

Apprentissage automatique

Implémentation de classifieurs

KNN, SVM, Softmax et réseaux neuronaux

Damien DOUTEAUX

Avril
25
2017



Table des matières

■ 1 • Introduction	4
■ 2 • Présentation de la base de données	5
1.1 Présentation de la base de données.	5
1.2 Chargement de la base données	5
1.2.1 Téléchargement de la base de données	5
1.2.2 Chargement en mémoire des données.	6
1.3 Visualisation des données	6
1.3.1 Vérification des dimensions	6
1.3.2 Vérification visuelle	7
■ 3 • Implémentation d'un algorithme KNN	8
2.1 Préparation des données.	8
2.2 Version du classifieur avec deux boucles	8
2.2.1 Algorithme utilisé.	8
2.2.2 Test de l'implémentation	9
2.3 Version du classifieur avec une boucle	10
2.3.1 Algorithme utilisé.	10
2.3.2 Test de l'implémentation	10
2.4 Version du classifieur sans boucle	10
2.4.1 Algorithme utilisé.	10
2.4.2 Test de l'implémentation	11
2.5 Résumé des versions	12
2.6 Mise en place de la cross-validation	12
2.6.1 Découpage du jeu de données.	12
2.6.2 Codage de la cross-validation	13
2.6.3 Déterminer la meilleure valeur de k	14
2.6.4 Test du meilleur modèle	15
■ 4 • Entraînement d'un SVM	16
3.1 Préparation des données.	16
3.1.1 Construction des ensembles d'apprentissage et de test	16
3.1.2 Pré-traitement des images	16
3.2 Calcul du gradient pour le SVM	17
3.2.1 Première estimation de la perte	17
3.2.2 Expression théorique du gradient	18
3.2.3 Implémentation naïve du gradient	18
3.2.4 Gradient et régularisation.	19
3.2.5 Tests du calcul du gradient	19
3.3 Version vectorielle du code	20
3.3.1 Calcul vectoriel du loss	20
3.3.2 Tests du calcul vectoriel du loss	22
3.3.3 Calcul vectoriel du gradient	22
3.3.4 Tests du calcul vectoriel du gradient	22
3.4 Descente stochastique du gradient.	23
3.4.1 Sélection aléatoire d'un jeu de données.	23
3.4.2 Mise à jour des poids	23
3.4.3 Résultats de la descente stochastique.	24
3.4.4 Évaluation du modèle	24

3.5	Optimisation des hyper-paramètres	25
3.5.1	Les hyper-paramètres	25
3.5.2	Recherche du meilleur couple de valeurs	25
3.5.3	Évaluation du meilleur modèle	28
3.6	Visualisation des poids	28
4	• Un classificateur softmax	30
4.1	Préparation des données	30
4.2	Calcul du gradient pour un classifieur Softmax	30
4.2.1	Perte calculée par le Softmax	30
4.2.2	Calcul théorique du gradient	31
4.2.3	Implémentation du gradient	32
4.2.4	Test du gradient	33
4.3	Vectorisation des calculs	34
4.3.1	Vectorisation du calcul du <i>loss</i>	34
4.3.2	Vectorisation du calcul du gradient	35
4.3.3	Comparer les implémentations	35
4.4	Descente stochastique du gradient pour le softmax	36
4.4.1	Entraînement du modèle	36
4.4.2	Entraînement du modèle	36
4.5	Optimisation des hyper-paramètres	36
4.5.1	Code pour réaliser l'optimisation	36
4.5.2	Résultats obtenus	37
4.5.3	Test du modèle optimal	37
4.6	Visualisation des poids du softmax	38
5	• Réseau neuronal à deux niveaux	39
5.1	Création d'un réseau jouet	39
5.1.1	Réseau jouet	39
5.1.2	Jeu de données pour le réseau	39
5.1.3	Structure obtenue	39
5.2	Passe forward	41
5.2.1	Calcul des scores	41
5.2.2	Calcul de la perte	42
5.3	Passe backward	43
5.3.1	Principe de la rétropropagation du gradient	43
5.3.2	Vectoriser les calculs	45
5.3.3	Implémentation de la rétropropagation	45
5.3.4	Test de l'implémentation	46
5.4	Entraînement du réseau	46
5.4.1	Descente stochastique du gradient	47
5.4.2	Test du réseau	47
5.5	Vraies données et vrai réseau	48
5.5.1	Chargement des données	48
5.5.2	Entraînement du réseau	49
5.5.3	Évaluer les problèmes	49
5.6	Optimisation des hyperparamètres	50
5.6.1	Les hyperparamètres	51
5.6.2	Influence de <i>num_iters</i>	52
5.6.3	Influence de <i>batch_size</i>	52
5.6.4	Influence de <i>learning_rate</i>	53
5.6.5	Influence de <i>learning_rate_decay</i>	54
5.6.6	Influence de <i>reg</i>	56

5.6.7	Influence de <i>hidden_size</i>	58
5.6.8	Choix d'un compromis	60
Conclusion	64

Liste des figures

1	Exemples d'images pour différents labels de CIFAR-10	7
2	Résultats obtenus en cross-validation avec KNN	15
3	Image moyenne calculée sur toutes les images de l'ensemble d'apprentissage	16
4	Illustration de l'effet de soustraire l'image moyenne en pré-traitement du SVM	17
5	Évolution de la perte au long des différentes étapes de l'apprentissage	24
6	Précisions obtenues pour les différentes combinaisons d'hyperparamètres testées en appren-	
	tissage et en test	27
7	Visualisation des poids appris par le meilleur modèle SVM entraîné	28
8	Visualisation des poids appris par le meilleur modèle softmax entraîné	38
9	Structure du réseau de neurones pour le problème jouet	40
10	Évolution de la perte au cours de l'entraînement de notre réseau	48
11	Évolution de la perte et des précisions sur les ensembles d'apprentissage et de test en fonction des itérations	50
12	Visualisation des poids appris par le réseau	51
13	Évolutions de la perte et des précisions en classification au cours de différents entraîne-	
	ments en faisant varier <i>num_iters</i>	52
14	Évolution de la précision en validation pour différentes valeurs de <i>learning_rate</i>	54
15	Évolutions de la perte et des précisions en classification au cours de différents entraîne-	
	ments en faisant varier <i>learning_rate</i>	55
16	Meilleurs poids obtenus pendant les essais de valeurs sur <i>learning_rate</i>	56
17	Évolutions de la perte et des précisions en classification au cours de l'entraînement pour <i>learning_rate_decay=0.9</i>	57
18	Évolution de la précision en validation pour le paramètre <i>reg</i> sur des puissances de 10	57
19	Évolutions de la perte et des précisions en classification au cours de différents entraîne-	
	ments en faisant varier <i>reg</i>	58
20	Évolution des poids appris pour différentes valeurs de <i>reg</i>	59
21	Évolution de la précision pour différentes valeurs de <i>hidden_size</i>	60
22	Évolution des poids appris pour différentes valeurs de <i>hidden_state</i>	61
23	Résultats obtenus pour le premier modèle optimisé	62
24	Visualisation des poids obtenus pour un modèle optimisé similaire à celui de l'énoncé	63
25	Résultats obtenus pour le premier modèle optimisé avec 4000 itérations	63

Liste des tables

1	Résumé des performances des différentes implémentations de KNN	12
---	--	----

Introduction

Ce rapport présente le code développé ainsi que les résultats obtenus pour le BE d'Apprentissage Automatique relatif à l'implémentation de classifieurs. Nous traiterons dans ce rapport l'intégralité du sujet à l'exception de la dernière partie sur les descripteurs. Pour chaque partie, le code réalisé sera présenté en précision, ainsi que les motivations théoriques et les résultats obtenus.

Ce travail s'articule autours de la présentation de différents classifieurs dans l'optique final d'implémenter un réseau de neurones. Ainsi, le rapport est structuré en cinq parties. Tout d'abord, nous nous intéresserons à la présentation de la base de données utilisée (CIFAR10), son chargement et sa prévisualisation.

Dans un second temps, nous nous intéresserons à notre premier classifieur, qui sera un KNN. Cette implémentation sera l'occasion de voir comment optimiser les performances du code en passant d'une écriture itérative à une écriture entièrement vectorisée. Ce sera également une première occasion pour nous de voir la méthode pour optimiser les hyper-paramètres d'un modèle.

Nous nous intéresserons ensuite à un second type de classifieur, les SVM. La méthode de travail sera la même que pour le KNN, à savoir une écriture itérative dans un premier temps que nous vectoriserons par la suite. Pour ce classifieur, nous aurons l'occasion de tester la réalisation d'une descente stochastique de gradient, qui sera réutilisée pour la suite du travail. Cette partie nous permettra enfin de tester la recherche de valeurs optimales pour un couple d'hyper-paramètre. Encore plus que pour le KNN, nous insisterons sur la nécessité d'une écriture vectoriel pour ce classifieur.

Le troisième classifieur que nous aborderons sera (comme nous le verrons), une variation du SVM appelée Softmax. Le fonctionnement de ce dernier sera similaire à celui du SVM, si ce n'est une fonction de coût qui diffère et fournit ainsi des résultats différents. Les travaux sur ce dernier seront ainsi fortement similaires à ceux sur le SVM. L'intérêt de ce nouveau classifieur résidera en partie dans son utilisation (de sa fonction de perte) pour le réseau de neurones.

Enfin, dans une dernière partie nous nous intéresserons à l'implémentation de notre réseau de neurones, en réutilisant quelques éléments vus précédemment. Pour l'implémenter, nous commencerons par une situation de test permettant de réaliser les différents éléments et tester leur bon fonctionnement. Dans un second temps, nous utiliserons ce réseau sur nos vraies données afin de tester ses performances.

Une dernière partie sur l'utilisation de représentation haut niveau des images comme les diagrammes HOG devait être abordée. Cependant, nous n'avons pu le faire pour deux raisons. En effet, je n'ai jamais réussi à accomplir totalement le chargement des données et leur pré-traitement sur mon poste (au bout de 25 minutes, pas de réponse et un ordinateur qui ne répond presque plus). Étant également en partie en retard pour le rendu de ce rapport et ayant travaillé seul, je ne me suis donc pas concentré sur la résolution de ce bug (diminuer la quantité de données à utiliser par exemple) et m'en excuse. Cette dernière partie est donc manquante, vous pourrez cependant trouver le code à exécuter dans l'archive, dans la manière où ce dernier ressemblait à des éléments déjà réalisés. Je n'ai cependant pas ses résultats...

En cas de question sur ce rapport ou le contenu du code qui vous a été fourni, n'hésitez pas à me contacter à l'adresse mail suivante :

damien.douteaux@ecl13.ec-lyon.fr

1 • Présentation de la base de données

Dans un premier temps, nous allons présenter succinctement la base de données utilisée, et la démarche réalisée pour la charger avec le code Matlab.

1.1 | Présentation de la base de données

Pour ce travail, nous allons utiliser la base de données CIFAR-10. Cette dernière est proposée par l'université de Toronto, et est une version « partielle » de la base de données d'images CIFAR-100.

Au total, la base contient 60000 images couleurs de taille 32×32 , réparties en dix classes (que nous montrerons un peu plus loin). Ces différentes images sont réparties comme suit :

- **Base d'apprentissage** Elle est constituée de 50000 images elles-mêmes réparties en groupes de 10000 images.
- **Base de test** Elle est constitué d'un groupe de 10000 images.

Une remarque importante sur ces données est que les classes sont strictement deux à deux disjointes. Ainsi, il n'existe pas d'image dans la base de données associées à deux labels différents.

1.2 | Chargement de la base données

Maintenant que nous avons une idée de cette base de données, nous pouvons la charger avec le code Matlab proposé. Dans un premier temps, nous allons illustrer le téléchargement de la base de données et regarder les différents objets créés. Dans un second temps, nous nous intéresserons à sa mise en mémoire pour utilisation par notre code (cette étape sera préalable à toutes les sections précédentes mais ne sera détaillée qu'ici).

1.2.1 Téléchargement de la base de données

CIFAR-10 étant disponible en ligne, la fonction de téléchargement (`get_datasets`) va « simplement » télécharger l'archive à l'URL spécifiée et la décompresser sur notre disque dur. Le code utilisé est donc :

```
1 disp('Downloading Cifar-10 dataset.....');
2 url = 'http://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';
3 untar(url);
```

À l'issue de cette étape, nous obtenons donc un répertoire (`cifar-10-batches-mat`) contenant les éléments suivants :

- **batches.meta.mat** Cette matrice contient les différents labels utilisés dans CIFAR-10.
- **data_batch_1.mat à data_batch_5.mat** Il s'agit des cinq groupes d'images d'apprentissage, chacun contient 10000 éléments répartis comme suit :
 - ▷ **data** Les données (ie. les images).
 - ▷ **labels** Les labels associés aux images
 - ▷ **batch_label** Le nom du jeu de données.
- **readme.html** Ce fichier reprend les informations du site CIFAT-10, on y a retrouvé en particulier les informations présentées à la Section 1.1.
- **test_batch.mat** Cette structure contient l'ensemble de test. Sa structure est similaire à celle des ensembles d'apprentissage.

1.2.2 Chargement en mémoire des données

Le code proposé pour le chargement de ces données est découpé en quatre étapes.

Vérification de la présence de la base de données Dans un premier temps, nous vérifions que la base de données a bien été chargée, si ce n'est pas le cas, on relance un téléchargement de la base en ligne.

Chargement des ensembles d'apprentissage Comme nous l'avons vu, l'ensemble d'apprentissage est découpé en cinq parties. Ces cinq parties sont chargées tour à tour et concaténées dans une seule et même matrice.

Chargement de l'ensemble de test On procède de même avec l'ensemble de test, qui est lui composé d'un seul groupe de données.

Chargement des meta données Pour terminer, on charge les noms des différents labels des images qui sont contenus dans la structure `batches.meta.mat`.

Résultat final Une fois les différents éléments chargés, on enregistre ces derniers au sein d'une structure nommée `imdb`, qui sera réutilisée dans la suite de ce BE pour manipuler les images. La structure de cet objet est la suivante :

- `train_data` L'ensemble de données d'apprentissage.
- `train_labels` Les labels associés aux données d'apprentissage.
- `test_data` L'ensemble des données de test.
- `test_labels` Les labels associés aux données de test.
- `class_names` Les noms des classes des labels. En effet, les ensembles `test_labels` et `train_labels` ne contiennent que des valeurs numériques comprises entre 1 et 10, cette dernière matrice fournit donc la correspondance avec les labels textuels associés à ces valeurs.

1.3 | Visualisation des données

Désormais, nous disposons de nos données chargées en mémoire, nous pouvons les observer un peu plus pour connaître leur aspect.

1.3.1 Vérification des dimensions

Dans un premier temps, nous commençons par afficher les dimensions des différents éléments de `imdb`. Nous obtenons alors le résultat suivant :

```

1 Training data shape:
2      50000           32           32           3
3
4 Training labels shape:
5      50000            1
6
7 Test data shape:
8      10000           32           32           3
9
10 Test labels shape:
11     10000            1

```

Nous retrouvons bien ici les valeurs avancées à la Section 1.1, à savoir :

- **Ensemble d'apprentissage** On retrouve bien :
 - ▷ **Les images** 50000 images, de dimensions 32×32 codée en RGB (donc trois niveaux de couleurs, d'où le 3 en fin de ligne).
 - ▷ **Les labels** On observe bien 50000 labels.

- *Ensemble de test* De même que pour l'ensemble d'apprentissage, mais avec uniquement 10000 images.

