Caractéristiques

Raw Pixels

Le premier *feature* utilisé est le très simple *raw pixels;* soit simplement la valeur de chaque pixel un après l'autre dans une liste. Nous avons essayé en premier lieu d'utiliser les tons de gris. Pour le Mnist qui est déjà en tons de gris, l'opération était triviale. Pour Cifar10, nous avons fait la moyenne des pixels rouge, vert et bleu. Présentement, nous utilisons plutôt la fonction « rgb2grey » de *scikit-image* que nous avons découvert un peu plus tard.

Nous avons ensuite essayé en alignant simplement chaque pixel pour faire un ordre RGBRGB. Les résultats ne sont pertinents que pour Cifar10 dans ce cas. Pour une configuration et un algorithme identique avec seulement 10% des données, l'*accuracy* avec les tons de gris et *cross-validation* (vérification avec l'ensemble de test) donne 16.21%. Elle est de 24.39% pour RGBRGB.

Quelques tests subséquents vont confirmer que les tons de gris baissent systématiquement, du moins avec nos *features* et nos algorithmes, la capacité des modèles à reconnaître une image. Cela semble logique puisque la couleur est pour l'humain une information supplémentaire nous permettant de distinguer des objets dans notre environnement. On réduit ainsi la quantité d'information utile.

Finalement, pour ce *feature,* nous avons aussi essayé avec une configuration RRGGBB; soit d'avoir tous les pixels pour le rouge au début, puis les verts et finalement les bleus. L'*accuracy* donne 23.71% ce qui est légèrement inférieur à notre ordre RGBRGB. Ceci nous a étonnés. Nous nous serions attendus à avoir un meilleur score puisque l'ordre de l'image est ainsi préservé. D'aligner les pixels donne une impression d'être quasi-aléatoire. Finalement, il semblerait que l'ordinateur y voit un petit quelque chose de plus. Nous avons retiré ce *feature* dans la version finale puisqu'il ressemble trop à RGBRGB et que ce dernier est légèrement meilleur.

Même si nous ne rapportons qu'un résultat pour chacun (et non des moyennes), nous avons essayé plus d'une fois et les résultats tournent autour des mêmes valeurs. Le RGBRGB reste le meilleur pour ce *feature* avec l'algorithme utilisé; soit le classificateur SGD de base avec comme *loss* la fonction logistique. Nous généralisons donc dans ce cas-ci.

	Logistic Regression (Training)	Logistic Regression (Test)	Adaboost (Training)	Adaboot (Test)
Mnist	94.03%	92.52%	85.92%	86.08%
Cifar10	48.17%	38.54%	39.66%	35.07%

L'Adaboost donne systématiquement de moins bons résultats que la régression logistique pour les *Raw*

Pixels. Un point sur lequel il faut toutefois faire attention; l'algorithme de régression logistique que nous avons est plus optimisé que celui d'Adaboost. Il est possible qu'Adaboost donne de meilleurs résultats que ceux-ci en lui passant de meilleurs paramètres ou en exécutant des optimisations (incluant de faire du prétraitement).

Local Binary Patterns

Nous avons essayé par la suite d'implémenter le *Local Binary Patterns* (*LBP*). Plus précisément, d'aller chercher une implémentation dans une librairie gratuite sur internet et de l'utiliser dans notre code. Le premier essai a été avec la librairie « Mahotas » pour python. Soit nous avions un problème au niveau de la compréhension de l'algorithme, soit la faible documentation faisait en sorte que nous ne recevions pas exactement ce que nous pensions, ou soit un beau mélange des deux. Quoi qu'il en soit, les résultats obtenus pour les deux ensembles de données n'étaient vraiment pas satisfaisants.

La libraire offre deux algorithmes; soit « lbp » et « lbp_transform ». Ils ne doivent certainement pas donner la même chose, mais c'est le deuxième qui donnait des résultats assez corrects pour nous donner de l'espoir. Nous avons essayé d'ajuster les deux paramètres, soit le nombre de points à prendre et le rayon où prendre les points.

Pour Mnist, les 10 meilleures paires de paramètres donnent des *accuracies* entre 87.5% et 89.33% (moyenne de 88.5%). C'est sensiblement ce que nous obtenions avec les *raw pixels*. En moyenne dans ce sous-ensemble des résultats, nous prenons 1.8 point et un rayon de 5.55. La moyenne est simplement pour donner une idée, et non pour choisir les paramètres.

Pour Cifar10, les 10 meilleures paires de paramètres donnent des *accuracies* entre 24.4% et 26.8% (moyenne de 25.16%). Encore une fois, c'est sensiblement ce qui nous obtenions avec les *raw pixels*. En moyenne dans ce sous-ensemble des résultats, nous prenons 2.9 points et un rayon de 5.3. Cela suggère que le rayon optimal pour les deux est probablement similaire, mais que Cifar10 a besoin d'un peu plus de points en moyenne que pour Mnist.

Nous avons essayé aussi de combiner les résultats de différentes « lbp » afin de faire un *feature* un peu plus gros. Pour certains paramètres, le résultat s'améliorait! Malheureusement, ce n'était pas vraiment de beaucoup. À partir d'environ trois « lbp », ça devenait trop long à calculer en le passant dans un algorithme de classification.

Quoiqu'il en soit, les résultats sont si proches de *raw pixels* que nous nous demandions si nous avions fait quelque chose de mal. Nous avons décidé de prendre une autre librairie, soit « scikit-image », afin de comparer. Cela donnait la même chose. Étant dans le gros doute, nous avons décidé de laisser faire ce *feature* et d'en prendre un autre à la place.

Histogram of Oriented Gradient (HoG)

La librairie « scikit-image » offrant le HoG, nous avons décidé de l'essayer. Intuitivement, nous pensions pouvoir obtenir de meilleurs résultats avec les images de Cifar10 puisque le *feature* extrait peut aller chercher des informations à propos de la forme. Nous ne pensions pas obtenir vraiment de meilleurs résultats pour Mnist toutefois.

