Rapport de Projet: Etape intermédiaire 2

Projet Annuel 3 Big Data

ALLEXANDRE Matthieu
HOLLANDER Stephane
TAVERNIER Antoine

22/04/2019

Table des matières

Introduction

Sujet

L'objectif de ce projet est d'implémenter et utiliser des modèles et algorithmes simples relatifs au Machine Learning, combiner le tout dans un cas pratique réel. Dans notre cas nous avons choisi la classification d'images. Nous cherchons à classer une image d'une capture de jeu vidéo et dire si cette image appartient au genre FPS, RTS ou MOBA.

Ce premier rapport a pour but de rendre compte de l'avancée du projet. L'objectif était de créer un dataset pour notre algorithme. De plus nous devions implémenter un neurone unique. Pour la classification nous utilisons la méthode de Rosenblatt pour l'apprentissage et nous allons essayer de voir comment les paramètres influent sur nos résultats.

Membres de groupe

Pour ce projet nous somme répartis par des groupes de 3 personnes. Nous sommes le groupe 6.

— ALLEXANDRE Matthieu (LittleSoap)
— HOLLANDER Stephane (stephaneArtist)
— TAVERNIER Antoine (Uranium2)

Analyse d'application

Cette section a pour but de présenter les structures et ainsi les fonctions principales de ce projet. Vous y trouverez aussi le modèle et les choix d'implémentation.

Structures principales

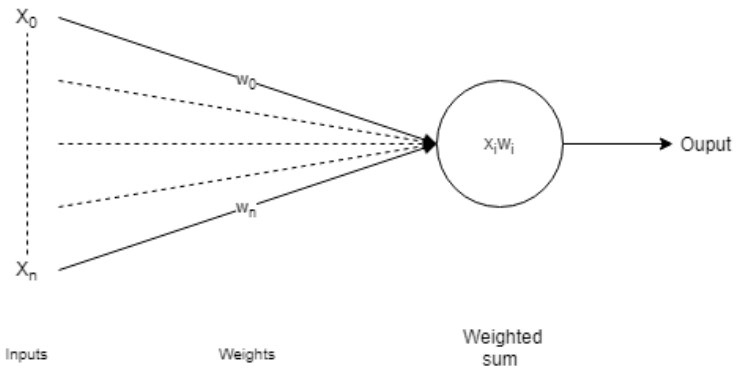
Structure Neuron

Ci-dessous la structure qui est le coeur de notre projet, le neurone. Le neurone permet de faire de simples calculs. Il va faire une somme pondérée de ses entrées. Dans cette forme basique nous avons un neurone dans sa version linéaire. Pour casser ce calcul linéaire il faut faire passer la sortie du neurone dans une fonction d'activation non linéaire tel que $tanh(X)$.

```
1 typedef struct neuron
2 {
3   double* inputs;
```

```
4   double* weights;
5   double output;
6   int nbInputs;
7   int typeActivation;
8   double bias;
9 } Neuron;
```

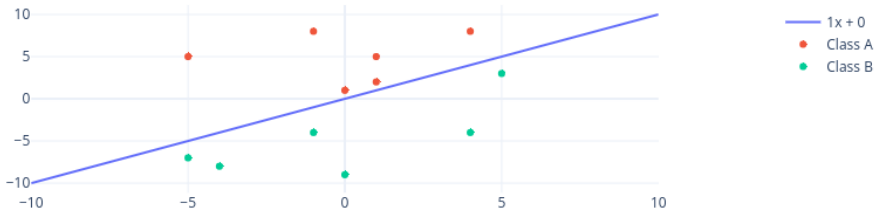
Les *inputs* sont les entrées. Pour une position donnée on l'écrit x_i.
Les *weights* sont les poids. Pour une position donnée on l'écrit w_i.