1.3.2 Vérification visuelle

Nous savons désormais que les dimensions sont cohérentes avec la quantité théorique de données. Nous allons alors visualiser des exemples d'images. Une possibilité vous est proposée à la Figure 1.

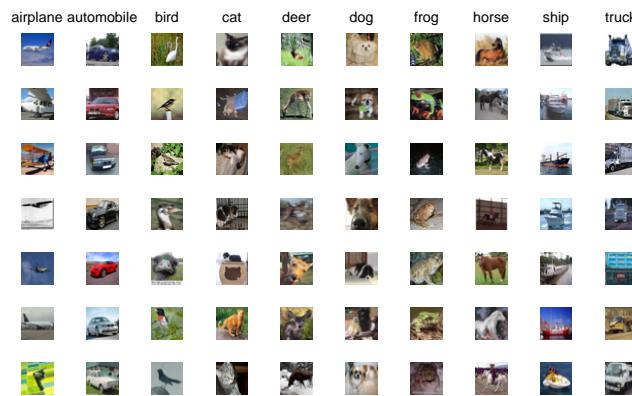


FIGURE 1 • Exemples d'images pour différents labels de CIFAR-10

On peut alors observer les dix labels utilisés dans CIFAR-10 :

Airplane	Automobile	Bird	Cat	Deer	Dog
Frog	Horse	Ship	Truck		

2 • Implémentation d'un algorithme KNN

Dans cette section, nous allons vous présenter le code réalisé ainsi que les différents résultats dans l'implémentation d'un algorithme de K-plus proches voisins (KNN, K-Nearest Neighbor) en Matlab.

2.1 | Préparation des données

Nous avons vu lors de la Section 1 le détail de la base de données. Dans la mesure où les calculs avec KNN peuvent être un peu long, nous allons commencer par ré-échantillonner les données afin d'en conserver uniquement un sous-ensemble représentatif et permettant des calculs plus rapides.

En utilisant le code proposé, on obtient alors bien les dimensions qui nous étaient fournies dans l'énoncé, à savoir :

```

1 Training data shape:
2      5000          3072
3
4 Training labels shape:
5      5000           1
6
7 Test data shape:
8      500          3072
9
10 Test labels shape:
11     500           1

```

Quelques remarques sur cette préparation des données :

- **Quantité de données** Suite à cette étape, nous n'avons conservé que 10% des données d'apprentissages (5000 images) et 5% des données de test (500 images). Il est remarquable de voir que nous avons changé le ratio entre apprentissages et test.
- **Aspect des données** Ces dernières étaient présentées sous la forme de matrices. Désormais, ces dernières sont présente sous la forme de tableaux. Ainsi, à chaque indice d'un des tableaux `train_data` ou `test_data` correspond un tableau de $3072 = 32 \times 32 \times 3$. Ainsi, pour chaque données, nous avons mis bout à bout tous les pixels des images, chaque pixel étant lui même caractérisé par une valeur par niveau de couleur (RGB).

2.2 | Version du classifieur avec deux boucles

2.2.1 Algorithme utilisé

En préambule, nous entraînons le classifieur KNN. Pour mémoire, ceci revient uniquement à retenir les données pour la suite et est réalisé avec le code suivant :

```
1 model = knn_train(imdb.train_data, imdb.train_labels);
```

Nous pouvons alors entamer la partie calculatoire de l'algorithme, à savoir les calculs des distances entre les images tests et celles de l'ensemble d'apprentissage.

Une première méthode pour réaliser cette opération est d'utiliser un algorithme utilisant deux boucles. Ainsi, cette première version va parcourir toutes les images de l'ensemble de test et pour chaque image va à nouveau parcourir toutes les images de l'ensemble d'apprentissage et calculer la norme deux entre ces deux images. Le calcul de cette norme utilise la relation (1).

$$\text{dist}(I_1, I_2) = \sqrt{\sum_{i=1}^{\text{nb pixels}} (I_1[i] - I_2[i])^2} \quad (1)$$

Nous avons alors l'implémentation suivante pour cette méthode, qui est celle correspondant au fichier `knn_compute_distance_two_loops`.

```

1 for i = 1:num_test
2   for j = 1:num_train
3     dists(i,j) = sqrt(sum((X(i,:) - model.X_train(j,:)).^2));
4   end
5 end

```

2.2.2 Test de l'implémentation

Calcul des distances On remarque que la complexité de cette algorithme est en $O(|I_1| \times |I_2|)$, ce qui passera difficilement à l'échelle.

Nous pouvons cependant tester notre algorithme en utilisant le code proposé :

```

1 %Test your implementation:
2 dists_two = knn_compute_distances_two_loops(model, imdb.test_data);
3 disp(size(dists_two))

```

Le résultat est bien celui attendu, à savoir :

```

1 500 5000

```

Ce résultat correspond à la dimension de la matrice de prédiction. On obtient ainsi une matrice avec 500 lignes (ie. 500 images de test) et qui dont les distances ont été calculées avec les 5000 images d'apprentissage (d'où les 5000 colonnes).

Prédiction des labels pour $k = 1$ Il nous reste alors à prédire les labels en utilisant ces distances et comparer ces dernières avec les labels réels. Ceci est réalisé en utilisant le code suivant, proposé dans le sujet.

```

1 % We use k = 1 (which is Nearest Neighbor).
2 k = 1;
3 test_labels_pred = knn_predict_labels(model, dists_two, k);
4 num_correct = sum(sum(test_labels_pred == imdb.test_labels));
5 num_test = length(imdb.test_labels);
6 accuracy = double(num_correct)/num_test;
7 fprintf('Got %d / %d correct => accuracy: %f\n', num_correct, num_test, accuracy);
8 fprintf('You should expect to see approximately 27% accuracy\n');

```

Le fait de choisir $k = 1$ impose que l'on va donner à une image le label de son voisin le plus proche. Nous obtenons alors le rendu suivant, qui est celui attendu dans le sujet :

```

1 Got 137 / 500 correct => accuracy: 0.274000
2 You should expect to see approximately 27% accuracy.

```

Prédiction des labels pour $k = 5$ Pour conclure sur cette version avec deux boucles, nous allons relancer la prédiction des labels, mais en utilisant cette fois-ci les cinq plus proches voisins d'une image pour calculer son score et décider du label qu'on lui attribue.

Pour cela, il suffit de reprendre le code précédent en remplaçant $k = 1$ par $k = 5$. Nous obtenons alors le retour suivant :

```

1 Got 139 / 500 correct => accuracy: 0.278000
2 You should expect to see a slightly better performance than with k = 1.

```

On obtient bien le résultat attendu, à savoir une très légère augmentation de la précision, mais rien de bien déterminant étant donné les valeurs assez faibles obtenues.

Analyse des résultats D'un point de vue pratique, on observe deux éléments importants :

- **Temps de calcul** Ce dernier est vraiment important, de l'ordre de XXX minutes.

- **Précision** Le modèle entraîné n'est pas performant, avec une précision atteignant seulement 27,4%.

Ainsi, on dispose de deux axes sur lesquels il serait intéressant d'améliorer notre modèle. D'une part, la rapidité de calcul doit être améliorée, et nous allons détailler les moyens d'y parvenir par la suite.

D'autre part, la précision pourra difficilement être augmentée sans changer de modèle ou de données d'apprentissage/descripteurs, et ne sera donc à priori par appelée à beaucoup évoluer en restant avec KNN. Il sera cependant intéressant de voir s'il existe une valeur de k optimale pour laquelle cette dernière atteint un maximum.

2.3 | Version du classifieur avec une boucle

2.3.1 Algorithme utilisé

L'objectif est désormais d'améliorer la vitesse d'exécution pour KNN. Comme nous l'avons vu, la solution à deux boucle est d'une complexité médiocre, nous allons donc essayer de vectoriser cette partie.

Dans un premier temps, nous allons essayer de gagner une boucle sur les deux proposées. La solution la plus naturelle est alors de supprimer la boucle sur les données d'apprentissage et d'essayer de réaliser cette dernière en une seule opération matricielle.

Pour cela, nous avons fait le constat de calcul suivant :

On remarque ainsi que pour se ramener à une cas avec une seule boucle, il suffit de dupliquer un nombre de fois suffisant l'image de test et de réaliser la boucle supprimée directement par des opérations matricielles.

```

1 for i=1:num_test
2     dists(i,:) = sqrt(sum(transpose((model.X_train - repmat(X(i,:), num_train, 1))
3 .^2)));
end

```

2.3.2 Test de l'implémentation

Le code développé peut alors être lancé de manière similaire à celle de la version avec deux boucles, si ce n'est que l'on réalise désormais l'appel suivant :

```
1 dists_one = knn_compute_distances_one_loops(model, imbd.test_data);
```

Pour vérifier que nous n'avons pas fait d'erreur, nous vérifions alors le taux de différence avec la matrice de distances calculées dans le cas à deux boucles. Le test avec l'implémentation fournie est alors un succès, avec le retour suivant :

```

1 Difference was: 0.000000
2 Good! The distance matrices are the same

```

Les performances prédictives restent ici les mêmes (puisque la matrice de distances est la même...), cependant on remarque une amélioration significative du temps de calcul, qui passe alors à XXX.

2.4 | Version du classifieur sans boucle

2.4.1 Algorithme utilisé

Si le temps de calcul est amélioré avec la vectorisation partielle de la solution à une boucle, il est possible d'aller plus loin en implémentant une solution totalement vectorisée, ie. sans boucle.

Aspects théoriques Avant d'en fournir le code, nous allons cependant justifier d'un point de vue théorique l'origine des relations utilisées.

Dans le suite, nous noterons j l'indice de l'image d'apprentissage (ie. son indice dans la liste des images d'apprentissage) et i celui de l'image de test. Notre objectif est de calculer la distance entre l'image de test X_i et celle d'apprentissage $X_{app,j}$, à savoir :

$$\|X_i - X_{app,j}\|_2$$

Nous pouvons alors développer cette expression :

$$\begin{aligned}
 \|X_i - X_{app,j}\|_2 &= \sum_{k=1}^{\text{nb pixels}} (X_{i,k} - X_{app,j,k})^2 \\
 &= \sum_{k=1}^{\text{nb pixels}} (X_{i,k}^2 + X_{app,j,k}^2 - 2 \cdot X_{i,k} \cdot X_{app,j,k}) \\
 &= \underbrace{\left(\sum_{k=1}^{\text{nb pixels}} X_{i,k}^2 \right)}_{\text{Somme pixels de test}} + \underbrace{\left(\sum_{k=1}^{\text{nb pixels}} X_{app,j,k}^2 \right)}_{\text{Somme pixels apprentissage}} - 2 \cdot \underbrace{\left(\sum_{k=1}^{\text{nb pixels}} X_{i,k} \cdot X_{app,j,k} \right)}_{\text{Terme croisé}}
 \end{aligned}$$

Concernant les termes croisés, ce dernier peut se réécrire sous la forme :

$$2 \cdot \sum_{k=1}^{\text{nb pixels}} X_{i,k} \cdot X_{app,j,k} = 2 \cdot \sum_{k=1}^{\text{nb pixel}} X_{i,k} X_{app,k,j}^t$$

On remarque donc que ce terme correspond uniquement à une « cellule » d'un produit matriciel. On peut alors réécrire la formule pour la cellule $D_{i,j}$ de la matrice de distance sous la forme suivante :

$$D_{i,j} = X_{\text{somme(ligne } i\text{)}}^2 + X_{\text{app, somme(ligne } j\text{)}}^2 - 2 \cdot [XX_{\text{app}}]_{i,j} \quad (2)$$

Nous pouvons donc exprimer la matrice de distance via uniquement des opérations matricielles.

Implémentation Du point de vue de l'implémentation, nous avons voulu conserver un code lisible. Ainsi, nous pré-calculons les différents éléments de la matrice de distance séparément avant de tous les ajouter, afin de mieux mettre en avant la correspondance entre le code et la relation (2).

Ainsi, nous avons proposé le code suivant pour la version sans boucle :

```

1 function [ dists ] = knn_compute_distances_no_loops( model, X )
2     train_data = model.X_train;
3
4     X_square = repmat(sum(X.*X, 2), 1, size(train_data,1));
5     X_train_square = repmat(transpose(sum(train_data.*train_data, 2)), size(X,1), 1);
6     X_X_train = -2*X*train_data';
7
8     dists = sqrt(X_square + X_train_square + X_X_train);
9 end

```

On retrouve bien les trois termes de la relation (2).

2.4.2 Test de l'implémentation

Une fois le code réalisé, il nous faut le tester. Encore une fois, il suffit de changer la fonction appelée, à savoir utiliser la méthode `knn_compute_distances_no_loops`.

```
1 dists_no = knn_compute_distances_no_loops(model, imdb.test_data);
```

Pour tester l'implémentation, on procède comme pour la version à une seule boucle, c'est-à-dire en comparant la matrice de distance obtenue avec celle obtenue par la méthode basique de deux boucles. Les résultats sont ici aussi bon, avec des matrices identiques, comme le prouve le résultat suivant.

```

1 Difference was: 0.000000
2 Good! The distance matrices are the same

```

La matrice obtenue est ainsi la même, nous ne perdons donc pas de précision avec cette vectorisation complète, cependant le temps d'exécution est réellement meilleur, et devient presque « indolore ».

2.5 | Résumé des versions

Nous venons donc de voir trois versions de l'algorithme KNN allant de deux à aucune boucle pour calculer la matrice de distance. Pour conclure sur ces différents essais, un code est proposé pour chronométrier les différentes implémentations. Les résultats obtenus sont résumés (ainsi que ceux sur la précision) à la Table 1.

Version	Deux boucles	Une boucle	Pas de boucle
Temps d'exécution (s)	520,98	188,57	2,76

TABLE 1 • Résumé des performances des différentes implémentations de KNN

Les chiffres fournis dans cette Table 1 proviennent du code proposé en comparaison, donc les retours Matlab ont été les suivants :

```
1 Two loop version took 520.981504 seconds
2 One loop version took 188.569674 seconds
3 No loop version took 2.758244 seconds
```

On remarque effectivement que les temps décroissent fortement avec la vectorisation. On remarquera également que la machine sur laquelle les tests ont été effectués n'était pas des plus performante avec plus de 8 minutes pour la version non vectorisée (contre une minute dans le sujet...)!

Quoiqu'il en soit, il nous conserverons donc la version totalement vectorisée pour la suite, dans la mesure où elle est indubitablement la plus rapide.

2.6 | Mise en place de la cross-validation

L'objectif de cet étape est de pouvoir déterminer une valeur optimale pour l'hyperparamètre k . En effet, la valeur actuelle (cinq) a été fournie arbitrairement, et n'est pas nécessairement celle qui fournira la meilleure précision pour le modèle.

L'idée est alors d'utiliser une cross-validation pour déterminer cette valeur.

2.6.1 Découpage du jeu de données

Dans un premier temps, il nous faut découper notre fichier en couche (*folds* en anglais). Pour que le code soit le plus générique possible, il est suggéré d'utiliser un paramètre *num_folds*. Ce paramètre permettra de savoir combien de couches doivent être utilisées.

Pour réaliser la découpe, on procède comme suit :

- **Déterminer la taille des couches** Nous disposons déjà du nombre de couches, nous récupérons alors dans une variable *n_train* qui contiendra la taille de l'ensemble d'apprentissage. La taille d'une couche est alors donnée par :

$$\text{Taille apprentissage} - \frac{1}{\text{Nombre couches}} \left\lfloor \frac{\text{Taille apprentissage}}{\text{Nombre de couches}} \right\rfloor$$

Nous obtenons ainsi la plus grande valeur permettant de découper notre ensemble d'apprentissage en *num_folds* ensembles distincts.