Rendus à celui-ci, nous n'avons pas cherché à modifier les paramètres. En premier lieu, nous avions des considérations de temps. Deuxièmement, il s'est souvent avéré que les paramètres de base étaient les meilleurs de toute façon. Une investigation rapide en essayant 5-6 nombres différents d'orientations donnait en effet que la valeur de base, soit 9, est possiblement la meilleure. Les deux autres sont « pixels_per_cell » (défaut : (8, 8)) et « cells_per_block » (défaut : (3, 3)). Essayer aveuglément des valeurs ne marche pas dans ce cas et les valeurs sont de toute façon les mêmes que dans les notes!

Pour Cifar10, nous avons essayé en premier lieu de passer une image en tons de gris à l'algorithme. Il s'est avéré là aussi que de donner les *raw pixels* avec l'ordre RGBRGB est supérieur que de réduire l'information en passant en tons de gris. Alors bien que la documentation de « scikit-image » indique de passer une image en tons de gris, nous arrivons à mieux avec les couleurs.

	Logistic Regression (Training)	Logistic Regression (Test)	Adaboost (Training)	Adaboot (Test)
Mnist	90.08%	90.80%	79.38	81.30%
Cifar10	44.65%	43.26%	32.02%	31.74%

Tout comme pour *Raw Pixels,* HoG donne de meilleurs résultats avec la régression logistique. Cela suggère que notre Adaboost, du moins avec les paramètres que nous avons, est systématiquement inférieur à une simple régression logistique.

Algorithmes

Stochastic Gradient Descent (SGD) (Loss : Logistic Sigmoid)

Notre premier essai a été avec un des exemples de base du cours, soit un classificateur linéaire avec descente de gradient stochastique et une fonction logistique comme *loss*. Nous utilisons « SGDClassifier » de « scikit-learn ».

Cette fonction possède plusieurs paramètres et nous avons essayé de trouver ceux qui nous permettraient d'obtenir les meilleurs résultats (*grid search*). Le nombre de paramètres diffère aussi

selon le « learning_rate » que nous choisissons. Les tableaux présentent des informations sur les 10 meilleurs résultats de chaque catégorie pour chacun des deux ensembles de données. L'information présentée n'est pas complètement comparable puisque le nombre de paramètres à évaluer est différent pour chacun. L'optimal a donc beaucoup moins d'éléments ce qui peut pousser sa moyenne vers le bas. À l'opposé, l'inverse *scaling* en a bien plus ce qui peut le favoriser.

Pour Mnist :

learning_rate	#1	#10	Moyenne
Constant	88.33%	86%	86.77%
Optimal	87.17%	86.17%	86.53%
Inversed scaling	89.33%	88.33%	88.63%

Pour Cifar10 :

learning_rate	#1	#10	Moyenne
Constant	28.2%	25.4%	26.6%
Optimal	28.2%	24.2%	25.48%
Inversed scaling	30.4%	29.2%	29.66%

On remarque quand même qu'on peut obtenir de meilleurs résultats avec le « learning_rate » *inversed scaling*. Le #1 de chacun possède exactement les mêmes paramètres. Si on regarde le mode pour les paramètres dans les 10 meilleurs, nous obtenons, sauf dans un cas, exactement les mêmes paramètres que le #1. Pour Mnist : alpha=0.0001 (60%), eta0=1 (20%, le mode est plutôt 5 avec 40%), n_iter=5 (50%), power_t=0.5 (50%). Pour Cifar10 : alpha=0.0001 (50%), eta0=1 (30%), n_iter=5 (40%), power_t=0.5 (50%). C'est donc les paramètres que nous avons retenus.

Logistic Regression

Malgré tout, ce n'est pas celui-là que nous avons retenu. Au départ, peut être avons-nous lu trop vite, mais nous pensions que « LogisticRegression » n'était que la fonction précédente, mais isolée. À la manière de « rgb2grey » et « rgb2gray » qui offrent deux façons différentes de faire la même chose (mais selon les préférences d'écriture / région du locuteur dans ce cas). Nous l'avons donc utilisé innocemment. Ce n'était pas toutefois pas la même chose et c'était même beaucoup mieux!

Pour donner une idée, en utilisant toutes les données et le *feature raw pixels*, le SGD nous donne 88.44% pour Mnist (*cross-validation*) en comparaison à 92.52% pour la régression. Dans le cas de Cifar10, la différence est plus impressionnante. Nous sommes passés de 22.39% à 38.54%.

Pourtant, en lisant la documentation, « SGDClassifier » implémente bel et bien une régression logistique. L'algorithme varie peut-être? Où est-ce nos hyperparamètres qui sont inadéquats? Le « SGDClassifier » en possède en effet beaucoup. En comparaison, le « LogisticRegression » possède plutôt des paramètres par défaut que nous n'avons pas eu à toucher pour la plupart. Nous n'avons joué qu'avec le type d'algorithme d'optimisation, soit le paramètre « solver », et le paramètre « multi_class » qui ne changeait au final pas grand-chose à nos résultats.

Notre hypothèse repose justement sur l'algorithme d'optimisation. Pour les cas avec plusieurs classes, « LogisticRegression » offre « newton-cg » (Nonlinear Conjugate Gradient?), « sag » (Stochastic Average Gradient) et « lbfgs » (Broyden-Fletcher-Goldfarb-Shanno?). Par défaut, il fait une classification linéaire (« liblinear ») dont les résultats sont similaires à nos résultats avec SGD. La documentation indique que cet algorithme ne peut pas apprendre réellement un modèle à plusieurs classes et il utilise une certaine stratégie (*one-versus-rest*) afin de se comporter comme un classificateur à plusieurs classes (http://scikit-learn.org/stable/modules/linear_model.html#logistic-regression).