La somme pondérée des entrées du neurone s'écrit :

$$Output = \sum_i^n x_i w_i => \text{f(X)} = \text{aX}$$

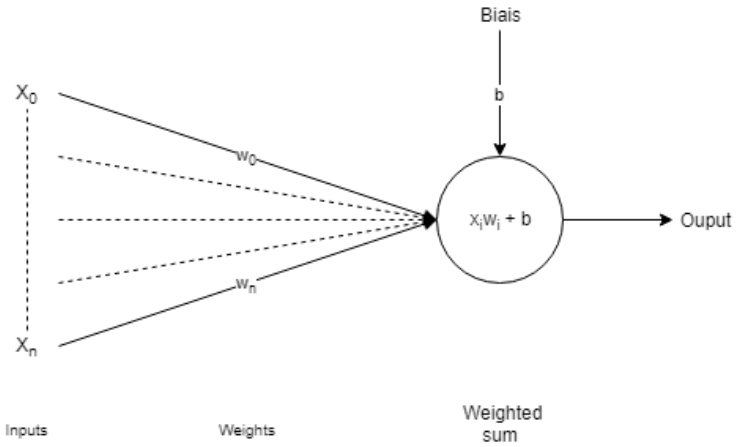
On se retrouve avec fonction linéaire de forme $f(X) = aX + b$ avec $b = 0$. On est donc capable de tracer une droite passant par l'origine.



Si nos données sont linéairement séparables par l'origine, alors ce modèle est suffisant. Cependant si nous avons un point de la classe A qui passe sous la droite $f(X) = 1X + 0$, alors nous ne sommes plus en mesure de classer nos données correctement. Il faut donc modifier l'ordonnée à l'origine b. Une autre solution à ce problème aurait été de normaliser les données pour les centrer afin de se retrouver avec $b = 0$.

On rajoute donc un biais à notre somme pondérée afin de générer l'ordonnée à l'origine de notre droite qui va séparer nos données.

$$Output = \sum_i^n x_i w_i + b => \text{f(X)} = \text{aX} + \text{b}$$



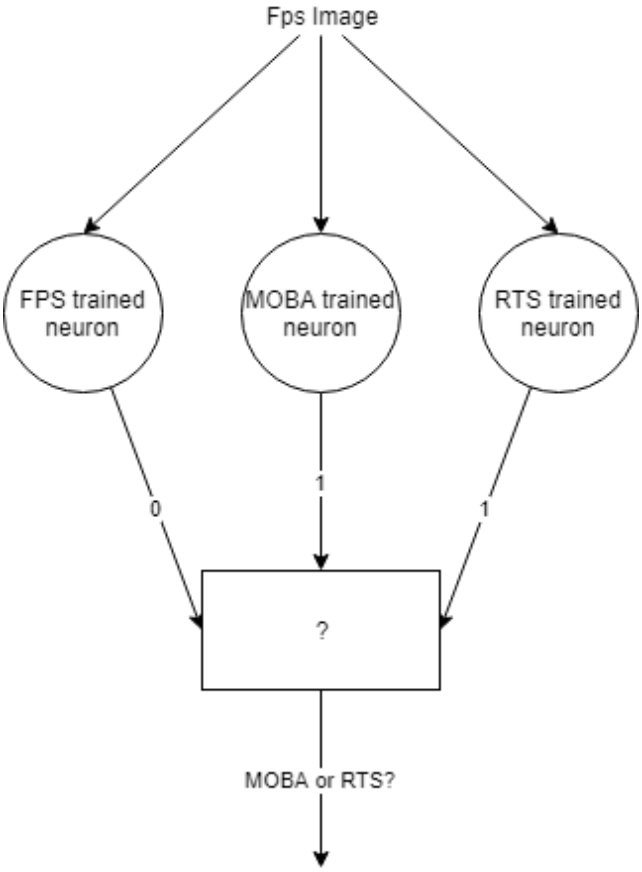
Dans le cas de notre projet nous avons en entrée des pixels d'une image. Pour des questions de simplicité nous avons choisi de ne pas prendre les 3 canaux d'un pixel, mais de conserver sa valeur en niveau de gris. Nous expérimenterons, plus tard, les couleurs, une fois convaincus du bon fonctionnement du neurone pour des tâches simples. Les pixels ont une valeur entre 0 et 255, imaginons une image totalement blanche (R :255 G :255 B :255). Que se passerait-il si nous faisions la somme pondérée de la valeur 255, nous risquerions d'avoir une valeur qui va exploser en infini. Pour remédier à ce problème nous normalisons nos valeurs d'entrées. Nous répartissons nos valeurs entre 0 et 1, c'est une normalisation min-max.