- **Découpage de l'ensemble** Pour réaliser cette découpe, le sujet nous encourage à utiliser la méthode *mat2cell*. Cette dernière prend trois paramètres :

- ▷ L'ensemble que l'on va découper (ici les labels ou les descripteurs de l'ensemble d'apprentissage).

- ▷ La taille des ensembles à créer à partir des ensembles de base selon la première dimension. Ici, cette dernière représentera donc le nombre d'image par couche.
- ▷ La taille des ensembles à créer à partir des ensembles de base selon la seconde dimension. Ici, il s'agira de la taille des images. Dans la mesure où l'on ne cherche pas à « couper » les images, on prend simplement le nombre de pixels par images.

En utilisant ces différents éléments, on obtient directement le code suivant pour réaliser cette découpe :

```

1 n_train = size(model.X_train,1);
2 folds_lengths = ones(1,num_folds) * ((n_train - mod(n_train, num_folds))/num_folds);
3
4 X_train_folds = mat2cell(model.X_train, folds_lengths, size(model.X_train,2));
5 Y_train_folds = mat2cell(model.y_train, folds_lengths, size(model.y_train,2));

```

2.6.2 Codage de la cross-validation

Nous disposons désormais des ensembles de travail pour mettre en place la cross-validation. Il nous reste alors à coder cette dernière.

Pour la mener, il nous faut entraîner plusieurs modèles. Dans le détail, nous avons en paramètre un ensemble de valeurs de k à tester, et nous allons entraîner un modèle par couple ($k, fold$). Ainsi nous entraînerons au total $10 \times 5 = 50$ modèles au total, dont 5 par valeurs de k .

L'idée est alors que l'on va moyennner des effets de sur-apprentissage en utilisant la cross-validation, en changeant à chaque étape l'ensemble de test et une partie de l'ensemble d'apprentissage.

Il nous était demandé de plus que le code stocke tous les résultats de précision pour un affichage par la suite. Pour le code, ce dernier est alors composé d'une boucle principale qui parcourt tous les couples mentionnés ci-dessus, et à l'intérieur de cette boucle, on retrouve les blocs suivants :

- *Initialisation des ensembles de test et d'apprentissage* La première partie revient à réaliser la rotation entre les couches pour créer les ensembles de test et d'apprentissage à partir des ensembles présentés à la section précédente. Les ensembles d'apprentissage sont stockés dans les variables *train_data_cross_validation* (attributs) et *train_label_cross_validation* (labels). Les ensembles de tests ont des noms similaires, mis à part le *train* qui est remplacé en *test*. Pour réaliser la rotation, on regarde simplement quelle couche est désignée dans la boucle par l'indice j , qui correspondra aux ensembles de tests, les autres indices constituant les ensembles d'apprentissage.
- *Entraînement du modèle* Une fois les ensembles de test et d'apprentissage créé, nous entraînons le modèle KNN en réutilisant un code similaire à celui vu auparavant. La version de KNN utilisée est ici celle totalement vectorisée, dans la mesure où elle est la plus rapide et permet donc d'entraîner les 50 modèles en temps raisonnable.
- *Enregistrement des résultats* Une fois le calcul de la précision réalisé, nous enregistrons ces résultats dans un dictionnaire nommé *k_to_accuracies* ayant pour clé le couple de valeur :

(k , Numéro couche de test)

On retrouve alors ces différents éléments dans le code développé qui vous est proposé ci-dessous :

```

1 for l = 1:length(k_choices)
2     for i = 1:num_folds
3         % Création des ensembles de tests et d'apprentissage pour cette boucle. L'
4             % ensemble de test
5             % est celui associé à la couche i.
6         train_data_cross_validation = [];
7         test_data_cross_validation = [];
8         train_label_cross_validation = [];
9         test_label_cross_validation = [];
10        for j = 1:num_folds

```

```

10         if(j ~= i)
11             train_data_cross_validation = [train_data_cross_validation;
12                 X_train_folds{j}];
13             train_label_cross_validation= [train_label_cross_validation;
14                 Y_train_folds{j}];
15         else
16             test_data_cross_validation = X_train_folds{j};
17             test_label_cross_validation= Y_train_folds{j};
18         end
19     end
20
21 % On entraîne un modèle en prenant la valeur de k à l'indice l de k_choices. Il
22 % nous faut à chaque fois
23 % réentraîner le modèle et recalculer les distances car les ensembles changent à
24 % chaque étape.
25     k = k_choices(l);
26     model_cross_validated = knn_train(train_data_cross_validation,
27         train_label_cross_validation);
28     dists_no_cross_validated = knn_compute_distances_no_loops(
29         model_cross_validated, test_data_cross_validation);
30     test_labels_pred = knn_predict_labels(model_cross_validated,
31         dists_no_cross_validated, k);
32
33 % Éléments correctement prévus par le modèle pour le calcul de la précision.
34     num_correct = sum(sum(test_labels_pred == test_label_cross_validation));
35     num_test = length(test_label_cross_validation);
36
37 % Enregistrement de la valeur finale obtenue pour le couple (k, numéro couche de
38 % test).
39     k_to_accuracies(l, i) = double(num_correct)/num_test;
40
41 end
42 end

```

À l'issue de ce code, nous obtenons différentes valeurs de précisions pour les différents k , qu'il nous faudra analyser. Une première solution pour les visualiser est d'utiliser la boucle proposée dans l'énoncé qui affiche directement les valeurs en console :

```

1 k = 1, accuracy = 0.263000
2 k = 1, accuracy = 0.257000
3 k = 1, accuracy = 0.264000
4 k = 1, accuracy = 0.278000
5 k = 1, accuracy = 0.266000
6 k = 3, accuracy = 0.239000
7 k = 3, accuracy = 0.249000
8 k = 3, accuracy = 0.240000
9 ...
10 k = 50, accuracy = 0.278000
11 k = 50, accuracy = 0.269000
12 k = 50, accuracy = 0.266000
13 k = 100, accuracy = 0.256000
14 k = 100, accuracy = 0.270000
15 k = 100, accuracy = 0.263000
16 k = 100, accuracy = 0.256000
17 k = 100, accuracy = 0.263000

```

On remarque que cet affichage n'est pas optimal, nous allons donc essayer de faire mieux.

2.6.3 Déterminer la meilleure valeur de k

Une meilleure solution pour visualiser ces données est de les représenter sous forme graphique. Pour cela, un code était déjà proposé qui permettait d'afficher les précisions en fonction des valeurs de k retenues.

On obtient ainsi un graphe où pour chaque k testé, on visualise cinq valeurs de précision à la verticale de cette valeur. On obtient alors le résultat de la Figure 2.

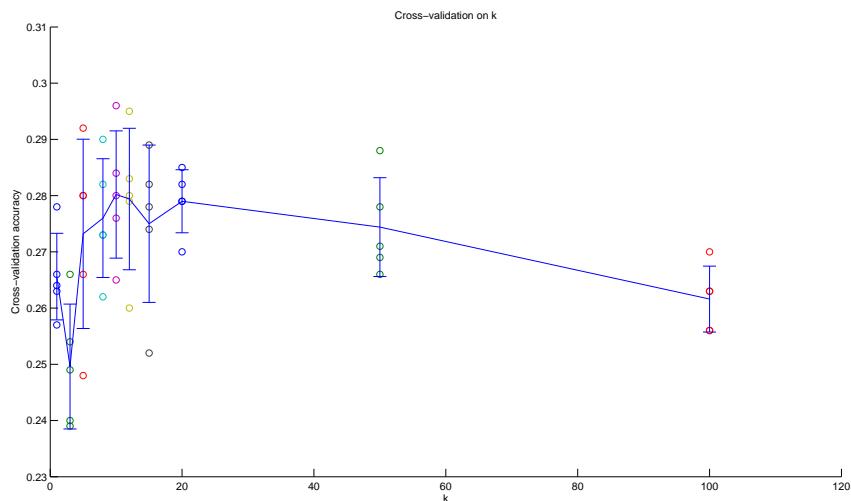


FIGURE 2 • Résultats obtenus en cross-validation avec KNN

Sur la Figure 2, les couleurs correspondent aux valeurs de k (ie. les couleurs sont à lire en verticale). De cette figure, on en retire également le fait que la valeur de k qui semble fournir les meilleurs résultats est ici $k = 10$.

2.6.4 Test du meilleur modèle

À partir de la valeur déterminer pour l'hyperparamètre k du modèle, nous avons entraîné un nouveau modèle avec $k = 10$. Nous obtenons alors les résultats suivant en précision pour ce nouveau modèle :

```
1 Got 141 / 500 correct => accuracy: 0.282000
```

Ainsi, le modèle fait légèrement mieux que le premier modèle qui avait lui obtenu une précision de 0,278. Cependant, le résultat obtenu est encore mauvais, avec une précision inférieure à 30%. Ce constat n'est pas si surprenant, dans la mesure où la stratégie ici retenue est simplement de regrouper les images par « proximité de leurs pixels » sans pré-traitement sur ces derniers, ce qui semble beaucoup trop variable pour obtenir des résultats parlant.

3 • Entraînement d'un SVM

3.1 | Préparation des données

3.1.1 Construction des ensembles d'apprentissage et de test

Pour le cas du SVM, la préparation des données reste sensiblement similaire à celle de la méthode KNN. Nous ne redétaillerons donc pas les actions réalisées, et fournissons donc juste à titre indicatif les dimensions des différents ensembles obtenus à la fin du processus.

```

1 Training data shape:
2      49000      3072
3
4 Validation data shape:
5      1000      3072
6
7 Test data shape:
8      1000      3072
9
10 Dev data shape:
11     500      3072

```

Encore une fois, on obtient un ensemble d'apprentissage dix fois plus conséquent que celui de test. La quantité de données par rapport au nombre de labels possibles peut nous permettre de nous éloigner de situation de sur-apprentissage.

3.1.2 Pré-traitement des images

Pour ce travail, nous allons nous ramener à une distribution des valeurs des pixels seront centrée. Pour cela, nous devons dans un premier temps calculer l'image moyenne sur tout l'ensemble d'apprentissage. Ceci est réalisé avec la commande suivante :

```

1 mean_image = mean(imdb.X_train, 1);
2 disp(mean_image(1:10)); % print a few of the elements
3 figure;
4 imshow(uint8(reshape(mean_image, 32, 32, 3)));

```

L'image obtenue est alors bien conforme aux attentes et vous est proposée à la Figure 3.

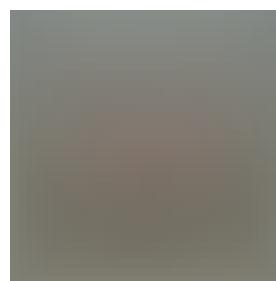


FIGURE 3 • Image moyenne calculée sur toutes les images de l'ensemble d'apprentissage

En plus de cette image, quelques valeurs peuvent être affichées, nous obtenons par exemple :

```

1 Columns 1 through 10
2
3    130.6419   130.0241   129.6634   129.4197   129.1715   128.8883   128.4675   127.9940
4      127.4662   127.2436

```

Les valeurs sont toutes comprises entre 0 et 255 puisqu'il s'agit de niveau de gris. On remarque que la plupart sont proche de 127,5 qui correspond à la valeur moyenne de tous les niveaux de gris. Ceci signifie que l'image moyenne est globalement proche du « gris moyen » comme nous l'avons vu avec la Figure 3.

Il nous reste alors à soustraire cette image de toutes les images que nous allons traiter par la suite avec notre SVM. Ceci est réalisé avec les commandes suivantes :

```
1 imbd.X_train = bsxfun(@minus, imbd.X_train, mean_image);
2 imbd.X_val   = bsxfun(@minus, imbd.X_val , mean_image);
3 imbd.X_test  = bsxfun(@minus, imbd.X_test , mean_image);
4 imbd.X_dev   = bsxfun(@minus, imbd.X_dev , mean_image);
```

Nous pouvons observer le résultat sur un cas concret qui vous est proposé à la Figure 4.



(a) Images originales

(b) Images après soustraction de l'image moyenne

FIGURE 4 • Illustration de l'effet de soustraire l'image moyenne en pré-traitement du SVM

On remarque que les images sont assombries, puisque l'on ramène les valeurs des pixels vers (0; 0; 0) c'est-à-dire du noir.

Enfin, le dernier traitement à faire revient à retirer le biais des images afin que la distribution des valeurs des pixels soit réellement centré sur zéro. Ceci nous permet en particulier d'avoir un modèle dans lequel il n'y a plus de constante (biais), et qui aura ainsi juste besoin d'entraîner une matrice W (le classifieur étant de la forme $f(x) = Wx + b$ et nous venons de faire en sorte que $b = 0$).

Le code qui était proposé pour réaliser cette action est le suivant :

```
1 imbd.X_train = cat(2, imbd.X_train, ones(size(imbd.X_train, 1), 1));
2 imbd.X_val   = cat(2, imbd.X_val , ones(size(imbd.X_val , 1), 1));
3 imbd.X_test  = cat(2, imbd.X_test , ones(size(imbd.X_test , 1), 1));
4 imbd.X_dev   = cat(2, imbd.X_dev , ones(size(imbd.X_dev , 1), 1));
```

3.2 | Calcul du gradient pour le SVM

3.2.1 Première estimation de la perte

Cette dernière (nommée *loss*) est déjà calculée dans le code fourni. Pour l'estimer, on génère un vecteur aléatoire W pour nos tests. On obtient alors le résultat suivant, qui varie un peu de celui proposé à cause du côté aléatoire de W :

```
1 loss: 9.150210
```

Ce résultat est cohérent. En effet, nous avons initialisé W avec une valeur proche de 0. Ainsi, les calculs des marges $\max(0, s_j + s_{y_i} + 1)$ seront tous très proche de 1, puisque les scores s_j et s_{y_i} seront quasiment nuls (ils proviennent du calcul vectoriel WX où $W \approx 0$). Ainsi, pour chaque image, la perte (*loss*) sera égale

à la somme de ces marges pour chaque label qui n'est pas le label théorique. Comme il y a dix labels au total, cela implique que ces *loss* seront tous proche de 9. Leur moyenne est donc naturellement proche de 9.

Nous allons par la suite nous intéresser au calcul du gradient.

3.2.2 Expression théorique du gradient

L'objectif est ici de calculer numériquement le gradient de la fonction de perte d'un SVM. Pour mémoire, cette fonction de perte notée \mathcal{L}_i est donnée par (3).

$$\mathcal{L}_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) = \sum_{j \neq y_i} \max(0, w_j x_i - w_{y_i} x_i + 1) \quad (3)$$

Dans cette relation (3), i désigne l'image courante et y_i son label théorique.

En reprenant les notations du code, nous pouvons définir la marge (*margin*) par la relation (4).

$$\text{margin}(i, j) = s_j - s_{y_i} + 1 = w_j x_i - x_{y_i} x_i + 1 \quad (4)$$

Pour réaliser la dérivation, nous allons alors distinguer deux cas :

- ◎ *Si $j = y_i$* Cela signifie que l'on a le bon label. Dans ce cas, la relation (3) se dérive par :

$$\nabla_{w_{y_i}} \mathcal{L}_i = - \left(\sum_{j \neq y_i} \mathbb{1}_{\text{margin}(i,j)>0} \right) x_i$$

Cette relation se justifie par le fait que si la marge est positive, seul le terme $-w_{y_i} x_i$ n'est pas constant et est de dérivée $-x_i$ (de plus ce terme est présent pour chaque élément de la somme). Pour les termes où la marge est nulle, il n'y a contribution. Ainsi, cette dérivée nous dit que la dérivée est calculée en faisant $\ell \times x_i$ où ℓ est le nombre de labels autres que y_i pour lesquels la marge était non nulle.

