soit relativement efficace, elle requiert beaucoup de stockage et l'utilisation de lumières dynamiques la rend plus susceptible à ne pas pouvoir représenter les effets hautes-fréquences.

Méthodes de régression

On s'intéresse en particulier à [8] puisque c'est ce qu'on aimerait reproduire avec certaines variations. Cette méthode utilise des réseaux de neurones profonds (avec deux couches cachées) mais de petite taille (20×10 neurones cachées, avec une entrée de dimension 15 et une sortie RGB, soit 530 paramètres à apprendre) pour apprendre la radiance en un point.

Plus précisément, elle utilise des réseaux de neurones pour apprendre une fonction globale d'illumination indirecte qui dépend de la position à la surface, de la direction de vue et de la position d'une lumière. On entraine un réseau de neurones pour une sous-section de la scène donnée, ce qui permet de pouvoir règler le niveau de précision voulu (mais bien sur ça prends plus de réseaux).

On peut utiliser n'importe quelle méthode de calcul d'illumination globale pour entraîner les réseaux de neurones, dans ce cas-ci on a utilisé un path tracer avec un certain nombre de rayons lancés pour chaque échantillon d'illumination indirecte.

3 Illumination indirecte

Pour tout point $\mathbf{x}$ sur une surface dans la scène on veut être capable de calculer l'illumination globale. Cette illumination dépend de la position de la lumière $\mathbf{l}$ (on considère le cas à une lumière, mais c'est facilement généralisable pour n lumières, puisqu'on peut prendre pour aquis que les effets de chaque lumière sont indépendants) et de la direction $\mathbf{v}$ depuis laquelle on voit le point.

L'illumination globale peut être séparée en un terme d'illumination directe et un terme d'illumination indirecte. L'illumination directe avec un nombre raisonnable de lumières est approximable en temps réel avec des techniques de rasterisation et de shadow mapping. Par contre l'illumination indirecte dépend de termes qui sont typiquement intractables (l'intégrale des rayons incidents) mais qu'on peut approximer avec un degré de fiabilité acceptable avec plus d'une technique, seulement pas en temps réel.

L'illumination indirecte est donc directement dépendante de trois termes, $\mathbf{x}, \mathbf{v}$ et $\mathbf{l}$. On voudrait donc découvrir une fonction r_i, facile à évaluer, qui à partir de ces termes calcule l'illumination indirecte. Pour expliciter certains paramètres et rendre plus facile et direct le calcul de l'illumination il peut être utile de donner plus de paramètres explicitement, notamment la normale $\mathbf{n}$ de la surface au point $\mathbf{x}$, les paramètres numériques $\mathbf{a}$ de la BRDF de la surface.

3.1 Collection des données

Pour la collection des données d'illumination indirecte on va utiliser un algorithme de path tracer. Comme cet algorithme est relativement bruité, surtout si on utilise un seul *point-light*, on utilise une variante qui à chaque intersection rayon-scène prend en compte l'illumination directe du point avec un certain facteur.

3.1.1 Mitsuba

Les données sont présentement extraites depuis un plugin intégrateur de Mitsuba[4], qui est un path tracer modifié.

Pour chaque configuration de la scène (on change la position de la lumière et du point de vue) et pour chaque rayon qui est lancé depuis la caméra, on garde la position de l'intersection, la direction vers la lumière, la normale à la surface, ainsi que la direction vers la vue, en laissant tomber l'illumination directe. Puis on laisse le pathtracer explorer un assez grand nombre (80 ici pour limiter le bruit) de chemins à partir de ce point et ainsi récupérer la valeur moyenne de l'illumination indirecte. Cette valeur devient la cible, et les autres valeurs gardées l'entrée d'un exemple.

Le problème de cette méthode est qu'on collectionne des points dans l'espace qui sont distribués selon ce qu'on échantillone depuis des points de vue choisis, ce qui ne représente pas une distribution uniforme de la scène, qui serait idéale pour l'apprentissage. Il aurait été plus intéressant d'échantillonner des points de manière uniforme, ou encore mieux, de manière à capturer plus de points représentant des hautes fréquences.

En effet, on peut voir aisément que si l'on prends un nombre trop faible configurations et que celles-ci capturent pas assez un aspect de la scène, alors le résultat visuel en sera très affecté. Il est arrivé dans nos test que, par hasard, un objet d'une scène ne soit échantilloné qu'une seule fois, ce qui a eu pour effet de le faire avoir une couleur absurde.