$$z = \frac{x - min(x)}{max(x) - min(x)}$$

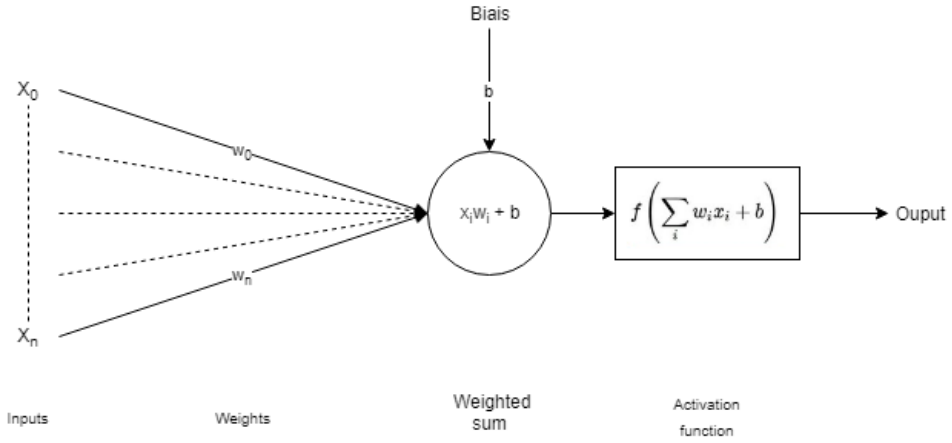
On se retrouve avec des valeurs plus petites sans pour autant perdre d'informations, cependant nous n'avons toujours pas résolu la possibilité de tendre vers l'infini lors de notre somme pondérée. En effet nous ne connaissons pas les valeurs possibles des poids. Il est possible qu'un pixel ait une importance infime dans le rôle de la détermination d'une image ce qui risque de créer un calcul qui tend vers l'infiniment petit. De même pour une valeur importante qui risque de produire une sortie très grande. Pour pallier ce problème nous voulons borner cette information afin de ne pas minimiser l'importance des valeurs extrêmes.

Le rôle de cette fonction d'activation a pour but de casser cette linéarité. La première fonction d'activation utilisé est celle par palier. On dit que si la

sortie du neurone est positive alors on peut affirmer que l'image appartient à la classe recherchée. C'est donc une classification binaire, "vrai" ou "faux". Mais est-ce applicable pour une classification de 3 classes ? Impossible pour le neurone de nous dire son avis sur 3 classes alors qu'il est binaire. Même si nous faisons 3 neurones indépendants que nous entrainons sur 3 classes séparément, que ce passe-t-il si 2 neurones nous sortent "vrais" ? Impossible de prendre une décision sur des valeurs aussi brutes. Il nous faut un intervalle de valeur plus large.



C'est pour cela que nous introduisons la fonction sigmoïde ($\frac{1}{1-e^x}$) ou la tangente hyperbolique ($tanh(x)$). La première solution va nous garantir une sortie comprise entre $[0, 1]$ alors que la seconde entre $[-1, 1]$. Dans notre cas nous avons choisi la fonction $tanh(x)$ pour conserver les valeurs négatives afin d'être plus sévère sur notre futur apprentissage sur les entrées peu importantes.



Règle de Rosenblat

Maintenant que nous savons comment fonctionne un neurone, nous pouvons modifier ses paramètres pour avoir une sortie qui reflète ce qu'on attend de lui. C'est à dire, prédire l'appartenance d'une image à une classe. Pour avoir la bonne sortie il faut jouer avec les poids du neurone. Nous verrons par la suite que d'autres paramètres rentrent en jeux, mais risquent de complexifier la tâche.