- ◎ *Si $j \neq y_i$* Dans ce cas là, seul un élément de la somme fera apparaître le w_j , la dérivée ne fait donc pas apparaître de somme et est simplement :

$$\nabla_{w_j} \mathcal{L}_i = \mathbb{1}_{\text{margin}(i,j)>0} x_i$$

Ainsi, nous aurons une contribution si la marge est non nulle pour ce terme, autrement il ne contribue pas au calcul du gradient.

3.2.3 Implémentation naïve du gradient

Ce gradient peut donc se calculer par simple additions/soustractions dans un vecteur. Dans le détail, nous allons réaliser ces opérations si la marge calculée est positive (nous venons de voir que c'était le dénominateur commun des deux situations de calcul du gradient). Concernant les opérations à réaliser, nous proposons donc le code suivant :

```

1 if margin > 0
2   loss = loss + margin;
3   % Ligne incorrecte, on ajoute le vecteur X_i
4   dW(j,:) = dW(j,:) + X(i,:);
5   % Ligne correcte, on retranche le vecteur X_i
6   dW(y(i),:) = dW(y(i),:) - X(i,:);
7 end

```

On pourra noter que toutes les opérations sont réalisées à chaque étape, ce qui peut être contre-intuitif avec la distinction de cas précédente. L'idée est en fait ici de réaliser des calculs qui seront complémentaires pour obtenir le bon résultat. Ainsi, on considère par défaut que tous les labels sont incorrects, d'où l'ajout de x_i en position j dans le vecteur de gradient. Cependant, nous allons dans la foulée retrancher un x_i au bon label.

Ainsi, si le label est le bon, les deux lignes donneront $\nabla_{w_j} = \nabla_{w_j} + x_i - x_i$ ie. une valeur inchangée. Si ce n'est pas le cas, on crédite donc bien la ligne correspondant au bon poids avec l'opération d'addition. Mais on débite également la ligne du bon labels par la valeur de x_i de la même manière et on compte ainsi bien le nombre de ligne où le label n'est pas le bon.

Pour terminer ce calcul, il nous suffit juste de normaliser le gradient, ce qui est fait avec la ligne de code suivante :

```
1 dW = dW/num_train;
```

Où `num_train` est le nombre d'éléments de l'ensemble d'apprentissage.

3.2.4 Gradient et régularisation

Deux éléments de régularisation sont utilisées, un pour la perte et un pour le gradient.

Régularisation de la perte Pour cette dernière, nous avions vu en cours qu'il était important de faire apparaître les poids dans le calcul du *loss* afin d'avoir un contrôle sur ces derniers et d'éviter que ces poids « n'exploserent ». La régularisation choisie ici est de type \mathcal{L}_2 , ie. la fonction *loss* est finalement donnée par :

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) + 0,5 \cdot \gamma \cdot \|W\|_2^2$$

Les notations sont les mêmes que précédemment, et on voit un facteur de régularisation γ faire son apparition.

Du point de vue du code, la transcription est immédiate et donne :

```
1 loss = loss + 0.5 * reg * sum(sum((W.*W)));
```

Cette dernière peut donc être facilement tronquée en prenant `ref` (γ) de valeur nulle.

Régularisation du gradient Le gradient de la fonction de *loss* va donc lui aussi être impacté par cette régularisation, puisque cette dernière a changée l'expression de cette fonction. Nous devons donc dériver $0,5 \cdot \gamma \cdot \|W\|_2^2$ par rapport à W et l'ajouter aux expressions précédentes.

Nous obtenons ainsi immédiatement la relation suivante :

$$\nabla_W \mathcal{L} = \text{éléments précédents} + \gamma \cdot W$$

Le code est ici aussi immédiat :

```
1 dW = dW + reg*W;
```

Nous avons ainsi terminé le code de l'implémentation naïve du gradient.

3.2.5 Tests du calcul du gradient

Nous pouvons alors tester notre implémentations avec le code proposé. Une première remarque est que les calculs sont longs (entre 3 et 5 minutes par modèle sur ma machine). Ainsi, je n'ai pas pu mener autant de tests que ceux proposés dans le sujet, puisqu'il m'aurait fallu plus d'une heure de calculs pour faire toutes les vérifications de gradient proposées.

J'ai ainsi biaisé, en utilisant la version vectorisé pour vérifier l'algorithme, dans la mesure où nous verrons plus loin que les résultats obtenus avec ces deux versions sont similaires. Ceci m'a permis de fortement réduire le temps de calcul.

Résultats sans régularisation En utilisant le code qui était proposé, nous obtenons les résultats suivants pour le calcul du gradient sans prise en compte de la régularisation :

```
1 numerical: 19.380616 analytic: 3.915530, relative error: 6.638474e-01
2 numerical: -0.648005 analytic: -0.804531, relative error: 1.077610e-01
3 numerical: -4.570463 analytic: -1.715290, relative error: 4.542292e-01
4 numerical: 0.688764 analytic: -0.119660, relative error: 1.000000e+00
```

```

5 numerical: -4.808132 analytic: -1.743966, relative error: 4.676618e-01
6 numerical: -33.554246 analytic: -6.957595, relative error: 6.565155e-01
7 numerical: -16.989836 analytic: -3.766604, relative error: 6.370664e-01
8 numerical: 10.189438 analytic: 2.032654, relative error: 6.673803e-01
9 numerical: -2.397566 analytic: -0.670936, relative error: 5.626946e-01
10 numerical: -5.449724 analytic: -1.197988, relative error: 6.395789e-01

```

Nous obtenons des résultats donc les erreurs relatives restent globalement faibles, ce qui est bon signe concernant la qualité de l'algorithme proposé. Les résultats sont cependant moins proches que ceux proposés dans le sujet (de l'ordre de 10^{-3}), je n'ai cependant pas réussi à trouver la cause de cet écart, qui est d'autant plus étonnant que tous les résultats de la suite de cette partie sont cohérents avec ceux proposés dans le sujet... (on remarque que nos valeurs sont toujours amplifiées par rapport aux analytiques).

Résultats avec régularisation En utilisant la régularisation, nous obtenons alors les résultats suivants :

```

1 numerical: -13.131484 analytic: -2.715970, relative error: 6.572357e-01
2 numerical: -27.019560 analytic: -5.168717, relative error: 6.788448e-01
3 numerical: 10.438349 analytic: 2.187200, relative error: 6.535279e-01
4 numerical: -3.333422 analytic: -0.736167, relative error: 6.382106e-01
5 numerical: 0.954749 analytic: 0.247321, relative error: 5.885085e-01
6 numerical: -2.045625 analytic: -0.343608, relative error: 7.123694e-01
7 numerical: -5.333761 analytic: -1.022005, relative error: 6.784006e-01
8 numerical: 24.274570 analytic: 5.677375, relative error: 6.209011e-01
9 numerical: -3.052256 analytic: -0.866996, relative error: 5.575706e-01
10 numerical: -27.378968 analytic: -5.689007, relative error: 6.559205e-01

```

Là aussi, les résultats sont proches de la valeur calculée analytiquement. On retrouve ici des résultats quasiment identiques à ceux proposés dans le sujet.

Valeurs abhérantes Il était mentionné dans le sujet qu'il était possible d'avoir des directions dans lesquelles les deux résultats ne soient pas proches. La raison est que la fonction de perte du SVM fait intervenir un maximum et n'est donc pas une fonction différentiable. Ainsi, il existe des directions privilégiées dans lesquels le calcul ne peut se réaliser et où les valeurs risquent de fortement diverger selon la position où l'on se situe près de cette direction.

3.3 | Version vectorielle du code

Comme pour le classifieur KNN, utiliser des boucles *for* a l'intérêt de fournir un code simple, cependant ce dernier met du temps à s'exécuter. Pour parer à cela, nous allons désormais nous intéresser au codage d'une version vectorielle de l'algorithme précédent.

Les résultats théoriques ont pour la plupart déjà été cités, nous allons donc nous intéresser ici plus en détails au aspects algorithmiques pour coder ces résultats. En effet, nous avions vu qu'il existait deux cas dans le calcul du gradient qu'il va nous falloir prendre en compte dans ce calcul vectoriel.

3.3.1 Calcul vectoriel du loss

Pour présenter ce code, nous vous proposons de vous fournir initialement le code et d'en détailler les différents blocs dans la suite de cette section. Le code proposé pour le calcul de la fonction de perte est alors le suivant :

```

1 % Les scores calculés
2 scores = W*X';
3 % Construction de la matrice des scores des labels théoriques
4 prov_ = eye(max(y));
5 theoric_labels = prov_(y,:); % 1 si c'est le bon label, 0 sinon
6 theoric_scores = theoric_labels.*scores; % scores des labels théoriques
7 theoric_scores_all = repmat(max(theoric_scores) + min(theoric_scores), 10, 1);
8 % On reprend un delta de 1

```

```

9  deltas = ones(10, num_train);
10 % Calcul final des scores pour les différentes images
11 L = scores - theoric_scores_all + deltas;
12 % On met à 0 les cases correspondants aux classes théoriques
13 L = L-theoric_labels'.*L;
14 % On applique le max sur les scores calculés
15 L(L<0)=0;
16 % Calcul des gains pour chaque image et régularisation
17 loss = sum(sum(transpose(L))/num_train + 0.5 * reg * sum(W'.*W'));
```

Les objectifs de ce code sont de réussir à calculer le *loss* pour tous les images, ie. d'implémenter la relation (3).

Calcul des scores Tout d'abord, cette relation fait intervenir les scores des différentes images calculés à partir des poids W. Ce calcul est immédiat par la relation WX (X est le vecteur d'images et W celui des poids). Le code est celui proposé en ligne 2.

Labels théoriques Pour exprimer la relation (3), on remarque qu'il nous faut pour chaque image connaître son label théorique et surtout le score de ce dernier (noté s_{y_i} dans la relation). Pour cela, nous procédons en trois étapes :

- **1. Binarisation des labels** On crée une matrice *theoric_labels* qui pour chaque image lui associe un tableau de dix entiers tous nuls sauf un vallant 1 pour le bon label. Si par exemple une image à le label « lapin » qui est le troisième label, alors sa ligne dans cette matrice sera : [0 0 1 0 0 0 0 0 0].

Ces opérations sont réalisées avec les lignes 4 et 5 du code.

- **2. Insertion des scores** La matrice précédente ne contient qu'une information binaire. Il nous suffit simplement désormais de faire une multiplication élément par élément avec la matrice de score pour avoir une matrice où les scores théoriques sont situées « à l'endroit des bons labels ». Pour reprendre notre exemple, si le score de « lapin » pour cette image est de 0,25, nous aurions alors la ligne de cette image dans la matrice qui sera : [0 0 0.25 0 0 0 0 0 0].

Cette opération est réalisée à la ligne 6 du code.

- **3. Duplication de l'information** Pour terminer, nous proposons la valeur obtenue pour ce label théorique à tous les autres labels. Pour mémoire, l'objectif de la matrice que nous construisons est qu'elle représente le « $-s_{y_i}$ » dans le calcul de \mathcal{L} .

L'astuce à cette étape est que les scores des mauvais labels sont à 0 dans *theoric_scores*, mais rien ne nous assure que le score de la classe théorique sera positif! Ainsi, pour récupérer ce score, nous sommes obligés de sommer le maximum et le minimum sur la ligne (un des deux sera nul) et de répliquer sur le reste de la ligne. Ceci est réalisé à la ligne 7 du code.

Calcul des marges Avant de calculer \mathcal{L} , il nous reste encore à créer le vecteur *delta* qui représente la constante dans le calcul du *loss*, et qui est mise à 1 dans ce code.

Nous pouvons alors calculer la marge pour chaque terme, c'est-à-dire la somme :

$$s_j + s_{y_i} + 1 = scores - theoric_scores_all + delta$$

Ceci est réalisé à la ligne 11 du code. La matrice obtenue contient donc toutes les marges. Cependant, nous avions que pour le calcul final de \mathcal{L} , la marge de la classe théorique n'intervenait pas. Ainsi, nous devons retirer cette dernière de cette matrice de marge.

La stratégie est alors d'utiliser la matrice des labels comme un masque sur *L* pour savoir quels sont les éléments à retirer. Ceci est exécuté à la ligne 13 du code.

Calcul du loss Pour terminer, nous pouvons alors calculer explicitement le *loss*. Pour commencer, nous appliquons à la ligne 15 le maximum pour ne conserver que les marges positives calculées.

Enfin, la valeur finale de la perte est obtenue en sommant tous les termes de la matrice de marges (pour mémoire, ceux des labels théoriques sont mis à zéro) et d'ajouter la régularisation. Ceci est exécuté à la ligne 17 du code.

Nous obtenons ainsi un calcul de la perte sans avoir à utiliser de boucles.

3.3.2 Tests du calcul vectoriel du loss

Un code proposé nous permet de comparer la rapidité d'exécution ainsi que les résultats obtenus avec les deux méthodes. Les résultats obtenus sont les suivants :

```

1 Naive loss: 9.150210e+00 computed in 134.268545s
2 Vectorized loss: 9.150210e+00 computed in 1.615899s
3 difference: -0.000000

```

Nous remarquons effectivement que la version vectorisée est bien plus rapide pour obtenir le même résultat.

3.3.3 Calcul vectoriel du gradient

Pour le calcul du gradient, le code vectorisé vous est proposé ci-dessous :

```

1 % On reprend le calcul intermédiaire du coût, calcul du premier terme
2 L = scores - theoric_scores_all + deltas;
3 % Calcul de la "dérivée du max"
4 L(L<0) = 0;
5 L(L>0) = 1;
6 % On retire la valeur du label théorique
7 L = L - theoric_labels' .* L;
8 % On retire la somme par colonne que l'on ajoute au label théorique
9 L = L - theoric_labels' .*(ones(10,1)*sum(L,1));
10 % On ajouter un terme
11 dW = L*X/num_train + reg*W;

```

Nous allons détailler les différentes étapes de ce code dans la suite de cette section.

Dérivation du maximum Le point de départ du calcul est ici le calcul des marges. Nous avions en effet vu que tout le calcul du gradient s'appuyait sur un comptage des marges non nuls par poids et par vecteur.

On recalcule donc le vecteur des marges (qui n'avait pas été stocké tel quel pour gagner un peu de place en mémoire), ceci est proposé à la deuxième ligne de code.

Ensuite, nous seuillons cette matrice. En effet, nous avions vu qu'il ne fallait pour le calcul du gradient que conserver les termes de marges positives. De plus, notre but étant uniquement de faire du comptage, nous binarisons les valeurs (`0` si la marge est négative, `1` sinon).

Retirer le label théorique Comme pour le calcul de la perte, le terme correspondant au label théorique n'intervient pas dans les sommes de calcul du gradient. Nous utilisons donc le même artifice qui revient à utiliser la matrice `theoric_labels` comme masque sur `L` pour supprimer les valeurs nécessaires.

Ceci est réalisé à la ligne 7 de ce code.

Nous pouvons remarquer qu'à cette étape nous avons donc calculé tous les gradients pour les situations où il ne s'agit pas du bon label.