4 Méthodes d'apprentissage

Comme ce problème est un problème où l'on peut générer un nombre arbitrairement grand d'exemples d'apprentissage et où ces données ont une certaine cohérence, il semble intéressant de s'y attaquer avec de telles méthodes. On a donc essayé par diverses méthodes d'apprendre une fonction de régression r_i^*qui approxime l'illumination indirecte.

Cette fonction peut être apprise pour une scène particulière, ou bien on pourrait vouloir essayer de généraliser encore plus et d'apprendre une fonction qui dépend aussi de la géométrie (ou indirectement d'une représentation intermédiaire de celle-ci) et non qui est attachée à une scène en particulier.

4.1 Arbres de décision

Les arbres de décision sont une méthode de classification ou de régression non-paramétrique simple et relativement efficace. Il s'agit d'apprendre par des exemples un arbre binaire de décision où chaque décision dépend d'un paramètre d'entrée (potentiellement plus d'une fois) et d'un test appris.

Un grand avantage de cette méthode est que malgré sa taille, l'arbre s'évalue environ en $\log n$, où n est le nombre de points d'apprentissage, pour n'importe quel exemple. Comme l'arbre est capable de produire des réponses à plus d'une dimension et qu'il est relativement facile de l'apprendre, c'est une solution à considérer pour ce problème. De plus, les arbres de décision par leur nature peuvent potentiellement apprendre des fonctions assez hautes fréquences.

Le désavantage majeur de cette méthode est qu'elle tend à mal généraliser, donc à ne pas prédire les bonnes choses pour des exemples qu'elle n'a jamais vus. De plus, la fonction qu'elle apprend est essentiellement une "step-function" qui dépend des décisions, ce qu'on voit s'avèrer être néfaste pour une application graphique.

4.2 Réseaux de neurones

Les réseaux de neurones artificiels *feedforward* sont une méthode très intéressante d'apprentissage machine. On appelle *feedforward* un réseau où l'information ne se propage que dans un sens, de l'entrée vers la sortie. Les réseaux de neurones sont généralement divisés en couches de neurones où les neurones dans une couche sont reliées à toutes les autres neurones de la couche suivante. On peut voir un réseau de neurone à une seule couche comme une matrice de poids W et un vecteur de biais b: chaque dimension x_i de l'entrée x envoie un signal à toutes les neurones h_j, signal qui est la valeur de x_i pondérée par un poids w_{ij}. Chaque neurone $h_j = \sum_i w_{ij}x_i$ est ensuite "activée" par une fonction dite d'activation f, généralement à allure sigmoidale (comme la tangente hyperbolique tanh ou la fonction logistique $1/(1+e^{-x})$). Chaque activation est centrée autour d'un biais b_j.

On a donc que la sortie du réseau à une couche est:

$$h(x) = f(xW + b)$$

Si l'entrée x est un vecteur de dimension d et la sortie $h(x)$ de taille m on a une matrice $W^{d\times m}$ et un vecteur

de biais de dimension m. L'idée de l'apprentissage est de trouver les W et b optimaux (et potentiellement les f optimales) à la résolution d'une tâche.

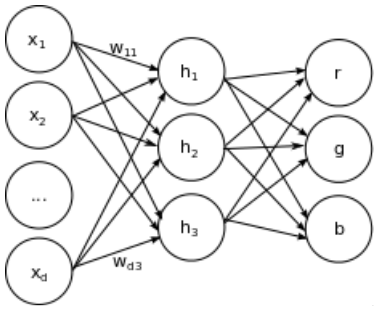

Figure 1, un réseau de neurone à une couche cachée de $m = 3$ neurones.

Grâce aux biais et à aux fonction d'activations, les réseaux de neurones peuvent modéliser des fonctions discontinues par endroit. Si pour un neurone h_j les poids w_{ij} sont très grands, une légère variation autour de $-b_j$ des entrées peut faire passer la fonction d'activation de son minimum à son maximum (de -1 à 1 par exemple pour une tanh). Il est beaucoup plus facile de modéliser des fonctions discontinues avec des réseaux plus profonds car on a l'occasion d'utiliser cette capacité à plusieurs reprises.