Pour modifier les poids du neurone il faut regarder la marge d'erreur ou la distance entre ce qu'on s'attendait à avoir et le résultat obtenu avec certains poids.

Nous devons créer un dataset d'image $XTrain$ ainsi que $YTrain$ les valeurs espérées suite à la prédiction du neurone. On modifie le poids du neurone si la sortie n'est pas identique à celle espérée grâce à cette formule :

$$w_i \leftarrow w_i + \alpha(y_i^k - g(x_i^k))x_i^k$$

— w_i : le poids d'un pixel x à la position i.
— α : le pas d'apprentissage fixé par l'utilisateur.
— y_i^k : la sortie attendue pour l'image k au pixel i.
— $g(x_i^k)$: la sortie calculée par le neurone pour l'image k au pixel i.
— x_i^k : la valeur du pixel de l'image k à la position i.

Nous suivons la règle de Rosenblatt qui permet de mettre à jour les poids d'un neurone en itérant sur les images qu'on lui fournit.

Maintenant que la théorie est posée, il serait intéressant de modifier les paramètres pour observer leur comportement sur un neurone.

Structure NeuralNet

```
typedef struct layer
{
  Neuron** neurons;
  int nbNeurons;
} Layer;

typedef struct neuralNet
{
  double* inputs;
  Layer** Layers; // Matrix of neurons
  int nbLayers;
  int* sizeLayers;
} NeuralNet;
```

Pour plus tard nous avons déjà entamé la structure possible qui servira à la construction d'un réseau de neurone. A l'instant présent cette structure semble robuste, du moins pour le réseau de neuron à plusieurs couches. Le principe est simple, un réseau de neuron possède des couches de neurones, cette couche est appelée *Layers*. Ce qui n'est finalement qu'une simple liste avec un *int* pour pouvoir itérer sur les élements de cette liste. Nous arrions pu utiliser les listes built-in en C++. Si nous avons des problèmes de performance, il faudra peut être revoir cette implémentation.

Lors de la construction d'un *NeuralNet*, nous définisson ses nombres de couches et de neurones par couche. Une fois les paramètres remplis, la liaison des couches est automatique. En effet chaque couche prend une liste neurone et chaque neurone prend une liste de *inputs*, quand nous construisons le réseau nous lions les *output* de tous les neurones de la couche précédente pour en faire les *inputs*. de la couche suivante.

Lors de la construction, les poids sont aussi aléatoires entre $[-1, 1]$ pour ne pas influencer les résultat. Une fois le réseau construit, nous faisons une première prédiction pour alimenter les *inputs* et *output* de chaque neurone de chaque couche, sinon nous aurions des poiteurs *null*.

Les biais et améliorations

Construction du data set

Notre première tâche a été de se créer notre propre jeu de donnée pour nourrir notre neurone. Nos images sont issues de jeux vidéo. Nous avons donc 3 possibilités pour trouver des images de jeux :

— Trouver les images sur Google ou banque d'image en ligne.
— Jouer aux jeux.
— Regarder les autres personnes jouer.

La première option est l'une des plus naïves quand on crée un dataset, on regarde s'il en existe déjà un, ou s'il est possible de les trouver facilement sur Google image. Nous avions un script python qui permettait de télécharger X images sur Google image avec certaines restrictions.

Le défaut majeur de cette méthode était qu'il fallait faire beaucoup de vérification humaine. En effet nous téléchargions des images de couvertures de jeux, de dessin, de cosplay et même des images de présentation de vidéo Youtube avec des gros caractères au milieu de l'image. Au final pour 100 images téléchargées, seules 10 étaient valides.

La seconde solution était de jouer nous-même à certains jeux et de nous enregistrer jouer. Cette idée était alléchante, agréable pour nous et aussi nous savions que nos images seraient très représentatives de ce que l'on cherche. Seul les menus et chargement du jeu devraient être écartés. Un défaut de cette méthode est le temps à investir pour créer une base de données de peu de jeu. Nous devions acheter de nouveaux jeux pour augmenter la variété de nos classes.