Calcul du gradient pour les bons labels Pour cela, il nous suffit de compter pour chaque image le nombre de marges positives, ie. le nombre de coefficients à `1` dans la matrice `L` pour la ligne de l'image concernée.

L'opposé de cette valeur est alors stocké dans le coefficient correspondant au label théorique. Nous avons donc bien réalisé le comptage attendu.

Finalisation du calcul Pour terminer, il nous reste à multiplier par le vecteur de données `X`, puisque les totaux étaient multipliées par ce dernier dans les relations théoriques vues précédemment. Nous ajoutons également la régularisation, comme cela est proposé dans la dernière ligne de code.

3.3.4 Tests du calcul vectoriel du gradient

Comme pour le calcul de la perte, nous allons alors comparer les deux algorithmes pour le calcul du gradient. Les résultats sont les suivants :

```

1 Naive loss and gradient: computed in 138.538762s
2 Vectorized loss and gradient: computed in 1.753995s
3 difference: 0.000000

```

Dans la mesure où le gradient retourné est un vecteur, nous comparons les deux implémentations en utilisant la norme de Frobenius de la différence. Pour mémoire, cette dernière est donnée par :

$$\|A\|_F = \sqrt{\sum_{j=1}^m \sum_{i=1}^n |a_{ij}|^2} = \sqrt{\text{Tr}(A^t A)}$$

Ainsi, cette mesure permettra de relever le moindre coefficient avec une différence importante entre les deux matrices.

Nous observons que là aussi les estimations sont les même, les codes sont donc bien équivalents.

3.4 | Descente stochastique du gradient

Dans cette partie, nous venons de voir comment calculer la perte et le gradient. Cependant, tous nos essais ont actuellement été réalisés avec un W aléatoire. L'objectif désormais est d'utiliser ces méthodes de calcul pour optimiser notre perte et apprendre un classifieur sur nos données.

Pour cela, nous allons devoir coder une solution qui scindera nos données en différents sous-ensemble pour les utiliser au fur et à mesure de l'apprentissage.

L'idée est ici de choisir aléatoirement un sous-ensemble de données à chaque itération de l'apprentissage et d'actualiser le vecteur W en utilisant sa dérivée ∇W (rappelons qu'il ne s'agit pas exactement d'une dérivée puisque la fonction de perte n'est pas à proprement parler dérivable).

3.4.1 Sélection aléatoire d'un jeu de données

Le fichier de code pour l'apprentissage est composé d'une boucle principale qui est itérée autant de fois que demandé en paramètre.

À chaque itération, nous allons sélectionner un *batch*, c'est-à-dire un sous-ensemble aléatoire des données d'apprentissage sur lequel nous allons apprendre notre modèle.

Le code réalisé est alors le suivant :

```

1 random_indexes = randsample(1:num_train, batch_size);
2 X_batch = X(random_indexes,:);
3 y_batch = y(random_indexes,:);

```

La fonction *randsample* nous permet de générer *batch_size* nombre aléatoires différents compris entre *1* et *num_train*. Il nous reste alors juste à sélectionner les données de l'ensemble d'apprentissage correspondant à ces indices, ainsi que leurs labels.

3.4.2 Mise à jour des poids

La suite de la boucle revient simplement à l'entraînement du modèle avec le code suivant :

```

1 [loss, grad] = self_loss(W, X_batch, y_batch, reg);
2 loss_hist(it) = loss;

```

On remarquera que la perte est stockée dans un vecteur pour affichage de son évolution à la fin. Le gradient calculé est ensuite utilisé pour mettre à jour le vecteur de poids *W*:

```
1 W = W - learning_rate*grad;
```

On remarquera l'utilisation d'un paramètre *learning_rate* qui précise l'importance du pas de mis à jour. Une valeur faible rendra le modèle plus lent à apprendre, mais peut être plus précis à la fin. Une valeur plus élevée pourra au contraire faire réaliser au modèle du « yoyo » autour du minimum recherché avec des pas par saut de gradient trop long.

3.4.3 Résultats de la descente stochastique

Nous obtenons le retour Matlab suivant pour l'évolution de la perte au long de l'apprentissage :

```

1 iteration 100 / 1500: loss 290.478747
2 iteration 200 / 1500: loss 109.267771
3 iteration 300 / 1500: loss 43.486785
4 iteration 400 / 1500: loss 19.308942
5 iteration 500 / 1500: loss 10.088612
6 iteration 600 / 1500: loss 7.119709
7 iteration 700 / 1500: loss 5.952824
8 iteration 800 / 1500: loss 5.755674
9 iteration 900 / 1500: loss 5.698171
10 iteration 1000 / 1500: loss 5.064359
11 iteration 1100 / 1500: loss 5.386994
12 iteration 1200 / 1500: loss 6.125514
13 iteration 1300 / 1500: loss 4.734350
14 iteration 1400 / 1500: loss 5.549989
15 iteration 1500 / 1500: loss 5.305891
16 That took 25.706522s

```

Cette évolution de la fonction de perte peut se visualiser plus clairement sur un graphe comme celui que nous avons obtenu à la Figure 5.

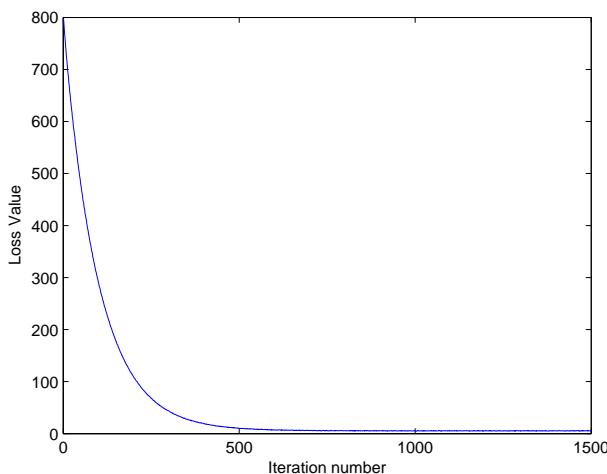


FIGURE 5 • Évolution de la perte au long des différentes étapes de l'apprentissage

On remarque également qu'il aurait été impossible de faire cela sans la version vectorisé tant le temps de calcul aurait été trop important! Concernant la décroissance de la perte, on remarque que l'on part d'une valeur très élevée pour revenir rapidement dans des valeurs acceptables. La décroissance se fait sous la forme d'une fonction $-\log$.

3.4.4 Évaluation du modèle

Nous avons désormais un modèle qui nous est donné, cependant il nous faut pouvoir l'évaluer. Pour cela, nous allons coder la méthode de test de ce modèle dans `linear_sum_predict.m`.

L'idée est ici très simple, il s'agit de calculer les scores obtenus avec notre vecteur W appris sur un ensemble de test. Nous calculons alors le score obtenu par chaque classe sur chaque image, et nous conservons pour chaque image la classe avec le meilleur label.

Le code associé est alors simple et réutilise des idées déjà vues précédemment :

```

1 scores = model.W*X';
2 for i = 1:size(X,1)
3     [~, index] = max(scores(:,i));

```

```
4     y_pred(i) = index;  
5 end
```

On remarquera que la ligne 3 nous permet d'obtenir directement dans *index* l'index du score maximum pour l'image, qui est ensuite enregistré dans *y_pred*.

L'utilisation de ce code nous fournit alors le résultat suivant :

```
1 training accuracy: 0.370102  
2 validation accuracy: 0.378000
```

Les précisions obtenues sont relativement similaires, signe que notre modèle n'a pas sur-appris sur l'ensemble d'apprentissage (c'était un des intérêts de la sélection aléatoire des données pour les itérations). Dans la mesure où le modèle propose 10 labels, un classifieur au hasard aurait en moyenne une précision de 10%, ainsi notre classifieur même s'il n'est pas extrêmement performant fait tout de même mieux que le hasard avec ses 37% de précision.

3.5 | Optimisation des hyper-paramètres

Les résultats obtenus à la section précédente l'ont été en prenant les paramètres fournis par défaut. Cependant, une solution pour améliorer notre modèle est de trouver des valeurs optimales pour les hyper-paramètres de ce dernier.

Pour se faire, nous allons entraîner des modèles par couple d'hyper-paramètre et regarder quelles valeurs semblent les plus intéressantes.

3.5.1 Les hyper-paramètres

Avant d'entrer plus en détail dans leur optimisation, rappelons que les deux hyper-paramètres pour notre SVM sont les suivants :

- ◉ *Taux d'apprentissage* Il s'agit du paramètre qui définit à quel point notre gradient va faire évoluer *W* à chaque itération. Il apparaît dans la relation de mise à jour de *W* :

$$W = W + \text{taux d'apprentissage} \times \nabla W$$

- ◉ *Poids de régularisation* Il s'agit du paramètre qui précise l'importance de la prise en compte des poids de *W* dans le calcul de la perte \mathcal{L} . Nous avions nommé ce paramètre γ dans les sections précédentes.

Ce sont ces deux variables que nous allons essayer d'optimiser pour obtenir de meilleures performances pour notre modèle.

3.5.2 Recherche du meilleur couple de valeurs

Pour cela, nous commençons par utiliser les deux tableaux de valeurs proposés pour ces paramètres, qui vous sont rappelés ci-dessous :

```
1 learning_rates = [1e-7, 2e-7, 3e-7, 5e-5, 8e-7];  
2 regularization_strengths = [1e4, 2e4, 3e4, 4e4, 5e4, 6e4, 7e4, 8e4, 1e5];
```

En plus de ces valeurs, des objets pour stocker les meilleurs résultats sont également définis. L'idée est alors de parcourir tous les couples possibles de la forme (*learning_rates*; *regularization_strengths*) et d'entraîner pour chacun d'eux un modèle. Pour chaque modèle, nous estimons sa précision et conservons celui qui fournit la meilleure.

Le code réalisé utilise ainsi deux boucles *for* imbriquées pour parcourir tous les couples possibles et reprend ensuite le code classique d'évaluation du modèle. Une fois le modèle évalué, on retrouve un petit test permettant de savoir si le modèle est le meilleur vu jusqu'à présent. Le cas échéant, on met à jour les variables enregistrant ses meilleures valeurs.

En plus de cela, nous stockons tous les résultats obtenus dans la variable *results*.

Le code est en lui-même assez limpide au final :

```

1 iter_num = 1000;
2 %iter_num = 100;
3
4 for i = 1:size(learning_rates,2)
    % Pour voir le travail avancer...
    disp(i)
    for j = 1:size(regularization_strengths,2)
        % On entraîne un modèle pour le couple d'hyper-paramètres
        learning_rate = learning_rates(i);
        regularization_strength = regularization_strengths(j);
        [model, loss_hist] = linear_svm_train(imdb.X_train, imdb.y_train,
            learning_rate, regularization_strength, iter_num, 200, 0);
        % Calcul de sa précision
        validation_accuracy = mean(imdb.y_val == linear_svm_predict(model, imdb.X_val
            )');
        % Enregistrement du résultat
        results(i,j,:) = validation_accuracy;
        % Mise à jour du meilleur couple si la précision est la meilleure obtenue.
        if best_val < validation_accuracy
            best_svm = model;
            best_val = validation_accuracy;
        end
    end
end

```

On remarquera que le nombre d'itération peut rendre ce code un peu long à exécuter. En effet, nous allons entraîner $5 \times 9 = 45$ modèles. Ainsi, en prenant 1000 itérations par modèle, cela représente 45000 itérations (ie. 45000 calculs de gradient et de perte), le temps de calcul n'est donc pas négligeable!

Les résultats obtenus sont alors les suivants. Nous présentons pour chaque modèle la valeur de *learning_rate* testée, la valeur de *regularization_rate* testée ainsi que les précisions obtenues en apprentissage et en test.

```

1 lr 1.000000e-07 reg 1.000000e+04 train accuracy: 0.348000 val accuracy: 0.348000
2 lr 1.000000e-07 reg 2.000000e+04 train accuracy: 0.378000 val accuracy: 0.378000
3 lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.374000 val accuracy: 0.374000
4 lr 1.000000e-07 reg 4.000000e+04 train accuracy: 0.375000 val accuracy: 0.375000
5 lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.374000 val accuracy: 0.374000
6 lr 1.000000e-07 reg 6.000000e+04 train accuracy: 0.385000 val accuracy: 0.385000
7 lr 1.000000e-07 reg 7.000000e+04 train accuracy: 0.375000 val accuracy: 0.375000
8 lr 1.000000e-07 reg 8.000000e+04 train accuracy: 0.373000 val accuracy: 0.373000
9 lr 1.000000e-07 reg 1.000000e+05 train accuracy: 0.367000 val accuracy: 0.367000
10 lr 2.000000e-07 reg 1.000000e+04 train accuracy: 0.387000 val accuracy: 0.387000
11 lr 2.000000e-07 reg 2.000000e+04 train accuracy: 0.371000 val accuracy: 0.371000
12 lr 2.000000e-07 reg 3.000000e+04 train accuracy: 0.386000 val accuracy: 0.386000
13 lr 2.000000e-07 reg 4.000000e+04 train accuracy: 0.376000 val accuracy: 0.376000
14 lr 2.000000e-07 reg 5.000000e+04 train accuracy: 0.367000 val accuracy: 0.367000
15 lr 2.000000e-07 reg 6.000000e+04 train accuracy: 0.368000 val accuracy: 0.368000
16 lr 2.000000e-07 reg 7.000000e+04 train accuracy: 0.347000 val accuracy: 0.347000
17 lr 2.000000e-07 reg 8.000000e+04 train accuracy: 0.368000 val accuracy: 0.368000
18 lr 2.000000e-07 reg 1.000000e+05 train accuracy: 0.333000 val accuracy: 0.333000
19 lr 3.000000e-07 reg 1.000000e+04 train accuracy: 0.383000 val accuracy: 0.383000
20 lr 3.000000e-07 reg 2.000000e+04 train accuracy: 0.349000 val accuracy: 0.349000
21 lr 3.000000e-07 reg 3.000000e+04 train accuracy: 0.359000 val accuracy: 0.359000
22 lr 3.000000e-07 reg 4.000000e+04 train accuracy: 0.370000 val accuracy: 0.370000
23 lr 3.000000e-07 reg 5.000000e+04 train accuracy: 0.381000 val accuracy: 0.381000
24 lr 3.000000e-07 reg 6.000000e+04 train accuracy: 0.351000 val accuracy: 0.351000
25 lr 3.000000e-07 reg 7.000000e+04 train accuracy: 0.338000 val accuracy: 0.338000
26 lr 3.000000e-07 reg 8.000000e+04 train accuracy: 0.341000 val accuracy: 0.341000
27 lr 3.000000e-07 reg 1.000000e+05 train accuracy: 0.363000 val accuracy: 0.363000
28 lr 5.000000e-05 reg 1.000000e+04 train accuracy: 0.196000 val accuracy: 0.196000
29 lr 5.000000e-05 reg 2.000000e+04 train accuracy: 0.136000 val accuracy: 0.136000

```

```

30 lr 5.000000e-05 reg 3.000000e+04 train accuracy: 0.155000 val accuracy: 0.155000
31 lr 5.000000e-05 reg 4.000000e+04 train accuracy: 0.094000 val accuracy: 0.094000
32 lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.066000 val accuracy: 0.066000
33 lr 5.000000e-05 reg 6.000000e+04 train accuracy: 0.068000 val accuracy: 0.068000
34 lr 5.000000e-05 reg 7.000000e+04 train accuracy: 0.087000 val accuracy: 0.087000
35 lr 5.000000e-05 reg 8.000000e+04 train accuracy: 0.087000 val accuracy: 0.087000
36 lr 5.000000e-05 reg 1.000000e+05 train accuracy: 0.087000 val accuracy: 0.087000
37 lr 8.000000e-07 reg 1.000000e+04 train accuracy: 0.353000 val accuracy: 0.353000
38 lr 8.000000e-07 reg 2.000000e+04 train accuracy: 0.326000 val accuracy: 0.326000
39 lr 8.000000e-07 reg 3.000000e+04 train accuracy: 0.299000 val accuracy: 0.299000
40 lr 8.000000e-07 reg 4.000000e+04 train accuracy: 0.298000 val accuracy: 0.298000
41 lr 8.000000e-07 reg 5.000000e+04 train accuracy: 0.339000 val accuracy: 0.339000
42 lr 8.000000e-07 reg 6.000000e+04 train accuracy: 0.269000 val accuracy: 0.269000
43 lr 8.000000e-07 reg 7.000000e+04 train accuracy: 0.276000 val accuracy: 0.276000
44 lr 8.000000e-07 reg 8.000000e+04 train accuracy: 0.294000 val accuracy: 0.294000
45 lr 8.000000e-07 reg 1.000000e+05 train accuracy: 0.338000 val accuracy: 0.338000
46 best validation accuracy achieved during cross-validation: 0.387000

```

On remarque rapidement que la plupart des résultats sont supérieurs à 0,3 en précision de validation, mis à part quelques résultats catastrophique avec des précision de l'ordre de 8-9% (pire que le hasard!).