Un réseau de neurones à plus d'une couche est dit profond, et les couches intermédiaires sont dites couches cachées. Pour un réseau de neurones *feedforward* il suffit "d'empiler" les couches en considérant l'entrée d'un couche comme la sortie de la précédente. De plus, chaque couche peut avoir sa fonction d'activation particulière. Voici un exemple avec une couche cachée:

$$h(x) = f_2(f_1(xW^1 + b^1)W^2 + b^2)$$

Modèles auto-encodeurs Les modèles de réseaux de neurones auto-encodeurs servent à apprendre une représentation h de leur entrée, et le font en minimisant une fonction de perte où le but est d'être capable de reconstruire l'entrée x dans r à partir de h:

$$\mathcal{L}(x) = |r - x|^2$$

Voici un exemple d'auto-encodeur à une couche qui utilise des poids liés (on utilise les même poids pour encoder et pour reconstruire):

$$r(x) = f_1'\left(f_1(xW + b)W^T + b'\right)$$

L'utilité de tels réseaux est qu'ils sont capables d'obtenir des représentations h qui peuvent être utilisées comme une autre représentation de x, ou d'un vecteur de *features* représentatives de x dans la dimension souhaitée.

Entrainement des réseaux de neurones Il existe une multitude de techniques pour entraîner un réseau de neurones pour divers tâches. La tâche qui nous intéresse ici est la régression, on fait donc de l'apprentissage supervisé, où à partir d'un exemple x (ici les paramètres de l'illumination, $\mathbf{x}, \mathbf{v}, \mathbf{l}, \mathbf{n}$) on veut pouvoir prédire une cible y (une couleur). On va utiliser ici la descente de gradient, qui est la base de beaucoup d'autres techniques d'entraînement.

Soit θ l'ensemble des paramètres du réseau (les W et b), et soit $h(x)$ la sortie du réseau, on définit une fonction de perte $\mathcal{L}(x, y, \theta)$ qui donne une valeur de l'erreur et de la non-régularité du modèle. On utilise dans notre un terme d'erreur relative qui semble mieux capturer les faibles variations sur les couleurs sombres:

$$\mathcal{L}(x, y, \theta) = \sum \frac{|h(x, \theta) - y|}{y + \epsilon}$$

On peut ajouter à cette fonction un coût pour régulariser les paramètres θ; on le fait en général pour éviter l'*overfitting*. L'overfitting est quand un modèle d'apprentissage apprends trop bien (voire par coeur) son ensemble d'apprentissage et qu'il généralise mal sur des exemples qu'il n'a jamais vus. L'*underfitting* est quand un modèle généralise trop, par exemple quand un modèle apprends une fonction linéaire pour approximer une fonction d'ordre 4, la fonction linéaire est trop générale et ne capture pas le détail de la fonction à apprendre. Un exemple de régularisateur simple est la régularisation L_2, la norme euclidienne de θ.

La descente de gradient consiste à calculer la dérivée partielle de $\mathcal{L}$ par rapport chaque paramètre θ_i, et de "descendre" la dérivée par petits incréments, dont on règle la "magnitude" avec μ le taux d'apprentissage. Les paramètres à l'itération $t + 1$ deviennent donc:

$$\theta^{t+1} \leftarrow \theta^t - \mu \frac{\partial \mathcal{L}}{\partial \theta^t}$$

On arrête l'apprentissage quand on considère que $\mathcal{L}$ est assez petit, ne descend plus, ou quand l'erreur sur un ensemble de validation (disjoint de l'ensemble d'entraînement) ne descent plus ou remonte (*early-stopping*). On utilise généralement $\mu < 1$ car on ne veut pas que les poids soient trop changés à chaque itération, car on pourrait aller trop loin et potentiellement faire remonter l'erreur, par contre si on prend un μ trop petit, on risque de rester pris dans des minima locaux de la fonction de perte. Le taux d'apprentissage est donc un hyper-paramètre qu'il faut découvrir par d'autre méthodes.

Les autres hyper-paramètres sont le nombre de neurones de chaque couche cachée, le nombre de couches cachées, ainsi que les fonctions d'activation de chaque couche.

Utilisation Cette méthode permet potentiellement d'approximer une fonction très complexe, on devrait donc être capable de modéliser relativement bien quelque chose comme une BRDF et l'illumination indirecte d'un point dans une scène. De plus un modèle assez profond devrait être capable de modéliser des effets hautes fréquences. Par contre, pour modéliser une scène au complet avec beaucoup d'objets, il va falloir un assez gros modèle, ce qui est problématique sur GPU au niveau de la mémoire mais surtout du temps d'exécution, si on doit évaluer un grand modèle pour chaque pixel en temps réel.