Bref, nous avons regardé les résultats pour chacun des trois « solver » multi-classes. Sauf pour l'*accuracy* en entraînement qui varie un peu, les résultats sont pratiquement identiques avec le *accuracy* de test. Nous avons retenu le Stochastic Average Gradient puisque la documentation indique qu'il peut être plus rapide dans le cas de gros ensembles de données. Nous ne savons pas si nos ensembles se qualifient comme étant de gros ensembles de données, mais puisqu'il n'y a pas plus de différence, c'est un gain potentiel qui ne peut pas faire de mal.

	Raw Pixels (Training)	Raw Pixels (Test)	HoG (Training)	HoG (Test)
Mnist	94.03%	92.52%	90.08%	90.80%
Cifar10	48.17%	38.54%	44.65%	43.26%

Le *feature* HoG donne de meilleurs résultats pour les images contenant plusieurs formes complexes et le *raw pixels* est légèrement meilleur pour Mnist. Nous retrouvons aussi notre plus grande différence entre l'*accuracy* des images utilisées en entraînement et celle obtenue en validant à l'aide d'un ensemble d'images que le modèle n'a jamais vue. 10% ne se qualifie pas vraiment comme de l'overfitting toutefois.

Adaboost

Nous voulions utiliser un algorithme de type « Bagging » afin de voir si notre classificateur précédent pouvait être amélioré en en créant plusieurs et en les faisant voter. Notre choix s'est arrêté,

un peu arbitrairement il faut dire, sur Adaboost. Il offre des améliorations intéressantes au simple Bagging sans être un Random Forest (nous connaissions une autre équipe qui l'utilise et voulions être originaux).

En théorie, nous pensions qu'il serait évident que nous allions gagner ne serais-ce qu'un peu en précision. Dans les faits, ça ne s'est pas avéré aussi facile. Premièrement, nous allons parler du cas de l'algorithme SAMME (*Stagewise Additive Modeling using a Multi-class Exponential loss function*) et du temps de calcul.

En utilisant notre *LogisticRegression*, nous pourrions peut-être obtenir de meilleurs résultats que ce que nous obtenons avec le SGD. Nous disons « peut-être », car le temps de calcul est beaucoup trop long même avec de petits ensembles de données. Il devient alors inconcevable avec notre puissance de calcul d'envisager faire un *grid search* sur celui-ci.

Si nous utilisons l'algorithme SAMME plutôt que SAMME.R (*real boosting algorithm*), c'est que le SGD ne supporte apparemment pas le calcul de la probabilité d'être dans une certaine classe ce qui est obligatoire afin d'utiliser le deuxième algorithme. Un peu comme avec le problème précédent concernant le *Regressor*, nous pensons que le calcul de probabilité joue beaucoup sur la précision de nos classificateurs. Malheureusement, utiliser un algorithme qui le permet est très exigeant sur nos pauvres ordinateurs avec Adaboost et nous ne pouvions pas espérer sortir des résultats satisfaisants à temps.

Nous nous contentons donc d'un véritable classificateur faible, soit notre SGD. Nous fondions alors tous nos espoirs sur les paramètres à ajuster, soit le nombre d'estimateurs et le taux d'apprentissage.

Pour Mnist :

Feature	Écart Min-Max	Moyenne	Meilleur paramètre (n_estimators, learning_rate)
Raw Pixels	[85, 90]	88.21%	([5, 25], 0.001)
HoG	[12, 77]	29.80%	(500, 0.5)

Pour Cifar10 :

Feature	Écart Min-Max	Moyenne	Meilleur paramètre (n_estimators, learning_rate)
Raw Pixels	[13.8, 34.2]	30.62%	(75, 0.000001)
HoG	[11.4, 29.2]	17.60%	([100, 500], 0.5)

On remarque tout de suite qu'il est possible d'obtenir de très bons gains en performance pour le *feature*

HoG en ajustant les paramètres. La majorité des combinaisons donne des résultats faibles comme en témoignent les deux moyennes, mais il existe des paramètres permettant d'obtenir de bons résultats. Les paramètres du meilleur résultat sont d'ailleurs identiques ce qui nous facilite la tâche.

Pour les *raw pixels*, nous avons choisi 75 estimateurs et un taux d'apprentissage de 0.000001. En général, pour ce *feature*, les meilleurs résultats s'obtiennent en utilisant moins d'estimateurs et un taux d'apprentissage très faible. C'est l'inverse avec HoG. C'est pourquoi nous avons des paramètres différents pour les deux.

	Raw Pixels (Training)	Raw Pixels (Test)	HoG (Training)	HoG (Test)
Mnist	85.92%	86.08%	79.38%	81.30%
Cifar10	39.66%	35.07%	32.02%	31.74%

Pour une raison étrange, le Adaboost donne toujours un meilleur score de test, même si de peu, que d'entraînement. Comme discuté plus haut, le Adaboost fait réduire systématiquement notre *accuracy*.

Underfitting et Overfitting

Nos résultats obtenus en utilisant 90% des données de l'ensemble d'entraînement ne présentent pas vraiment de cas d'overfitting. Nous en avions par contre avec Cifar10 et le feature *raw pixels* quand nous n'utilisions que 10% des données afin de grandement réduire le temps de calcul. Avec LogisticRegression, nous avions par exemple un score en entraînement de 68% et en test de 33%. Avec le SGDClassifier, nous avions encore pire avec un score en entraînement de 86.27% et en test de 24.39%! Le classificateur apprenait donc notre petit ensemble de données par coeur et avait alors beaucoup de difficulté à utiliser ses connaissances face à un ensemble de données qu'il n'avait jamais vu. Surtout que nous ne réduisions pas l'ensemble de tests qui était alors très imposant en comparaison. La différence entre entraînement et test s'est grandement réduite avec l'utilisation d'un plus grand ensemble de données d'entraînement.

Concernant l'underfitting, nous en avions surtout au début avec Cifar10. Avec le SGDClassifier et 90% des données, nous avions par exemple un score en entraînement de 24.41% et en test de 22.39%. Pour ce cas-ci, l'augmentation du nombre de données n'a pas aidé. Il a juste fait disparaître l'overfitting comme présenté dans le paragraphe précédent. Pour améliorer l'underfitting, nous avons dû changer carrément d'algorithme. Cela nous a permis de pratiquement doubler notre *accuracy* pour Cifar10.