Nous apprenons également que le meilleure modèle obtient une précision en test de 38,7% ce qui est un peu mieux que les modèles vus précédemment (un peu plus de 1% de mieux). Les valeurs associées sont les suivantes :

$$\text{learning_rate}: 2 \cdot 10^{-7} \quad \text{regularization_rate}: 1 \cdot 10^4$$

Cependant, cet affichage textuel n'est pas le plus optimal, nous fournissons alors à la Figure 6 un résumé visuel de ces valeurs.

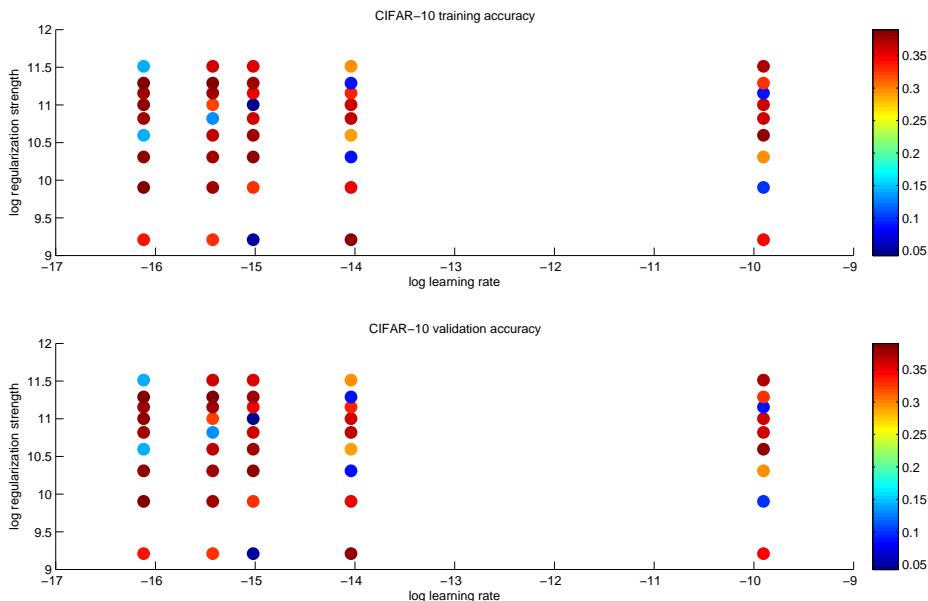


FIGURE 6 • Précisions obtenues pour les différentes combinaisons d'hyperparamètres testées en apprentissage et en test

La Figure 6 présente en haut les valeurs obtenues en apprentissage et en bas celles en test. Chaque point sur les graphiques représente un modèle qui a été entraîné pour une combinaison d'hyperparamètre. Les valeurs de ces derniers peuvent être lues en abscisses et en ordonnées (attention cependant, les valeurs sont proposées en logarithme). Enfin, la couleur d'un point dépend de la précision obtenue, plus cette dernière est élevée, plus le point vire au bordeaux; plus elle est faible et plus le point vire au bleu foncé.

On remarque ainsi que les précisions obtenues en apprentissage et en test suivent globalement les même tendance (ie. elles sont « bonnes » ou « mauvaises » au même endroits). On remarque également des précisions qui semblent très légèrement meilleures en apprentissage, ce qui est normal.

On retrouve également le meilleur précision proposé par le retour « textuel » de l'algorithme.

Enfin, on retrouve aussi que les taux d'apprentissage les plus faibles donnent globalement de meilleurs résultats, comme nous l'avons mentionné en présentant cet hyperparamètre. Pour le poids de régularisation, il est plus difficile de tirer une conclusion générique, même s'il semblerait que des valeurs en log entre 10,8 et 11,3 soient favoriser.

3.5.3 Évaluation du meilleur modèle

Nous pouvons enfin conclure en évaluant notre meilleur modèle sur l'ensemble de test. Le résultat est alors le suivant :

```
1 linear SVM on raw pixels final test set accuracy: 0.366000
```

Le résultat reste cohérence avec les valeurs précédentes. Là encore, la précision n'est pas exceptionnellement bonne (mais un peu meilleure que celle vue auparavant). Il est cependant important de préciser que le modèle n'apprend pas sur des descripteurs de l'image, mais uniquement sur les couleurs des pixels...ce qu'un humain ne ferait pas manière intuitive.

Ainsi, ce présupposé initial pour construire le modèle semble impliquer qu'il ne sera pas possible d'obtenir un modèle performant juste avec cette idée (un peut trop) simple.

3.6 | Visualisation des poids

Pour terminer, nous allons regarder visuellement ce qui a été appris par le modèle. Pour cela, nous reprenons le meilleur modèle appris, et on affiche des poids qui auront été étalés entre 0 et 255 pour correspondre à des composantes RGB.

Nous obtenons alors le résultat de la Figure 7.

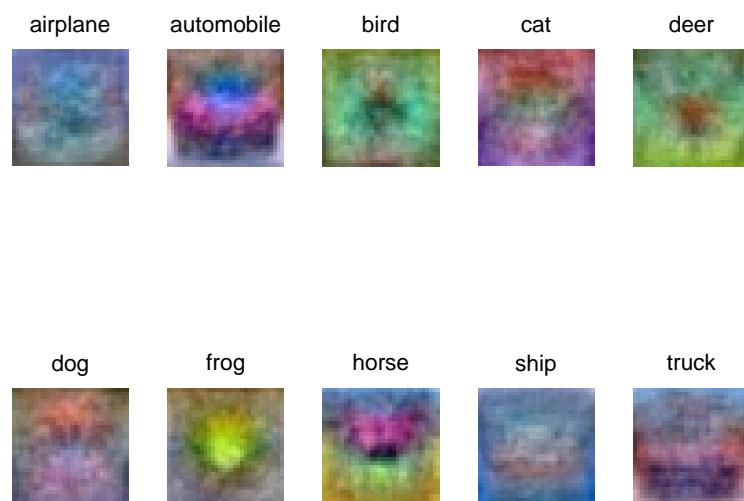


FIGURE 7 • Visualisation des poids appris par le meilleur modèle SVM entraîné

Une première remarque est que sur la plupart des poids présenté, le centre de l'image semble être

ce qui se détache le plus. Ceci est donc un premier indicateur, qui est que notre modèle a considéré que toutes les scènes avaient des objets à peu près centré.

Concernant la reconnaissance des objets, on peut penser « deviner » une voiture, un cheval ou un camion, mais pas de manière nette. Cependant cela semble indiquer que notre SVM est capable d'extraire des formes mêmes si elles restent flous.

Enfin, un indicatif intéressant est la couleur des poids. On remarque ainsi que pour la grenouille la partie centrale de l'image est verte (comme une grenouille), pour un bateau les contours de l'image sont bleus (comme l'étendue d'eau sur laquelle il peut être). De même pour le cheval où encore le cerf où la couleur centrale rappelle celle de l'animal d'origine. De même, pour tous les animaux sauvages (ie. pas le chat ni le chien), on remarque que les couleurs périphériques tendent à du vert.

Ainsi, nous pouvons avancer en regardant ces poids que le modèle appris s'est construit autours des éléments suivants :

- Mise en avant des éléments au centre de l'image plus que des éléments périphériques.
- Pour certains éléments, une vague forme générale semble être sortie par l'algorithme.
- Un élément central semble être la détection des couleurs qui au vue des poids proposés semble jouer un rôle important dans le calcul des scores.

4 • Un classificateur softmax

4.1 | Préparation des données

Encore une fois, les traitements sont fortement similaires, en appelant à nouveau la méthode de prétraitement des données `prepare_datasets`. Nous fournissons ainsi juste la dimension des données obtenues à l'issue de cette étape :

```

1 Training data shape:
2      49000      3073
3
4 Training labels shape:
5      49000      1
6
7 Validation data shape:
8      1000      3073
9
10 Validation labels shape:
11      1000      1
12
13 Test data shape:
14      1000      3073
15
16 Test labels shape:
17      1000      1

```

Nous retrouvons les éléments habituels, à savoir que nous traitons des images de dimension 32×32 avec trois canaux couleurs RGB. Ces images sont représentées par des vecteurs à une dimension en concaténant tous les canaux de couleurs et les pixels, d'où les 3073 dans les dimensions. Nous notons également que le jeu total de données dispose de 50000 images, que nous avons découpé en un ensemble d'apprentissage de 49000 images et un ensemble de test de 1000 images.

Les labels sont eux encodés numériquement de 1 à 10 dans des vecteurs.

4.2 | Calcul du gradient pour un classifieur Softmax

4.2.1 Perte calculée par le Softmax

Rappel théorique Pour mémoire, la fonction de perte utilisée dans le cas d'un classifieur Softmax est donnée par (5).

$$\mathcal{L}_i = -\log \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right) \quad (5)$$

Nous rappelons que l'indice i désigne l'image pour laquelle on calcule la perte, et que les termes de la forme s_j sont des scores issus du calcul WX . La notation y_i quant à elle désigne le label théorique de l'image.

Implémentation du loss L'implémentation proposée est la suivante, que nous allons vous commenter par la suite :

```

1 % Grandeur générales
2 num_class = size(W,1);
3 num_train = size(X,1);
4
5 for i = 1:num_train

```

```

6 % Calcul des scores pour l'image
7 scores = W*X(i, :)';
8 % Le score théorique
9 correct_class_score = scores(y(i));
10 % Calcul du dénominateur (somme des exponentielles).
11 exp_scores = exp(scores);
12 exp_regularization = sum(exp_scores);
13 % Calcul explicite du loss
14 loss = loss - correct_class_score + log(exp_regularization);
15 end
16
17 % Moyenne et ajout de la régularisation
18 loss = loss/num_train + .5*reg*sum(sum(W.*W));

```

Nous retrouvons dans un premier temps deux constantes pour nous aider, à savoir le nombre de classes et le nombre d'images à traiter. La suite du code est constituée d'une boucle qui va traiter les images une à une. Ce traitement comporte les étapes suivantes :

- **1. Calcul des scores** On utilise pour cela la ligne correspondant à l'image ($X(i, :)$) que l'on multiplie au vecteur de poids W .
- **2. Extraction du score théorique** On cherche le bon label, qui est à l'index $y(i)$ des labels.
- **3. Calcul du dénominateur** Il est ici question de calculer la somme des exponentielles. Pour cela, on passe tous les scores calculés à l'exponentielle avant de simplement sommer le vecteur.
- **4. Assemblage** Nous utilisons ici une autre forme de l'expression du *loss* que nous verrons un peu plus bas. L'idée est ici simplement de décomposer la fraction du logarithme pour éviter une division (l'addition sera légèrement plus rapide et nous gagnons un calcul d'exponentielle).

Pour terminer, on moyenne toutes les valeurs obtenues et on ajoute la régularisation en norme \mathcal{L}_2 des poids.

Test de l'implémentation Il ne nous reste plus qu'à tester l'implémentation. La solution retenue est d'utiliser un vecteur W proche de 0 comme nous l'avions fait pour le SVM. Le résultat obtenu est le suivant :

```

1 loss: 2.322556
2 sanity check: 2.302585

```

Nous obtenons bien un résultat proche de celui annoncé. Ceci est cohérent, car, comme dans le cas du SVM, tous les scores vont être proches de 0. Ainsi, les exponentielles apparaissant dans la relation (5) vont toutes avoir une valeur proche de 1, la relation (5) revient alors simplement à :

$$\mathcal{L}_i \approx -\log \left(\frac{1}{\sum_j 1} \right) = -\log(0, 1)$$

Il y a en effet 10 classes dans notre cas, la somme au dénominateur a donc une valeur de 10.

4.2.2 Calcul théorique du gradient

Avant de fournir le code calculant le gradient de manière naïve, nous allons présenter quelques éléments théoriques expliquant le code réalisé.

Pour commencer, nous allons redévelopper un peu l'expression (5).

$$\mathcal{L}_i = x_i w_{y_i} + \log \left(\sum_j^L e^{x_i w_j} \right)$$

Dans cette relation, L désigne le nombre de classes possibles. Comme pour le SVM, ce qui nous intéresse alors est de dériver cette relation en fonction de W . Nous allons donc regarder ce qu'il en est pour un w_i

donné :

$$\nabla_{w_k} \mathcal{L}_i = \begin{cases} x_i + \nabla_{w_k} \mathcal{L} \left[\log \left(\sum_j^L e^{x_i w_j} \right) \right] & \text{si, } k = y_i \\ \nabla_{w_k} \mathcal{L} \left[\log \left(\sum_j^L e^{x_i w_j} \right) \right] & \text{sinon} \end{cases}$$

Il nous reste donc à estimer la dérivée restante. Pour rappel, les règles de dérivation usuelles fournissent :

$$\nabla_x (\log(f(x))) = \frac{\nabla_x f(x)}{f(x)} \quad \text{et} \quad \nabla_x (e^{f(x)}) = \nabla_x (f(x)) e^{f(x)}$$

En utilisant ces deux règles, nous obtenons la dérivée cherchée :

$$\nabla_{w_k} \mathcal{L} \left[\log \left(\sum_j^L e^{x_i w_j} \right) \right] = \frac{x_i e^{x_i w_y}}{\sum_j^L e^{w_j x_i}}$$

Nous avons donc réussi à calculer la dérivée. Il est possible de condenser les deux cas en utilisant une indicatrice, nous aboutissons donc à la relation (6) pour la dérivée.

$$\nabla_{w_k} \mathcal{L}_i = -x_i \cdot \left(\mathbb{1}_{y_i=k} - \frac{e^{w_y x_i}}{\sum_j^L e^{w_j x_i}} \right) = -x_i \cdot \left(\mathbb{1}_{y_i=k} - \frac{e^{s_{y_i}}}{\sum_j^L e^{s_j}} \right) \quad (6)$$

Il ne nous reste alors plus qu'à implémenter cette relation.