La dernière solution est venue très rapidement après la seconde, pourquoi s'embêter à jouer alors qu'il existe Twitch ? La plus grosse plateforme de streaming de jeu vidéo. Avec une base de 2.2 millions de streamer au quotidien, nous avons donc trouvé une source quasi infinie d'heure de jeux sur n'importe quels plateformes et type de jeux, le tout sans rien dépenser.

Cependant nous avons fait un choix sur le type d'image, le format et son contenu. En premier lieux nous nous sommes focalisés sur maximum 4 types de jeux par classe (FPS, MOBA, RTS). De plus nous voulions éviter les éléments superflus aux jeux, c'est à dire une webcam, une alerte, une image fixe. Le but, au début, est d'avoir une image la plus pure qui caractérise le plus possible une catégorie de jeu. Plus tard dans l'année il faudra rajouter ces éléments pour rajouter du bruit à l'image afin que l'algorithme soit plus robuste. Un atout de cette méthode est de pouvoir se créer en permanence un nouveau jeu de donnée que l'algorithme n'aura jamais vu auparavant.

Paramètre d'apprentissage

La taille du data set

N'ayant pas de repère sur le nombre d'image à avoir, nous avons fait des recherches pour savoir quelles étaient les bonnes pratiques pour réaliser un jeu d'image représentatif de la réalité. Nous avons à notre disposition des images contenants $1920 * 1080$ pixels, soit 2.073.600 pixels par images. D'après la litérature dédiée aux réseaux de neurones il faudrait prendre 100 fois plus

d'images que nous avons d'arguments. Il nous faudrait donc 207.360.000 images à diviser en 3 classes, soit 69.120.000 images par classes. Même si Twitch est une source infinie d'images pour peupler notre jeu de données, les nombres semblent astronomiques pour la tâche à effectuer. C'est pour cela que nous avons pris la décision de commencer par environ 1.500 images par classe soit un total de 4.500 images. Si nous devions respecter la proportion images par nombre de pixel, nous devrions utiliser des images avec 45 pixels. Tous ces chiffres sont strictement spéculatifs et ne sont pas déterministes, ils peuvent changer. En effet, pour un humain il est assez compliqué de différencier un type de jeu sur une image ne contenant que 45 pixels. Peut-être que la machine, oui ? Mais à quel prix ?

RGB or not RGB ?

L'idéal est de ne perdre aucune information dans nos images, cependant des choix ont été faits pour, en premier lieux, vérifier le bon fonctionnement d'un neurone. Nous avons donc donné des valeurs en niveau de gris au neurone pour s'assurer des bons calculs du neurone. Nous pensons qu'il est impératif de conserver l'information des couleurs. De plus il nous semble complexe pour un seul neurone de traiter autant d'informations à la fois et de s'attendre à des résultats fidèles. Plus tard dans l'année nous rajouterons les trois canaux de couleur, ensuite nous pourrons en déduire de sa réelle importance sur la vitesse d'apprentissage et sa précision.

L'époque

L'époque (Epoch) est le nombre d'itération que nous allons effectuer sur chaque images dans notre dataset. Il joue un rôle crucial sur l'apprentissage de l'algorithme. Une Epoch trop faible rendrait l'algorithme peu performant, alors qu'une trop grand mènerait à un apprentissage par coeur du dataset et risquerait de mal représenter la réalité lors des tests inconnus.

Le pas d'apprentissage

Le pas d'apprentissage nommé α est une valeur qui va nous permettre de gérer la finesse de notre apprentissage. Si l'on prend une petite valeur, nous allons nous rapprocher de façon lente vers notre objectif inconnu. Il faudra donc augmenter le nombre d'époque pour etre sûr d'atteindre notre objectif. Si le pas d'apprentissage est trop petit, certes nous serions précis dans notre recherche des poids, mais cela nous demanderait beaucoup plus de temps pour trouver l'objectif.