Matrices de confusion

Pour faire les matrices de confusion, nous avons utilisé la fonction de Scikit Learn. Les lignes représentent la bonne réponse et les colonnes représentent la prédiction du classificateur.

	0	1	2	3	4	5	6	7	8	9	
0	956	0	1	4	1	5	6	2	4	1	980
1	0	1117	6	2	0	1	3	1	5	0	1135
2	4	11	923	15	9	4	15	9	30	3	1032
3	4	0	17	924	2	24	3	9	20	7	1010
4	1	2	6	3	918	0	8	6	10	28	982
5	11	3	2	37	10	767	15	8	34	5	892
6	7	3	5	3	3	20	913	2	2	0	958
7	1	8	22	10	5	1	0	945	3	33	1028
8	9	12	6	22	6	22	7	13	864	13	974
9	8	7	1	8	24	5	0	21	11	924	1009
	1001	1163	989	1028	978	849	970	1016	992	1014	10000

Mnist --- Logistic Regression --- Raw Pixels
Test Accuracy: 92.51%

Notre meilleur résultat avec Mnist a été obtenu en utilisant la régression logistique et en alignant tout simplement les pixels selon l'ordre RGBRGB. Le modèle réussit assez bien à prédire les chiffres même si nous n'atteignons pas un score aussi impressionnant qu'avec les réseaux de neurones.

Parmi les erreurs les plus fréquentes, on remarque que le 5 a été confondu plusieurs fois pour un 3 et vice-versa. Le 2 et le 5 ont été confondus pour un 8 assez souvent aussi. Tout comme le 4 et le 7 qui ont été confondus pour un 9. Les deux chiffres les plus souvent mal classés sont le 5 avec 125 mauvais classements et le 8 avec 110.

Parmi les erreurs les moins fréquentes, nous avons le 0 avec seulement 24 chiffres mal classés et le 1 avec seulement 18 chiffres.

Si on regarde maintenant les prédictions les plus souvent fausses, nous avons le 8 encore une fois qui a été faussement prédit pour 128 chiffres. Nous avons en deuxième position le 3 avec 104 fausses prédictions. En contrepartie, le 0 et le 1 continuent d'être dans les meilleurs avec seulement 45 et 46 mauvaises prédictions respectivement.

	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck	
airplane	512	35	73	22	46	9	34	44	189	36	1000
automobile	30	554	11	26	14	14	55	37	124	135	1000
bird	100	22	289	59	146	119	106	68	49	42	1000
cat	68	76	101	160	86	140	155	90	44	80	1000
deer	39	17	90	43	405	57	178	118	30	23	1000
dog	29	29	107	64	93	363	122	110	30	53	1000
frog	19	51	55	47	124	45	562	27	24	46	1000
horse	33	16	48	35	98	83	66	540	24	57	1000
ship	168	120	28	16	34	21	36	44	453	80	1000
truck	41	205	15	43	16	32	29	40	91	488	1000
	1039	1125	817	515	1062	883	1343	1118	1058	1040	10000

Cifar10 --- Logistic Regression --- HoG
Test Accuracy: 43.26%

Pour ce qui est de Cifar10, c'est encore une fois en utilisant la régression logistique que nous avons obtenu nos meilleurs résultats. Le *feature* HoG s'est toutefois avéré plus efficace pour extraire des caractéristiques pertinentes aux images.

Parmi les erreurs les plus fréquentes, nous avons le *truck* souvent pris pour une automobile, le bateau souvent pris pour un avion et l'avion souvent pris pour un bateau. Pour les animaux, les grenouilles et les oiseaux ressemblent apparemment assez aux cerfs pour causer plusieurs erreurs. Les chats et les chiens aussi, mais dans ce cas, il est plus simple de faire un lien de ressemblance.

Il n'y a pas vraiment une classe qui se distingue des autres comme étant plus facile à classer. Les grenouilles et les avions sont les deux « meilleures », mais elles le sont de peu. En moyenne, 433 images sur 1000 ont été bien classées. En contrepartie, il y a des classes qui performent beaucoup moins bien que la moyenne. Les oiseaux et les chats ont rarement été bien identifiés.

C'est par contre les grenouilles et les cerfs qui ont été le plus faussement prédits avec 781 et 657 images respectivement. Les chats n'ont eu que 355 mauvaises prédictions, mais le classificateur semble juste ne pas vouloir se prononcer trop souvent pour cette classe.

Réseau de neurones

Multilayer Perceptron

Pour la mise en place du multilayer perceptron, nous avons basé notre code sur un exemple fourni sur le github de Keras[1]. Le code reste très simple, puisque le nous n'avons que des layers qui sont, dont les neurones, sont entièrement connectés. Nous séparons tout d'abord nos données en 3 ensembles, ceux d'entrainement, de test et de validation. Pour nos tests, nous n'avons pas fait varié le nombre de layers, qui reste à 6, ni l'organisation du network. C'est-à-dire que chaque layer fully connected est suivie d'une activation de type ReLu et d'un dropout de 0.25, à l'exception du dernier dropout qui est de 0.5 et de la dernière activation qui est un softmax.

Nous avons fait nos tests en deux étapes. Tout d'abord, pour chaque base de donnée, nous testons trois hyper paramètre, soit le nombre de batch, le nombre de filtres, le nombre d'epoch. Pour Mnist, ces tests sont faits avec le classificateur Adadelta, alors que pour Cifar10, nous avons utilisé SGD.

La deuxième partie de nos tests servent à trouver les meilleurs hyper paramètres pour les classificateurs. Nous utilisons les meilleurs paramètres obtenus lors des premiers tests, et nous cherchons à trouver les meilleurs hyper paramètres pour Adadelta et SGD.

Tous nos résultats en détail peuvent être retrouvés dans les fichiers excel *Grid_Search_MPL.* Cela inclus les paramètres de chaque test. Nous nous concentrerons sur les résultats intéressants.