4.2.3 Implémentation du gradient

Algorithme proposé À partir des calculs réalisés, l'implémentation du gradient est assez simple. Nous allons ainsi insérer le code suivant à la fin de l boucle calculant le loss :

```

1 for j = 1:num_class
2   if(j == y(i))
3     dW(j,:) = dW(j,:) + (exp_scores(j)/exp_regularization-1)*X(i,:);
4   else
5     dW(j,:) = dW(j,:) + (exp_scores(j)/exp_regularization)*X(i,:);
6   end
7 end

```

Les calculs réalisés traduisent directement les relations démontrées. On retrouve le premier cas qui correspond à la situation où on dérive sur le label théorique (d'où le -1 à l'intérieur des parenthèses); alors que le deuxième cas correspond à la situation où on ne dérive pas sur le bon label.

Après ces calculs, nous devons comme pour la fonction de perte moyenner ce vecteur et ajouter le terme de régularisation (son origine est la même que pour le SVM) en utilisant ce code :

```
1 dW = dW/num_train + reg*w;
```

Rien de nouveau ici encore.

Question de l'instabilité Dans le sujet, il est fait mention que les calculs peuvent être instables. En cause, la division des exponentielles, dont les valeurs individuelles peuvent être très élevées. Si un point d'attention était mis sur cet aspect, aucun problème n'a été rencontré lors de l'exécution de notre code.

Afin d'être totalement complet cependant, nous nous sommes renseignés sur Internet concernant les moyens de limiter cette instabilité (je n'avais aucune idée à chaud pour le faire). La solution trouvée est alors de remarquer que :

$$\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} = \frac{e^C}{\sum_j e^{s_j}} \times \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} = \frac{e^{s_{y_i}+C}}{\sum_j e^{s_j+C}}$$

L'idée est alors de prendre une constante $C = -\max_j(\{0; s_j\}) \leq 0$. Ainsi, pour tout j , $s_j + C \leq 0$, et donc $e^{s_j+C} \leq 1$. Ceci nous garantit que les grandeurs que nous aurons n'auront pas de valeurs trop importante.

Le code modifié pour le calcul du gradient est alors le suivant :

```

1 % Calcul de la constante pour rectifier
2 C = max(0, max(scores));
3 % Nouveau calcul des exponentielles
4 exp_scores = exp(scores+C);
5 exp_regularization = sum(exp_scores);
6 % Calcul du gradient inchangé (seules les matrices ont changé).
7 for j = 1:num_class
8     if(j == y(i))
9         dW(j,:) = dW(j,:) + (exp_scores(j)/exp_regularization-1)*X(i,:);
10    else
11        dW(j,:) = dW(j,:) + (exp_scores(j)/exp_regularization)*X(i,:);
12    end
13 end

```

Nous obtenons ainsi le même résultat, mais avec moins d'instabilité numérique.

4.2.4 Test du gradient

Pour terminer sur cette implémentation naïve du gradient et de la fonction de perte, nous allons nous intéresser à comparer les résultats à ceux d'un gradient obtenu analytiquement.

Comme pour le SVM, le temps de calcul d'un seul modèle softmax étant long, nous avons anticipé sur la suite et utilisé la version vectorielle pour réaliser ces comparaisons. L'intérêt est alors de fortement diminuer les temps de calcul. De plus, la différence entre les deux étant négligeables, le résultat reste similaire.

Sans régularisation Comme nous l'avions dit pour le SVM, il est possible de désactiver la régularisation en prenant un poids de régularisation nulle. Nous allons faire ceci dans un premier temps.

Les résultats obtenus sont alors les suivants :

```

1 numerical: 1.561442 analytic: 1.561442, relative error: 3.807356e-08
2 numerical: 1.039400 analytic: 1.039400, relative error: 1.148115e-08
3 numerical: -4.794890 analytic: -4.794890, relative error: 8.058074e-09
4 numerical: -0.488162 analytic: -0.488162, relative error: 6.483646e-09
5 numerical: -1.346944 analytic: -1.346944, relative error: 1.402692e-08
6 numerical: 0.120068 analytic: 0.120068, relative error: 3.873910e-08
7 numerical: -0.545961 analytic: -0.545961, relative error: 3.705597e-08
8 numerical: -3.979085 analytic: -3.979085, relative error: 1.452250e-08
9 numerical: 0.698970 analytic: 0.698970, relative error: 5.616316e-08
10 numerical: -0.164754 analytic: -0.164754, relative error: 2.043165e-07

```

Nous obtenons des écarts relatifs très faibles (et meilleurs que ceux que nous avions obtenu pour le SVM). Ceci semble indiquer que le calcul réalisé était le bon.

Avec régularisation Pour conclure sur cette vérification du code, nous utilisons désormais le calcul dans un cas avec régularisation. Les résultats obtenus sont les suivants :

```

1 numerical: -2.848628 analytic: -2.843879, relative error: 8.342946e-04
2 numerical: 2.837119 analytic: 2.847489, relative error: 1.824310e-03
3 numerical: -2.168466 analytic: -2.166968, relative error: 3.454958e-04
4 numerical: 0.395343 analytic: 0.409910, relative error: 1.808924e-02
5 numerical: 1.302744 analytic: 1.317357, relative error: 5.577377e-03
6 numerical: -0.463066 analytic: -0.447855, relative error: 1.669822e-02
7 numerical: 3.891436 analytic: 3.905082, relative error: 1.750333e-03
8 numerical: 0.434086 analytic: 0.426090, relative error: 9.296042e-03
9 numerical: 0.088595 analytic: 0.097976, relative error: 5.027746e-02
10 numerical: -0.249656 analytic: -0.231660, relative error: 3.738967e-02

```

Nous retrouvons encore une fois des résultats très similaires. Cependant, les erreurs relatives sont ici plus importante, dans la mesure où nous utilisons les poids en plus du calcul « pur » du gradient, qui seront plus bruités. Les écarts restent cependant tout à fait acceptable pour notre application.

4.3 | Vectorisation des calculs

Comme pour les deux classifieurs précédents, l'algorithme actuellement proposé à base de boucle fournit certes de bons résultats, mais il n'est cependant pas rapide. Nous allons donc ici aussi proposer une version vectorielle de cet algorithme.

Nous présenterons dans un premier temps la vectorisation du *loss* avant de nous intéresser à celle du calcul du gradient.

4.3.1 Vectorisation du calcul du loss

Le code Pour la fonction de perte, les éléments vont être sensiblement similaires à ce précédent. Le code proposé est alors le suivant :

```

1 num_class = size(W,1);
2 num_train = size(X,1);
3
4 % Calcul des scores
5 scores = W*X';
6
7 %% Calcul de la perte
8 % Calcul des termes de normalisation
9 sum_by_images = log(sum(exp(scores)));
10 % Calcul des termes hors normalisation
11 prov_ = eye(max(y));
12 theoric_labels = prov_(y,:); % 1 si c'est le bon label, 0 sinon
13 theoric_scores = theoric_labels' .* scores; % scores des labels théoriques
14 % On combine le tout ensemble
15 loss = - sum(theoric_scores) + sum_by_images;
16 % On ajoute la régularisation
17 loss = sum(transpose(loss))/num_train + .5*reg*sum(sum(W' .* W'));

```

Nous retrouvons comme dans la version itérative la définition de nos constantes générales donnant le nombre d'images et le nombre de classes possibles.

Calcul des scores Nous retrouvons également le calcul des scores, si ce n'est que dans cette nouvelle version nous calculons directement le produit des deux matrices au lieu de faire un produit matrice-vecteur. On notera l'utilisation de X' (la transposée) pour des raisons de dimension des vecteurs.

Calcul du terme de normalisation Pour le calcul de la perte nous réutilisons l'idée de décomposer le logarithme. Pour cela, nous commençons par calculer le terme de normalisation (la somme des exponentielles des scores) pour chaque image que nous stockons dans *sum_by_images*.

Calcul des termes hors normalisation Vient ensuite le calcul des termes hors normalisation, i.e. du numérateur de la relation (5). Pour cela, nous utilisons la même méthode que pour le SVM. Nous recalculons ainsi une matrice *theoric_labels* qui contient une ligne par image dont les coefficients sont tous à 0 sauf celui du label théorique qui est à 1. Pour obtenir les scores des labels théoriques, nous multiplions alors simplement cette matrice terme à terme avec celle des scores précédemment calculés.

On remarquera ainsi que la matrice *theoric_scores* contient les scores des labels théoriques *non portés à l'exponentielle*. En effet, nous utilisons la formule où le logarithme a été développé, nous n'avons donc pas besoin de cette exponentielle !

Calcul effectif de la perte Nous disposons donc à cette étape de tous les éléments intervenant dans la formule avec régularisation de la partie, à savoir :

- `theoric_scores` Les scores des labels théoriques image par image, non portés à l'exponentielle.
- `sum_by_images` La somme des exponentielles servant dans la relation donnant le *loss*.
- `num_train` Le nombre d'image dans l'ensemble d'apprentissage.
- `W` Le vecteur des poids qui est utilisé pour la régularisation.

Avec ces éléments, nous faisons dans un premier temps le calcul suivant :

```
1 loss = - sum(theoric_scores) + sum_by_images;
```

Ce calcul correspond au calcul de la fonction de perte pour chaque image, et sans prendre en compte la régularisation. En particulier, la variable `loss` est alors un vecteur colonne, avec autant de lignes que d'images, et donc chaque valeur correspond au *loss* de l'image de cette ligne.

Pour terminer le calcul, nous réalisons alors l'opération suivante :

```
1 loss = sum(transpose(loss))/num_train + .5*reg*sum(sum(W'.*W'));
```

Il s'agit ici simplement de sommer toutes les pertes et d'ajouter le terme de régularisation en fonction de la norme \mathcal{L}_2 de W . On retrouve la même idée qu'avec le SVM.

4.3.2 Vectorisation du calcul du gradient

Cette vectorisation est encore plus simple que dans le cas du SVM dans la mesure où nous avons déjà calculé tous les éléments intermédiaires qui étaient nécessaires à ce calcul, c'est-à-dire :

- `scores` La matrice de scores.
- `theoric_labels` La matrice avec des `1` uniquement pour les labels des différentes images. Il est intéressant de voir qu'ici cette matrice fournit directement la forme matricielle de l'indicatrice qui apparaît dans le calcul du gradient.
- *Les différentes constantes* Que ce soit le vecteur des images `X`, celui des poids `W` ou encore le coefficient de régularisation `reg`.

Ces différents éléments se combinent dans le code ci-dessous :

```
1 %% Calcul du gradient
2 dS = exp(scores)./(ones(10,1)*sum(exp(scores))) - theoric_labels' ;
3 dW = dS*X/num_train + reg*W;
```

Sur la première ligne qui calcul le terme `dS` qui correspond au terme dans la parenthèse de la relation (6). Nous remarquons la multiplication au dénominateur par `ones(10,1)` qui permet de réaliser une somme sur tous les labels possibles d'une image.

La seconde ligne contient elle la moyenne des dérivées à laquelle on ajoute la régularisation comme nous l'avions fait pour le SVM.

4.3.3 Comparer les implémentations

Il convient désormais de tester si cette implémentation est bien équivalente à la version itérative proposée précédemment. Nous allons ainsi comparer les temps d'exécution, les valeurs obtenues pour les pertes et les gradients. Concernant les gradients, la même question se pose quand à la manière de les comparer, nous utiliserons donc ici aussi la norme de Frobenius.

Les résultats obtenus sont les suivants :

```
1 Naive loss: 2.322556e+00 computed in 51.758478s
2 Vectorized loss: 2.322556e+00 computed in 1.322337s
3 Loss difference: 0.000000
4 Gradient difference: 0.000000
```

Nous obtenons des pertes identiques, de même que la norme de la différence des gradients. Les temps de calculs sont particulièrement impactés, avec une version vectorisée qui est 50 fois plus rapide que la version itérative.

4.4 Descente stochastique du gradient pour le softmax

Nous avons désormais une méthode capable de calculer la perte et le gradient de cette dernière pour une étape de notre classifieur softmax. Il reste désormais à utiliser ces résultats pour pouvoir faire des mises à jour des poids W à chaque étape pour diminuer la perte et affiner le modèle.

Les différents éléments qui vous seront présentés ici sont très proches de ce qui avait été fait pour le SVM, nous ne nous attarderons donc pas outre mesure à en redétailler tous les aspects

4.4.1 Entraînement du modèle

Pour réaliser cet entraînement, nous procédons comme pour le SVM en réalisant des *batchs* dans l'ensemble d'apprentissage à chaque itération de l'apprentissage. On retrouve alors un code identique :

```
1 random_indexes = randsample(1:num_train, batch_size);
2 X_batch = X(random_indexes, :);
3 y_batch = y(random_indexes, :);
```

Nous estimons alors un modèle sur ce sous-ensemble de données et utilisons le résultat du gradient obtenu pour mettre à jour les poids W . Là aussi, un paramètre fixant le taux d'apprentissage est utilisé :

```
1 W = W - learning_rate*grad;
```

Le code est donc semblable en tout point à celui du SVM.

4.4.2 Entraînement du modèle

Le code est encore identique à celui du SVM, nous ne commenterons donc pas plus que cela cette contribution :

```
1 scores = model.W*X';
2 for i = 1:size(X,1)
3     [~, index] = max(scores(:,i));
4     y_pred(i) = index;
5 end
```

4.5 Optimisation des hyper-paramètres

Ce modèle présente lui aussi deux hyper-paramètre, dont il est bon de chercher la combinaison optimale afin d'obtenir un modèle affiné. Pour mémoire, ces hyper-paramètres sont :

- *learning_rates* Pour déterminer le pas d'apprentissage et évaluer à quelle vitesse on s'autorise à faire varier la valeur des poids W .
- *regularization_strengths* Pour déterminer l'importance des poids de W dans le calcul du *loss* et du gradient, ie. à quel point nous allons essayer de contrôler ces derniers.

4.5.1 Code pour réaliser l'optimisation

Nous retrouvons un principe de code similaire. Nous allons donc ici aussi parcourir tous les couples, stocker la meilleure valeur et stocker également les valeurs intermédiaires pour affichage ultérieur.

Le code suivant est donc assez similaire à celui du SVM, dans la mesure où nous y retrouvons exactement les mêmes éléments :

```
1 iter_num = 1000;
2
3 for i = 1:size(learning_rates,2)
4     disp(i)
```

```
5     for j = 1:size(regularization_strengths,2)
6         learning_rate = learning_rates(i);
7         regularization_strength = regularization_strengths(j);
8         [model, loss_hist] = linear_svm_train(imdb.X_train, imdb.y_train,
9             learning_rate, regularization_strength, iter_num, 200, 0);
10        validation_accuracy = mean(imdb.y_val == linear_svm_predict(model, imdb.X_val
11            ));
12        results(i,j,:) = validation_accuracy;
13        if best_val < validation_accuracy
14            best_softmax = model;
15            best_val = validation_accuracy;
16        end
17    end
18 end
```

Pour mémoire, les sorties utiles sont les suivantes :

- `results` Cette structure va associer à une paire de valeurs pour nos hyperparamètre la précision obtenue.
- `best_val` Ce nombre stocker la meilleure précision observée.
- `best_softmax` Cette structure va retenir le meilleur modèle. Ce modèle est constitué des poids qui ont fournis la meilleure précision.

4.5.2 Résultats obtenus

Pour le softmax, nous testerons moins de modèle que nous ne l'avons fait pour le SVM. En effet, les deux hyper-paramètres vont prendre leurs valeurs dans :

```
1 learning_rates = [1e-7, 5e-7];
2 regularization_strengths = [5e4, 1e8];
```

Nous n'allons ainsi essayer que quatre modèles différents. En ajoutant à ceci le fait que nous réalisons 1000 itération pour décider d'un modèle, nous n'allons donc réaliser au total que 4000 itérations, c'est dix fois moins que pour le SVM.