Une solution est de diviser la scène en sous-espaces [8] et d'entrainer un petit modèle pour chaque. Cela permet plusieurs choses: on peut diviser la scène tant qu'on veut pour obtenir la précision voulue, et on peut choisir la taille des modèles pour qu'ils soient assez petits pour être évalués en temps réel.

Une autre solution explorée ici est une variante de mixture d'experts [1], où on entraîne un grand nombre de sous réseaux, et on entraîne en même temps un réseau qu'on appelle le *gater*, qui en fonction de l'entrée choisit quel sous réseau utiliser pour trouver la solution. C'est donc une solution semblable mais où au lieu de diviser la scène de manière géométrique (ce qui est plus ou moins un choix arbitraire), on apprends la division, qui est une division non linéaire dans tout l'espace des entrées (et non juste dans les 3 dimensions de la position $\mathbf{x}$).

5 Résultats

5.1 Modèles

Tous les modèles apprennent une fonction de régression avec comme entrée un vecteur de taille 12 $(\mathbf{x}, \mathbf{v}, \mathbf{l}, \mathbf{n})$ et une sortie RGB.

5.1.1 Arbres de décision

Les arbres de décision se sont avérés être un mauvais choix. Bien qu'efficaces, les arbres de décision ont le désavantage d'apprendre une fonction de régression linéaire par morceaux, ce qui pour une application graphique produit des discontinuités visibles et indésirables.

5.1.2 Réseau de neurones simple

Un réseau de neurones seul peut suffire à modéliser une scène assez simple. Malheureusement, cela nécessite un réseau de grande taille. Pour modéliser la Cornell Box avec une assez grande fiabilité, il a fallu un réseau de neurones à deux couches cachées de 200 et 100 neurones cachés, avec fonctions d'activations tanh et sigmoide en dernière couche (cette fonction borne la sortie entre 0 et 1).

Ce modèle, entrainé avec une simple descente de gradient sur environ 500000 exemples suffire à modéliser la scène avec fidélité (erreur relative de moins de 1%).

Un réseau de beaucoup plus petite taille ($<$20 neurones cachés par couche) peut-être utilisé en temps réel (voir section sur le temps réel) mais rends la scène avec une bien moins bonne fidélité($>$8% d'erreur), et sur une scène plus complexe s'avère assez médiocre.

5.1.3 Division par kd-tree

Pour pouvoir augmenter la précision de la régression sans toutefois perdre beaucoup de vitesse, la solution proposée par [8] est de diviser la scène avec un arbre, et pour chaque sous section de la scène entraîner un sous-réseau unique. L'avantage de cette solution est sa simplicité: plus on veut de précision, plus on divise profondément dans l'arbre, et peu importe la profondeur (jusqu'à un certain point) un petit réseau pourra faire l'affichage en temps réel.

Par contre, comme il s'agit d'une division assez arbitraire des entrées (que ce soit dans les trois dimensions

spatiales ou dans les 12 dimensions de l'entrée), cela force souvent un trop grand nombre de sous réseaux (plus de 6000 sous réseaux pour la Cornell Box seulement). Cela est dû au fait que quand on divise on ne sait pas d'avance sur quel axe ni où la division sera la plus bénéfique, et donc quand on divise on obtient une certaine amélioration qui est assez grande pour être considérée, mais généralement en nécessite quand même d'autres par la suite.

5.1.4 Mélanges d'experts

La solution explorée ici est une nouvelle variante des modèles de mixture d'experts et de calcul conditionnel. L'idée principale est de diviser la scène en sous-espaces avec un *gater,* et d'entrainer un réseau assez petit pour le temps réel pour chaque sous-espace.