Si nous prenons un pas d'apprentissage trop élevé, nous nous rapprochons plus rapidement de notre objectif, par contre la probabilité d'arriver proche de notre objectif est faible. Il serait donc impossible d'être précis.

L'objectif ici est de jouer avec le pas d'apprentissage selon le nombre d'époque ou d'une autre métrique. C'est à dire avoir un pas d'apprentissage élevé pour les premiers cycles d'apprentisage et de diminuer le pas au fur et à mesure afin d'affiner la précision sur la recherche des poids.

Nous avons bien conscience de ce problème et des solutions possibles pour le résoudre, cependant à l'heure actuelle du projet nous mettons des valeurs selon nos envies pour ressentir les effets sur le résultat.

Test opérationnel

Avant même de vouloir tester notre neurone sur nos propres données. Nous avons créé un nouveau dataset contenant 3 classes. Chaque classe possède des images de taille $10 * 10$. Une classe contenant des images blanches, une des images noires et la dernière des images bleues. Le but ici est de vérifier le bon fonctionnement du neurone.

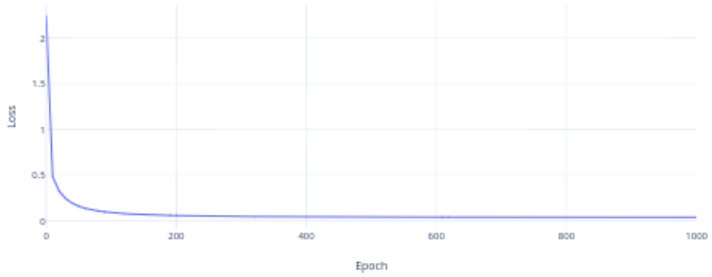
Nombre d'époques

Le premier test à été de charger 5 images de chaque classe mais avec des tailles de 1 pixel. Un taux d'apprentissage fin et un nombre d'époques très élevé. Nous savons que ce ne sont pas des paramètres paufinés, mais le but est de vérifier le fonctionnement du neurone et de comprendre le rôle de ses paramètres.

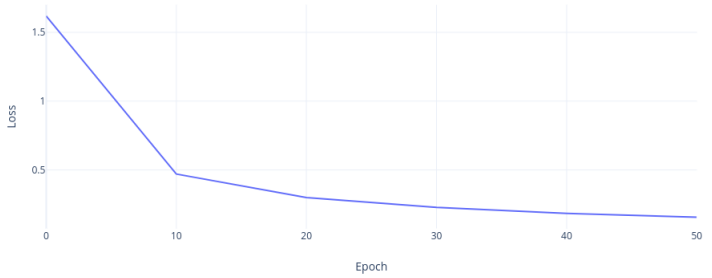
On charge donc 15 images, 5 de chaque classe de taille 1 pixel. Les classes sont les classes A, B et C. Ici nou voulons entrainer notre neurone a détecter une image de la classe A.

```
Please wait until we load 15 images of size 1x1
Epoch: 0 loss: 2.244073
Epoch: 10 loss: 0.483139
Epoch: 20 loss: 0.324155
Epoch: 30 loss: 0.242835
Epoch: 40 loss: 0.192996
...
Epoch: 960 loss: 0.039179
Epoch: 970 loss: 0.039170
Epoch: 980 loss: 0.039162
Epoch: 990 loss: 0.039154
Epoch: 999 loss: 0.039148
I think this image is from class: A
```

On obtient ce graphique ci dessous pour avoir une bonne représentation de nos résultats.



On observe donc que pour 1 pixel, 1000 époques c'est en effet bien trop. En effet si on refait le même test avec 50 époques on se retrouve presque au même resultat. On obtient 0.156150 Loss contre 0.039148 avec 1000 époques.