Mnist

Pour tester les différents paramètres sur Mnist, nous avons choisi d'utiliser le classificateur Adadelta avec les valeurs par défauts. Adadelta, un classificateur dérivant d'Adagrad, est reconnu

[1] https://github.com/fchollet/keras/blob/master/examples/mnist_mlp.py

comme étant un peu sensible au changement de certains de ses paramètres tel le learning rate. Puisque Mnist ne contient que des images simples, Adadelta avec ses paramètres par défaut nous semblait un bon choix pour éviter d'introduire un biais à nos tests sur le nombre de batch, filtres et epoch. Nous avons testé sur 100% des données.

Hyper paramètres de batch, epoch et filtre pour Mnist

Résultat pour nombre de batch (100% des données utilisées):

Batch Size	128	256	512	1024
Nombre d'epoch	12	12	12	12
Nombre de filtre	128	128	128	128
Validation Loss	0,1113	0,1155	0,1181	0,138
Validation Accuracy	0,9705	0,9692	0,9672	0,963
Test Score	0,1207	0,124	0,124263	0,1404
Test Accuracy	0,9692	**0,9669**	**0,9648**	**0,9612**

Il n'est pas très surprenant que les résultats varient peu d'un batch size à un autre. Mnist n'est pas un ensemble de données très énorme et il est probable que nous convergeons rapidement vers une accuracy optimal après peu d'entrainement.

Résultat pour le nombre de filtres (100% des données utilisées) :

Batch Size	128	128	128	128
Nombre d'epoch	12	12	12	12
Nombre de filtre	128	512	1024	2048
Validation Loss	0,1113	0,0714	0,078	0,085
Validation Accuracy	0,9705	0,9835	0,984	0,9837
Test Score	0,1207	0,0792	0,08641	0,096309
Test Accuracy	**0,9692**	**0,9819**	0,9829	**0,9821**
Temps moyen (min)	1	3	8,5	27

Pour ces tests, nous avons rajouté une donnée intéressante, soit le temps de calcul. Comme on peut s'y attendre, plus l'on ajoute de filtres, meilleur sont les résultats, car le réseau apprend plus à chaque layer. Toutefois, il arrive un point ou doublé le nombre de filtres triple le temps de calcule et donne des résultats très similaires, voire moins bons. Dans le cas présent, considérant la différence entre l'accuracy pour 512 filtres et pour 1024 qui est de 0.001, il pourrait être plus intéressant de prendre moins de filtres et minimisé le temps de calcule.

Résultat avec différent epoch (100% des données utilisées):

Batch Size	128	128	128
Nombre d'epoch	12	24	48
Nombre de filtres	128	128	128
Validation Loss	0,1113	0,1022	0,1151
Validation Accuracy	0,9705	0,973	0,9775
Test Score	0,1207	0,10872	0,127
Test Accuracy	**0,9692**	**0,9735**	0,9757

Le nombre d'epoch affecte considérablement le temps de calcule, car nous faisons une forward pass et backward pass sur l'ensemble des données d'entrainement à chaque epoch. Nous nous serions attendus à voir une plus grande différence d'accuracy entre les différents nombres d'epoch. Mnist étant toutefois un ensemble de données relativement facile à faire l'apprentissage, nous commençons déjà à plafonner avec 12 epoch.

Hyper paramètres de classificateur pour Mnist

Pour adadelta, nous avons regardé le learning rate et l'epsilon (fuzz factor), alors que pour SGD, nous avons regardé le learning rate et le momentum.

Learning rate :

Adadelta (100% des données)

Adadelta	**Lr=0.01**	**Lr=0.1**	**Lr=1**	**Lr=1.5**
Loss	0,0568	0,0558	0,1841	0,9137
Accuracy	0,9845	0,9839	0,9506	0,7063
Validation Loss	0,0967	0,0784	0,1212	0,5323
Validation Accuracy	0,9778	0,9818	0,9668	0,856
Test Score	0,1127	0,07762	0,12744	0,5344
Test Accuracy	**0,9763**	0,9803	**0,9635**	**0,8523**

SGD (10% des données)

SGD	Lr=0,001	Lr=0.01	Lr=0.1	Lr=1	Lr=1.5
Loss	1,4891	0,2307	0,5147	14,1404	14,6048
Accuracy	0,5107	0,9335	0,8752	0,1033	0,0939
Validation Loss	1,0598	0,2121	0,2445	14,4526	14,8018
Validation Accuracy	0,7133	0,94	0,9267	0,1033	0,0817
Test Score	1,0749	0,2401	0,39	14,4611	14,5481
Test Accuracy	**0,6943**	0,9308	**0,9094**	**0,1028**	**0,0974**

Comme mentionné, Adadelta est moins sensible au changement de learning rate que SGD. On se serait toutefois attendu à avoir de meilleur résultat avec le learning rate par défaut, soit 1.0 . On voit qu'au-dessus de 0.1, le learning rate pour SGD donne rapidement des résultats aberrants et ne permet plus de reconnaitre les images de Mnist. C'est ce qui rend Adadelta intéressant comme classificateur, le fait qu'il donnera de bons résultats sans avoir besoin de chercher le meilleur hyper paramètre possible.

Fuzz Factor et Momentum :

Nous avons retenu le meilleur learning rate pour chacun des classificateurs avant de modifier leur second hyper paramètre.

Learning rate = 0.1

Adadelta	E=1e-08	E=1e-07	E=1e-06	E=1e-05	E=1e-04
Loss	1,0636	0,5395	0,3987	0,3611	0,3579
Accuracy	0,6531	0,8352	0,8846	0,8939	0,8959
Validation Loss	0,6981	0,3704	0,3232	0,3118	0,3157
Validation Accuracy	0,8	0,8767	0,9183	0,91	0,9083
Test Score	0,7	0,3749	0,335	0,3245	0,3335
Test Accuracy	**0,7936**	**0,8894**	**0,8996**	0,9012	0,9018

Learning rate = 0.01

SGD	M=0.9	M=0.8	M=0.7
Loss	0,2407	0,3628	0,4782
Accuracy	0,9417	0,8961	0,8593
Validation Loss	0,3547	0,2619	0,3206
Validation Accuracy	0,915	0,9117	0,8933
Test Score	0,4503	0,289	0,3373
Test Accuracy	**0,9081**	0,9148	**0,9006**

Dans les deux cas, la modification de l'hyper paramètre n'apporte pas autant d'amélioration à l'accuracy que pour le learning rate. Adadelta est celui qui fait le plus grand gain, avec 1% entre un epsilon de 1e-08 et 1e-04.