Notre approche consiste à utiliser un réseau auto-encodeur comme *gater* naïf, avec une contrainte additionnelle pour s'assurer d'une répartition équitable des sous-espaces. L'intuition ici est que l'auto-encodeur va représenter chaque point $(\mathbf{x}, \mathbf{v}, \mathbf{l}, \mathbf{n})$ de manière à encoder le plus d'information sur chaque point en un nombre limité d de dimensions (ici d est le nombre de sous-réseaux) dans un vecteur de probabilités h. On utilise une minimisation de la variance à travers les d classes de la probabilité moyenne à travers les K exemples d'une *batch*:

$$\mathcal{L}_{\mathcal{V}} = \eta \mathrm{Var}\left(\frac{1}{K}\sum_{i=1}^{K} P\left(h_j|x_i\right) \Big| j \in 1..d\right)$$

Ce coût additionnel à pour effet de forcer le réseau à ne pas toujours utiliser les mêmes sous-réseaux pour l'ensemble de la scène, et ainsi permettre une variété plus saine à travers la scène des sous-réseaux utilisés. On va se servir de cette représentation h pour choisir, étant donné un point, quel sous-réseau utiliser en prenant l'argmax de h. Comme de différentes représentations vont capturer des groupes distincts de la scène, cela risque d'être une division utile.

Bien que cette division spaciale marche pour une quantité limitée de sous-réseaux, il semble quand même y avoir une grande difficulté, avec les techniques d'optimisations qu'on utilise, à aller chercher le détail voulu.

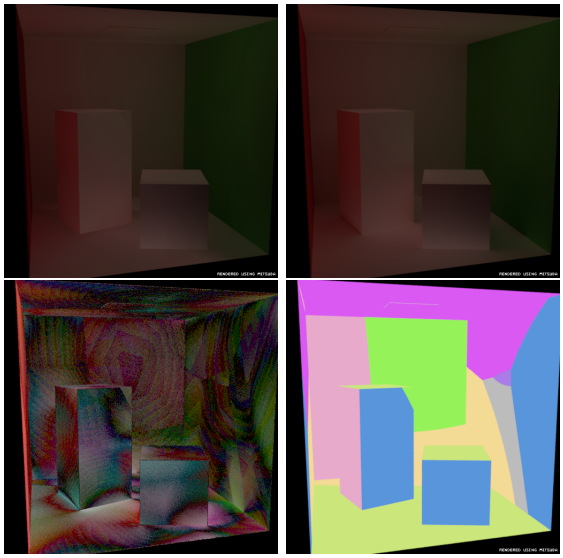

Figure 2, a) l'illumination indirecte produite par l'algorithme avec 10 sous-réseaux, b) l'illumination indirecte de référence produite par un path tracer, c) la log-différence des deux, d) la segmentation de la scène par l'algoritme.

Problèmes d'entrainement En particulier, on observe deux problèmes. Le premier est qu'il y a souvent des discontinuités aux frontières de décision du gater, à cause de la manière dont on entraîne les réseaux, il est possible qu'on ne donne pas assez de points limitrophes au réseau qui sont à l'extérieur de la région de décision, ou encore que le réseau accorde trop d'importance au points plus au "centre" de la région. Si c'est la cas, on pourrait vouloir utiliser des techniques de raffinement pondéré, soit donner un encore plus grand poids aux exemples qui sont les moins biens réussis (ce que fait naturellement le gradient, à un certain degré).

Le second problème est la manière dont on choisis les régions de décisions. Bien que ce soit la méthode la plus optimale de celles essayées, elle reste problématique, car elle ne prends en compte que la géométrie et non les couleurs associées aux points, et donc par exemple pour un effet haute fréquence sur un petit endroit d'une surface plane cela est néfaste.

Un dernier problème quand même notable est celui des gradients de couleurs. Les gradients appris par le réseaux sont "trop parfaits", si on les compare à l'image un peu bruitée produite par un pathtracer. Les gradients très

progressifs (comme c'est le cas dans la Cornell Box) vont donc être visibles à l'oeil comme des grosses bandes de la même couleur (on le voit bien dans la différence d'images de la Figure 2c).

Algorithmes essayés À cet effet, nous avons essayés différentes variantes de *gater*.

La première variante de *gater* consistait à pondérer la perte par la probabilté accordée par le *gater* à chaque sous-réseaux, ainsi les réseaux et le gater s'ajustaient mutuellement. Malheureusement, après un certain nombre d'itérations le gater ne s'ajustait presque plus, et ce malgré une répartition de la scène parfois assez mauvaise.

La seconde consistait à générer pendant l'entraînement une distribution multinomiale en fonction des probabilités données par le gater, et de sélectionner et d'entraîner sur une région les sous-réseaux plus probables. Ici, on avait le même problème mais pour une raison différente, la nature probabilitiste de la sélection se traduisant mal en un gradient stable, le *gater* ne s'optimisait pas très bien.