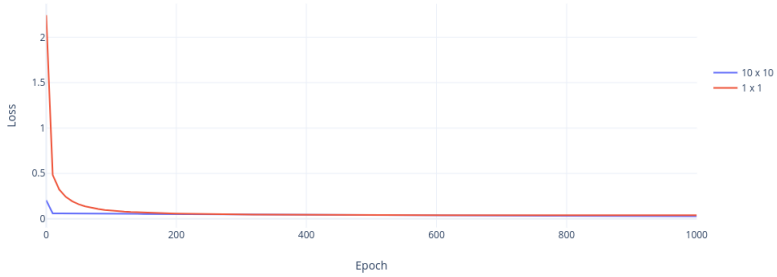


On se rend compte que le nombre d'époque est important pour définir la précision de notre résultat. Mais si nous faisons trop d'itération nous risquons le surapprentissage. En effet nous avons fait apprendre à notre neurone ce qu'était un pixel blanc (Classe A). Si nous lui donnons un pixel gris-blanc, alors les prédictions sont moins bonnes. Alors que si nous baissons le nombre d'époques nous donnons une marge d'erreur peut être plus grande, mais le neurone est plus souple dans ses résultats. Moins précis, mais s'adapte mieux.

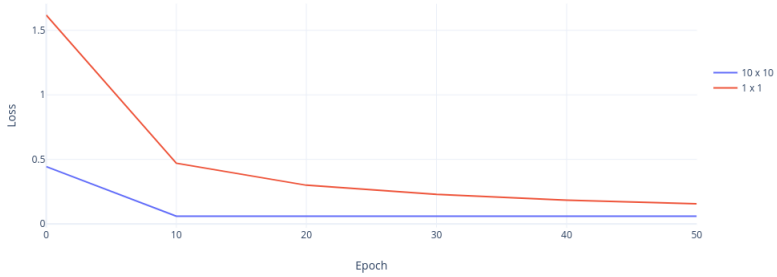
Nombre de pixels et d'images

Notre second test est basé sur le nombre de nos arguments du neurone. Nous avons seulement augmenté le nombre de pixels dans les images sans toucher

au nombre d'époque.



Plus le nombre de paramètres augmente plus l'apprentissage semble efficace. Maintenant une combinaison du nombre d'époques et de paramètres d'entrée.



Avec ce graphique plus zoomé on se rend compte que la première itération d'apprentissage 3 fois plus efficace que l'apprentissage sur une image de 1 pixel. On pourrait donc encore diminuer le nombre d'époques vu que nous sommes sur une ligne droite, pour la perdre à partir de 10 itérations ! Attention, augmenter le nombre de paramètres augmente la charge des calculs et donc directement du temps d'apprentissage. Le nombre d'images donne environs les mêmes résultats que l'augmentation du nombre de pixels.

Pas d'apprentissage

Le prochain test est sur l'influence du pas d'apprentissage. D'après nos tests, plus le pas d'apprentissage est petit plus notre recherche de minimisation de l'erreur sera précise, mais plus longue. Généralement si on diminue le pas d'apprentissage on augmente le nombre d'époques, sinon le neurone n'aura pas assez appris.

Test sur notre Dataset

Les tests effectués sur notre dataset contenant nos 3 genres de jeux sont plus aléatoires. En effet il est rare que le neurone arrive à déterminer à lui seul l'appartenance d'une image à l'une des trois classes. A chaque nouvel apprentissage nous avons des résultats différents des précédents, même en conservant les mêmes paramètres. Nous avons essayé de modifier les hyperparamètres pour trouver une stabilité dans nos résultats. Mais sur des images de $1090 * 1080$ pixels en niveau de gris avec un unique neurone, on ne sait pas ce qu'il cherche réellement à apprendre. Ce qu'il semble apprendre c'est le poids du niveau de gris sur d'énorme images. Ce qui rend la tâche très complexe. Il faudrait peut-être rajouter les couleurs pour comparer avec nos résultats actuels. Cependant nous doutons que rajouter la couleur influe beaucoup sur nos espérances d'apprentissage du neurone. Tout ce qu'il réalise c'est une modification de poids des pixels, donc une tâche très basique. Il va falloir faire évoluer notre modèle pour faire apprendre à notre réseau de neurones de nouvelle fonctionnalité qui permettraient peut-être de stabiliser nos prédictions sur une image de jeu vidéo.