Nous avons testé l'ensemble des résultats avec chacun des classificateurs et les meilleurs hyper paramètres trouvé, comparé avec les paramètres par défaut :

	Adadelta	Defaut	SGD	Defaut
Loss	0,00919	0,057	0,0943	0,4034
Accuracy	0,9742	0,9841	0,974	0,885
Validation Loss	0,081	0,0756	0,085	0,2502
Validation Accuracy	0,977	0,98	0,946	0,9295
Test Score	0,0772	0,08226	0,0839	0,2513
Test Accuracy	**0,9777**	0,9796	0,9455	**0,9226**

La différence entre le gain d'accuracy pour les hyper paramètres optimiser et ceux par défaut pour Adadelta est minime. On voit un gain plus grand pour SGD, qui voit une différence de plus de 2% en comparaison aux paramètres par défaut. Il ne réussit toutefois pas à atteindre les résultats d'Adadelta. Il est possible qu'en testant plus de paramètres et avec des mesures plus précises nous réussissions à battre Adadelta avec SGD. SGD semble être le classificateur qui a le plus à gagner à des modifications infimes de ses hyper paramètres, étant plus sensible qu'Adadelta à ceux-ci.

Cifar10

Nous avons appliqué la même procédure à cifar10, quoiqu'avec seulement 10% de l'ensemble pour réduire le temps de calcul. Pour les hyper paramètres(batch, filtre et epoch) nous arrivons aux mêmes conclusions que pour Mnist.

Résultat pour nombre de batch (100% des données utilisées):

Batch Size	32	64	128	256
Nombre d'epoch	10	10	10	10
Nombre de filtres	128	128	128	128
Validation Loss	2,0476	2,0917	2,174	2,222
Validation Accuracy	0,2133	0,2	0,185	0,1583
Test Score	2,0456	2,0841	2,1428	2,2048
Test Accuracy	0,2173	**0,2046**	**1928**	**0,1666**

Résultat pour nombre de filtres (10% des données utilisées):

Batch Size	32	32	32	32
Nombre d'epoch	10	10	10	0
Nombre de filtres	128	256	512	1024
Validation Loss	2,0476	1,9285	1,8083	1,7921
Validation Accuracy	0,2133	3	0,3267	0,3433
Test Score	2,0456	1,9304	1,8561	1,8503
Test Accuracy	**0,2173**	**0,2781**	**0,3255**	0,3268

Résultat pour nombre d'epoch (10% des données utilisées):

Batch Size	32	32	32	32
Nombre d'epoch	10	20	30	40
Nombre de filtre	128	128	128	128
Validation Loss	2,0476	1,9069	1,8562	1,7665
Validation Accuracy	0,2133	0,2683	0,34	0,3367
Test Score	2,0456	1,9525	1,8764	1,8126
Test Accuracy	**0,2173**	**0,2719**	**0,3127**	0,3244

Le nombre de batch donne un ici encore un meilleur résultat, lorsque plus petit, bien que la différence entre les résultats d'accuracy soit plus grand. Cela est probablement dû à la petite taille de nos données et le fait que l'apprentissage sur Cifar10 soit plus difficile que Mnist. Cela explique aussi les plus grandes différences d'accuracy entre les résultats pour le nombre de filtres et d'epoch. Avec Mnist, nous atteignions rapidement une accuracy très haute, alors qu'un algorithme appliqué à Cifar10 a plus à gagner si on le laisse apprendre plus des données d'entrainement. Mais tout comme pour Mnist, on voit peu de différence dans les résultats une fois passer un certain seuil de filtres, soit entre 512 et 1024, et d'epoch, soit entre 30 et 40.

Hyper paramètres de classificateur pour Cifar10

Learning rate :

Adadelta	Lr=0.01	Lr=0.1	Lr=1	Lr=1.5
Loss	2,291	2,0248	1,9743	1,9701
Accuracy	0,1215	0,2324	0,2502	0,242
Validation Loss	2,2748	1,9291	1,9734	1,9452
Validation Accuracy	0,2	0,3233	0,245	0,2917
Test Score	2,2757	1,9482	1,9767	1,557
Test Accuracy	**0,2072**	0,2876	**0,2534**	**0,279**

SGD	Lr=0.001	Lr=0.01	Lr=0.1	Lr=1.0
Loss	1,9215	1,8645	14,4526	2,3103
Accuracy	0,2752	0,2963	0,1033	0,0974
Validation Loss	1,8473	1,8562	14,4794	2,3027
Validation Accuracy	0,3233	0,34	0,1017	0,1167
Test Score	1,8706	1,8764	145062	2,306
Test Accuracy	**0,3056**	0,3127	**0,1**	**0,1**

Pour les deux classificateurs, nous obtenons le même learning rate optimale que pour Mnist. On voit encore l'accuracy du SGD descendre à 10% avec un learning rate de 0.1, suggérant un cas d'underfitting. Un learning rate trop élevé ne permet pas à SGD de faire assez de progrès pour bien apprendre. On voit toutefois que SGD est meilleure qu'Adadelta lorsqu'appliquée à Cifar10. Peut-être que la façon de procéder par exponential decay d'Adadelta lui nuit pour des ensembles de données plus complexes. Cette différence apparait un peu plus flagrante si les deux classificateurs sont appliqués à l'ensemble des données de Cifar10 :

	Adadelta	Ada défaut	SGD	SGD défaut
Loss	1,6046	1,6439	1,6918	1,5802
Accuracy	0,4334	0,4183	0,3988	0,4345
Validation Loss	1,6996	1,6926	1,666	1,5617
Validation Accuracy	0,4032	0,414	0,4058	0,4496
Test Score	1,6928	1,6772	1,6659	1,5499
Test Accuracy	**0,4118**	0,4149	**0,4105**	0,4586

On voit une différence de près de 4% entre l'accuracy d'Adadelta et SGD. Pour les deux classificateurs, les paramètres par défaut on permit d'obtenir les meilleurs résultats. Ceux-ci sont probablement mieux

balancés que nos hyper paramètres, que nous avons testés de façon un peu indépendante les unes des autres.