La troisième variante est celle des auto-encodeurs présentée plus haut.

5.2 Implémentation temps réel

On a implémenté naïvement l'algorithme à un réseau (20-10 neurones cachées) dans un shader GLSL et obtenu de bonnes performances, ce qui laisse suggérer qu'une meilleure implémentation permettrait d'utiliser un modèle bien plus complexe, par exemple un gater avec de nombreux sous-réseaux.

Sur un GPU GeForce GT 650M, l'illumination indirecte d'une image 1920x1080 prends 30ms à calculer.

6 Conclusions

Comparaison à l'état de l'art Un effet assez imposant de l'utilisation de la descente de gradient est le temps d'entraînement qui est assez grand. Si on compare avec les statistiques de temps et d'erreur rapportés par [8], leurs réseaux étaient beaucoup plus nombreux mais probablement aussi beaucoup moins précis et leurs temps d'entraînement rapportés sont du même ordre de grandeur que les notres, mais pour une quantité de deux à trois ordres de grandeur de plus de sous-réseaux, et surtout, pour une erreur numérique très semblable.

Améliorations futures Dans l'optique qu'il est possible d'améliorer le placement des sous-réseaux, il semble donc envisageable d'obtenir de meilleures performances à un coût similaire.

En effet, l'utilisation de techniques de calcul conditionnel dans les réseaux de neurones étant relativement nouvelle, il reste énormément d'options à explorer et on peut s'attendre à découvrir de nouvelles techniques qui vont permettre d'améliorer potentiellement dramatiquement les performances des modèles actuels. Il est clair qu'une répartition intelligente est ici la barrière majeure, et peut mener à une modélisation très efficace de l'illumination indirecte par peu de sous-modèles.

References

[1] Ronan Collobert, Yoshua Bengio, and Samy Bengio. *Scaling Large Learning Problems with Hard Parallel Mixtures*, volume 2388 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[2] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[3] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

[4] Wenzel Jakob. Mitsuba renderer, 2010. http://www.mitsuba-renderer.org.

[5] Alexander Keller. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[6] Samuli Laine, Hannu Saransaari, Janne Kontkanen, Jaakko Lehtinen, and Timo Aila. Incremental instant radiosity for real-time indirect illumination. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, EGSR'07, pages 277–286, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[7] Mangesh Nijasure, Sumanta Pattanaik, and Vineet Goel. Real-time global illumination on gpus. *Journal of Graphics, GPU, and Game Tools*, 10(2):55–71, 2005.

[8] Peiran Ren, Jiaping Wang, Minmin Gong, Stephen Lin, Xin Tong, and Baining Guo. Global illumination with radiance regression functions. *ACM Trans. Graph.*, 32(4):130:1–130:12, July 2013.

[9] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph.*, 21(3):527–536, July 2002.

A Annexe

Le code source utilisé pour ce projet est disponible à cette addresse: `github.com/bengioe/mlillumination/`

Hyperparamètres Les sous-réseaux utilisés avaient deux couches cachées de 20 et 10 neurones respectivement avec activations tanh et une couche de sortie avec activation sigmoïde. Pour les entraîner on utilise un taux d'apprentissage $\mu_0 = 0.5$ avec une diminution $\tau = 50$ selon l'époque e, ainsi qu'un *weight-decay* de 10^{-4}:

$$\mu_e = \frac{\mu_0 \tau}{\tau + e}$$

Fonctions de pertes Nous avons essayé différentes fonctions de pertes pour apprendre des valeurs RGB.

La fonction de perte traditionelle d'erreur quadratique moyenne, $\sum(y - f(x))^2$, semble performer assez mal puisque les erreurs sur les régions sombres sont sous-représentées. La sortie de ces réseaux est généralement trop claire.

La fonction de perte par valeur absolue, $\sum |y - f(x)|$ marche beaucoup mieux, non seulement en donnant une meilleure erreur numérique, mais aussi mieux les régions sombres.

La fonction que nous avons retenue, car elle donne les meilleurs résultats, est une fonction d'erreur relative, avec $\epsilon = 10^{-2}$, qui permet d'avoir un gradient plus équilibré envers les différentes teintes d'intensités différentes.

$$\mathcal{L}(x, y) = \sum \frac{|f(x) - y|}{y + \epsilon}$$