Réseau de neurones à convolution

Pour nos tests sur un réseau à convolution, nous avons procédé de la même façon que pour le multilayer perceptron. Notre code se base sur celui provenant du github de keras[2], que nous avons modifié pour avoir plus de layers. Plutôt que de faire un pooling pour chaque 2 convolutions telles que le modèle vu en classe, nous le faisons après chaque convolution. Le nombre de filtres est aussi statique pour chaque convolution, plutôt que d'aller en incrémentant, car nos résultats étaient sensiblement meilleurs de cette façon (gain de 0.5 à 1% d'accuracy).

Nous avons exécuté les mêmes types de tests sur les hyper paramètres que pour le mutilayer perceptron. Toutefois, à cause de la lenteur de l'exécution du code et la limitation des accès aux ordinateurs de l'école, nous n'avons pu faire de comparatif sur l'ensemble des données entre nos deux classificateurs. Comme pour le multilayer perceptron, nous avons testé le nombre de batch, de filtres et d'epoch avec Adadelta pour Mnist et SGD pour Cifar10.

Mnist

L'accuracy obtenu avec le multilayer perceptron pour Mnist était déjà très élevé, atteignant 98% lors de nos tests avec 1024 filtres. Il était peu probable que le réseau à convolution démontre des résultats pouvant faire beaucoup mieux, et comme de fait, le maximum atteint a été de 99,3% . Nous allons présenter les résultats obtenus pour les hyper paramètres de batch, filtre et epoch, puis pour les classificateurs.

Hyper paramètres de batch, epoch et filtre pour Mnist

Résultat pour différent nombre de batch (10% données utilisées):

Batch Size	32	64	128	256
Nombre d'epoch	12	12	12	12
Nombre de filtre	32	32	32	32
Validation Loss	0,1791	0,1808	0,1894	0,415
Validation Accuracy	0,9533	0,95	0,945	0,8867
Test Score	0,1697	1776	0,2004	0,4544
Test Accuracy	0,9477	**0,949**	**0,9417**	**0,8802**

Les résultats pour le nombre de batch n'ont rien donné d'inattendu, avec le plus petit nombre donnant la meilleure accuracy. Toutefois, la différence entre 32 et 128 étant relativement petite, nous avons

[2] https://github.com/fchollet/keras/blob/master/examples/mnist_cnn.py

utilisé des batch de 128 lors de nos tests sur l'ensemble des données de Mnist pour accélérer le temps de calcul.

Résultat pour différent nombre de filtres (10% données utilisées):

Batch Size	32	32	32	32
Nombre d'epoch	12	12	12	12
Nombre de filtres	32	64	128	256
Validation Loss	0,0667	0,1087	0,1269	0,0732
Validation Accuracy	0,09828	0,9633	0,9667	0,975
Test Score	0,0525	0,09963	0,08926	0,06376
Test Accuracy	0,984	**0,9704**	**0,9745**	**0,9813**

Les résultats des tests sur les filtres sont un peu étonnants, on se serait attendu à de meilleurs résultats en augmentant le nombre de filtres. Il est possible que cela soit l'effet du hasard, car il semble que l'accuracy recommence à augmenter avec 256 filtres. Il est aussi possible que l'apprentissage ait déjà plafonné autour de 97-98% et que l'augmentation du nombre de filtres ne donnera jamais de résultat beaucoup mieux.

Résultat pour différents epoch et filtres (100% des données) :

Batch Size	128	128	128	128
Nombre d'epoch	12	12	24	24
Nombre de filtre	32	64	32	64
Validation Loss	0,0667	0,0452	0,0532	0,0409
Validation Accuracy	0,09828	0,9898	0,9878	0,9905
Test Score	0,0525	0,0323	0,038922	0,02616
Test Accuracy	**0,984**	**0,9908**	**0,9887**	0,9933

Comme on a pu le constater, les résultats avec Mnist sont déjà très bons. Pour nos derniers tests sur ces hyper paramètres, nous avons essayé avec différente epoch et filtre. Nous avons réussi à atteindre un accuracy de 99.3% avec un epoch de 24 et 64 filtres.

Hyper paramètres pour Adadelta

Nous avons testé différent learning rate pour Adadelta. Malheureusement pour ce test, nous n'avons pas utilisé les meilleurs hyper paramètres obtenus pour le nombre de batch, filtres et epoch car l'algorithme aurait pris trop de temps à rouler et les résultats de Mnist étaient déjà très bons. Nous

avons surtout cherché à voir si nous pouvions améliorer notre classificateur.

Résultat pour le learning rate (100% des données utilisées):

AdaDelta	**Lr=0,01**	**Lr=0.05**	**Lr=0.1**	**Lr=1,0**
Validation Loss	0,133	0,1376	0,1395	0,1619
Validation Accuracy	0,9553	0,9533	0,9533	0,9467
Test Score	0,0992	0,1014	0,1144	0,1397
Test Accuracy	0,9681	**0,9677**	**0,9638**	**0,9553**

Nous nous serions attendus à ce que le learning rate de 0.1 performe le mieux, tel qu'avec le multilayer perceptron. Ce ne fut toutefois pas le cas et un pas plus petit de 0.01 a donné les meilleurs résultats. La différence entre l'accuracy de lr(0.1) et lr(0.01) n'est que de 0.0043, il est probable que de faire rouler l'algorithme de nouveau sans utilisé le random seed de numpy nous donnerait des résultats différents. La différence est trop petite pour que nous puissions véritablement dire que l'un est meilleur que l'autre, en ce qui a trait à l'accuracy. Lr(0.1) est possiblement plus intéressant pour le maximisé le temps d'exécution.

Cifar10

Pour Cifar10, nous avons utilisé SGD avec un learning rate de 0.01 plutôt que 0.1 puisque nos résultats avec le multilayer percetron semblait indiqué une meilleure performance avec ce taux.

Hyper paramètres de batch, epoch et filtre pour Mnist

Les résultats pour l'augmentation du nombre de filtres et d'epoch montraient encore une fois une amélioration de l'accuracy. Le test qui fut réellement intéressant fut sur le nombre de batch.

Résultat pour différent nombre de batch (100% des données utilisées) :

Batch Size	32	100	1000
Nombre d'epoch	12	12	12
Nombre de filtre	32	32	32
Validation Loss	2,1296	1,5572	1,3433
Validation Accuracy	0,2299	0,4317	0,52
Test Score	2,1374	1,5902	1,3495
Test Accuracy	**0,2299**	**0,4158**	0,5127

On voit une amélioration très significative entre l'utilisation de 32 batch et de 1000. L'accuracy plus que double entre les deux tests. Il est possible que cela soit dû à la nature de Cifar10 et de notre classificateur SGD. SGD est un classificateur qui, en théorie du moins, devrait performer mieux avec plus de batch. Toutefois, jusqu'à présent un nombre de batch plus petit (32) a toujours performé mieux qu'un nombre

élevé. Pour Mnist, cela s'expliquait en partie par l'atteinte d'accuracy déjà très élevé qui ne laissait pas beaucoup de place à l'amélioration. Il était plus difficile d'expliqué les raisons d'un accuracy moins bon pour des batch élevé lors de l'entrainement sur Cifar10 avec le multilayer perception. La majeure différence entre les deux tests est le nombre de données. Nous n'avons utilisé que 10% des données pour le multilayer perceptron alors qu'ici nous avons roulé sur l'ensemble de Cifar10.

Nous avons choisi de rouler un test de plus qu'avec le multilayer perceptron, soit l'inclusion de data augmentation en temps réel. Cela veut dire qu'à chaque batch, une transformation est appliquée aux images. Il existe énormément de types de transformation, nous avons pris ceux que suggéraient keras, soit la rotation, la translation horizontale et verticale, et le flip horizontal. Toutes ses transformations sont appliquées de façon aléatoire à chaque batch. Considérant la nature de Cifar10, nous nous attendions à une amélioration notable de l'accuracy, ce qui n'a pas été nécessairement le cas. Nous avons testé avec Adadelta et SGD.

Résultat de l'ajout de data augmentation (100% des données utilisées) :

Batch Size	32	32	32	32
Nombre d'epoch	12	12	12	12
Nombre de filtres	32	32	32	32
Data augmentation	Oui	Oui	Non	Non
Classifier	SGD	Adadelta	SGD	Adadelta
Validation Loss	0,7308	0,7602	0,7006	0,6543
Validation Accuracy	0,7512	0,7447	0,76	0,7774
Test Score	0,73079	0,649	0,73225	0,6939
Test Accuracy	0,7488	0,7747	0,7472	0,7608

Pour SGD, la différence entre l'accuracy avant et après l'utilisation des transformations est négligeable. Adadelta voit une augmentation d'environ 1%, ce qui n'est pas si mal, mais pas non plus si intéressante. Il est possible qu'en testant avec différent data augmentation, nous réussissions à obtenir un plus grand gain, mais cela n'est pas garantie à la lumière des résultats obtenus.

Hyper paramètres pour SGD

Nous avons testé différent learning rate pour SGD. Nos résultats semblent démontrer beaucoup d'underfitting.

Résultat pour learning rate (10% des données utilisées) :

SGD	Lr=0,001	Lr=0.01	Lr=0.1	Lr=0.011	Lr=0.02
Loss	1,9913	2,3021	2,3101	2,303	2,3037
Accuracy	0,2593	0,1069	0,0989	0,961	0,967
Validation Loss	1,9568	2,3035	2,3029	2,3039	2,3032
Validation Accuracy	0,2817	0,085	0,1017	0,085	0,105
Test Score	1,9582	2,3032	2,3042	2,3035	2,3035
Test Accuracy	0,2822	0,1	0,1	0,1	0,1

Plus encore qu'avec le multilayer perceptron, SGD se montre très sensible au changement de taux d'apprentissage. Il faut descendre à un learning rate de 0.001 pour obtenir un taux un tant soit peu valable. Il est clair que SGD demande beaucoup plus d'ajustement pour obtenir de bon résultat, en comparaison à Adadelta.

Conclusion sur multilayer perceptron et réseau à convolution

Pour ce qui est de Mnist, si nous avions à choisir un réseau, nous irions avec le multilayer perceptron. Les deux types de réseaux ont très bien performé sur Mnist, obtenant constamment des résultats au-delà de 95%. La majeure différence se retrouve dans le temps d'exécution du code. Rouler l'ensemble des données sur Mnist avec le réseau à convolution prenait entre 45 et 1h sur les ordinateurs de l'école. Le multilayer perceptron en comparaison pouvait produire des résultats à l'intérieur d'une minute. Le temps le plus haut obtenu fut de 27 minutes lors de l'utilisation de 2048 filtres. Tout dépendant de la précision nécessaire et à moins d'avoir besoin d'une accuracy dépassant 99%, notre réseau à convolution n'était pas le réseau le plus approprié pour bien classifier les données de Mnist.

Cela change complètement lorsqu'il est question de Cifar10. Le multilayer perceptron peine à franchir la barre des 45% pour cet ensemble, alors que le réseau à convolution atteint un incroyable 77,4% avec Adadelta. Le réseau à convolution réussi beaucoup mieux à extraire les caractéristiques discriminantes de chaque classe de Cifar10 que le multilayer perceptron.