Ce choix est réalisé dans la mesure où le SVM nous a indiqué quelles valeurs semblaient les plus intéressantes (faible pas d'apprentissage et régularisation assez élevée). Ainsi, les valeurs pré-choisies semblent en accord avec cette idée.

Les résultats obtenus sont les suivants :

```
1 lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.347000 val accuracy: 0.347000
2 lr 1.000000e-07 reg 1.000000e+08 train accuracy: 0.087000 val accuracy: 0.087000
3 lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.328000 val accuracy: 0.328000
4 lr 5.000000e-07 reg 1.000000e+08 train accuracy: 0.087000 val accuracy: 0.087000
5 best validation accuracy achieved during cross-validation: 0.347000
```

Nous remarquons que les résultats sont un peu moins bons que le SVM, mais nous atteignons toujours les presque 35% en précision, ce qui est honorable.

4.5.3 Test du modèle optimal

Nous avons désormais trouvé un modèle optimal, avec les valeurs suivantes pour ses hyper-paramètres :

`learning_rate: 1 · 10-7 regularization_rate: 5 · 104`

On remarque que ces valeurs sont avoisinantes de celles obtenues pour le SVM (ce qui conforte notre remarque précédente sur le choix restreint de valeurs). Reste alors à évaluer le modèle qui est associer à cette paire de valeurs sur l'ensemble de test.

La précision obtenue est alors de :

```
1 linear Softmax on raw pixels final test set accuracy: 0.340000
```

Nous obtenons une précision de 34% à peine inférieure à celle obtenue en apprentissage. Encore une fois, ceci est un signe que le modèle a fortement évité le sur-apprentissage.

Les mêmes remarques que le SVM s'appliquent, à savoir que ce modèle fait mieux que le hasard, mais n'est pas non plus exceptionnellement bon. La raison reste là aussi une approche uniquement fondée sur de l'analyse pixel par pixel...

4.6 | Visualisation des poids du softmax

Pour terminer sur l'implémentation de ce classifieur softmax, nous pouvons observer les poids appris dans W pour chaque classe. Une représentation de ces derniers vous est proposée à la Figure 8.

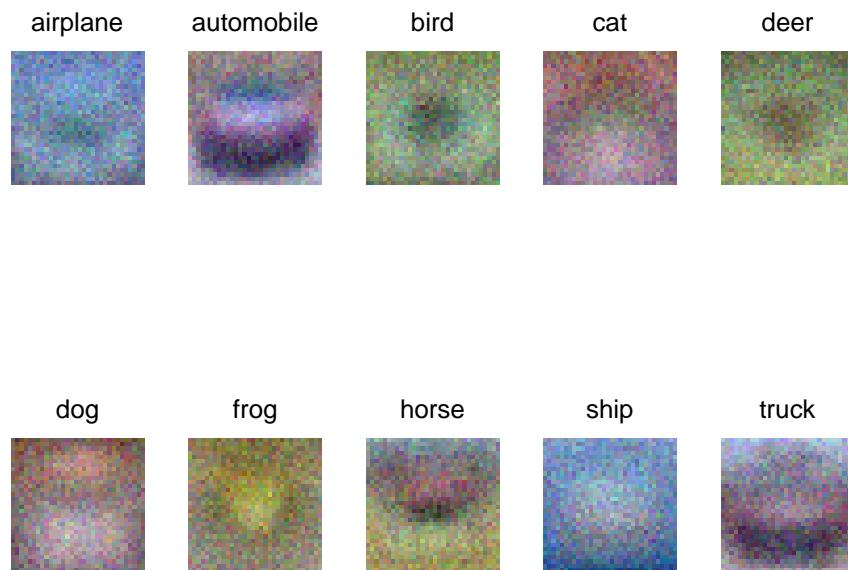


FIGURE 8 • Visualisation des poids appris par le meilleur modèle softmax entraîné

On remarque que les visuels sont assez similaires à ceux obtenus avec le SVM. Ainsi, la plupart des conclusions tenues sur les SVM sont encore valables ici, à savoir :

- Mise en avant des éléments centraux de l'image.
- Structure générale qui est dégagée pour certaines classes comme la voiture ou le cheval.
- Une très forte importance des couleurs, que ce soit pour l'élément central ou les couleurs de pour-tours (couleurs d'herbe, de mer,...).

Ceci illustre bien le fait que le SVM et le softmax sont une famille d'algorithme assez proche avec des idées globalement similaires. En effet, la seule différence vue ici a été la fonction de calcul du *loss*, le reste de la méthode étant identique. Nous voyons également qu'il serait possible d'adapter ces méthodes en utilisant une autre fonction de perte, et que nous obtiendrions des résultats encore légèrement différents mais qui resteraient normalement dans le même ordre d'idée.

5 • Réseau neuronal à deux niveaux

5.1 | Création d'un réseau jouet

5.1.1 Réseau jouet

Avant d'utiliser nos vraies données et un réseau « complet », nous allons travailler sur un modèle et des données plus simples. L'idée est ici de pouvoir tester simplement nos implémentations (en terme de temps de calculs) avant de l'appliquer sur des exemples plus complexes et plus long à calculer.

Nous utiliserons ici un modèle à deux couches, qui est initialisé par la méthode `init_toy_model` qui appelle elle-même la méthode `twolayernet_init`.

```
1 model = init_toy_model(input_size, hidden_size, num_classes);
```

On remarquera que par rapport au code proposé, nous avons externalisé la fonction de création dans un fichier, car la définition de fonction dans un script a été refusée par Matlab.

Comme nous l'avons dit, ce modèle est qualifié de jouet car il utilise un nombre restreint de neurones et de couches. De même pour les données, qui au lieu de présenter les 3072 paramètres des images n'en auront que quatre. De même, on ne considérera que trois classes possibles en sortie au lieu des 10 possibles pour les images.

L'idée est donc bien ici d'avoir un réseau où les calculs sont rapides, afin de pouvoir vérifier nos implémentations.

5.1.2 Jeu de données pour le réseau

Comme le réseau attend moins d'entrées et fournit moins de sorties que le réseau complet pour nos images, il nous faut générer des données spécifiques pour ce dernier.

Ceci est réalisé par la méthode `init_toy_data`, appelée ci-dessous :

```
1 [X,y] = init_toy_data(num_inputs, input_size);
```

Les constantes utilisées ont pour valeurs :

```
1 input_size = 4;
2 num_classes = 3;
3 num_inputs = 5;
```

Nous générerons donc cinq vecteurs contenant quatre variables que nous stockons dans la matrice `X`. Les classes possibles des vecteurs sont au nombre de trois, et sont enregistrées dans `y`.

Dans le détail, l'initialisation de `X` est réalisée à partir de valeurs tirées aléatoirement selon une loi normale centrée réduite. On augmente cependant la variance en multipliant les valeurs obtenues par `10`. Dans l'idée de fournir des résultats reproductibles par rapport à ceux de l'énoncé, l'aléatoire est cependant bloqué en imposant la « graine » pour générer ces valeurs aléatoires. Il est à noter que la graine générant les vecteurs de poids aléatoirement est elle aussi fixée.

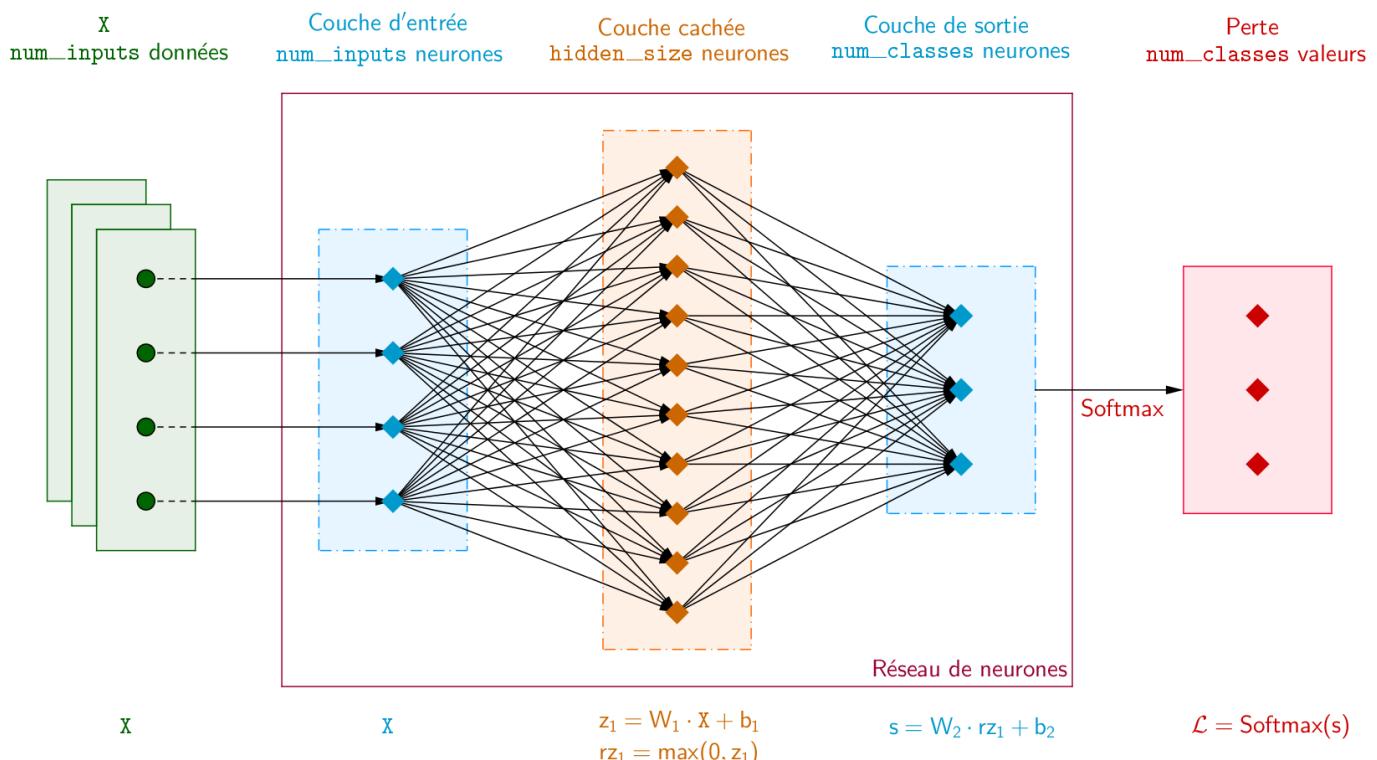
Pour `y`, les valeurs sont directement fixées dans le code. L'idée est en effet d'obtenir une situation reproductible.

5.1.3 Structure obtenue

Dans la mesure où nous venons d'expliquer que tous les poids étaient déterministes, nous pouvons fournir ci-dessous le résultat de ce petit réseau.

La structure du réseau vous est rappelée à la Figure 9.

Données en entrée Pour visualiser ces dernières, nous proposons de regarder la matrice obtenue en concaténant `X` et `y`.

**FIGURE 9** • Structure du réseau de neurones pour le problème jouet

```

1 [X y] =
2
3 -6.4901 -5.7266 -8.5189 1.6681 1.0000
4 11.8117 -5.5868 8.0032 -19.6542 2.0000
5 -7.5845 1.7838 -15.0940 -12.7007 3.0000
6 -11.0961 -1.9686 8.7587 11.7517 3.0000
7 -8.4555 5.8644 -2.4279 20.2916 2.0000

```

Première couche de neurones La couche en entrée contient quatre neurones (un par descripteur sur les données). La couche cachée contient elle 10 neurones (c'est un paramètre du modèle), et chaque neurone va recevoir une entrée de la part des quatre neurones de la couche d'entrée. Vous pouvez retrouver ces informations et les notations associées à la Figure 9.

On obtient ainsi une couche qui va réaliser l'opération suivante (sans prendre en compte l'utilisation de la fonction ReLU).

$$W_1 X + b_1$$

Où $X \in \mathcal{M}_{5,4}$, et ainsi $W_1 \in \mathcal{M}_{4,10}$ et $b_1 \in \mathcal{M}_{1,10}$. Les valeurs obtenues sont les suivantes :

```

1 model.W1 =
2
3 0.0538 0.0319 0.3578 0.0725 -0.0124 0.0671 0.0489 0.0294
4 -0.1069 0.0325
5 0.1834 -0.1308 0.2769 -0.0063 0.1490 -0.1207 0.1035 -0.0787
6 -0.0809 -0.0755
7 -0.2259 -0.0434 -0.1350 0.0715 0.1409 0.0717 0.0727 0.0888
8 -0.2944 0.1370
9 0.0862 0.0343 0.3035 -0.0205 0.1417 0.1630 -0.0303 -0.1147
10 0.1438 -0.1712

```

Nous voyons que par défaut le biais est mis à zéro pour la couche.

Pour la couche suivante (qui est la couche de sortie du réseau), elle va réaliser l'opération suivante :

$$W_2 \underbrace{(W_1 X + b_1)}_{\in \mathcal{M}_{5,10}} + b_2$$

Nous aurons donc d'un point de vue dimensions $W_2 \in \mathcal{M}_{10,3}$ et $b_2 \in \mathcal{M}_{1,3}$. La sortie étant elle aussi en définitive un vecteur de $\mathcal{M}_{1,3}$, avec un score possible par classe en sortie.

Les valeurs obtenues à l'initialisation de W_2 et b_2 sont :

```

1 model.W2 =
2
3 -0.0102   -0.0864   -0.1089
4 -0.0241    0.0077    0.0033
5  0.0319   -0.1214    0.0553
6  0.0313   -0.1114    0.1101
7 -0.0865   -0.0007    0.1544
8 -0.0030    0.1533    0.0086
9 -0.0165   -0.0770   -0.1492
10 0.0628    0.0371   -0.0742
11 0.1093   -0.0226   -0.1062
12 0.1109    0.1117    0.2350
13
14 model.b2 =
15
16 0 0 0

```

Une fois ces considérations de dimensions établies, nous pourrons plus facilement visualiser les calculs à mener pour combiner ces matrices entre elles. Nous remarquons également que le nombre de paramètres de ce (petit) réseau est de :

$$(4+1) \times 10 + (10+1) \times 3 = 50 + 33 = 83 \text{ paramètres}$$

5.2 | Passe forward

Cette passe correspond à l'envoi de données en entrée du réseau, pour obtenir le calcul des scores pour ces données en sortie. À partir de ce calcul des scores, nous pourrons réaliser une deuxième passe en arrière dite *backward* pour mettre à jour les poids (ce sera l'objet de la section suivante).

5.2.1 Calcul des scores

Dans un premier temps, l'objectif est simplement d'obtenir la valeur des scores en sortie du réseau. Nous rappelons la manière de calculer les différents états du réseau au fur et à mesure de la propagation des données pour un vecteur x en entrée (les notations sont celles proposées à la Figure 9) :

- **En entrée** On prend en entrée le vecteur x et donc ses quatre composantes.
- **Couche suivant l'entrée (cachée)** Cette couche réalise en fait deux calculs. Un premier qui est un simple calcul de régression ($W_1 x + b_1$), suivi d'une fonction de seuillage. La fonction utilisée pour ce BE est la fonction ReLU bien que d'autres choix soient possibles. En résumé, cette couche réalise l'opération suivante :

$$z_1 = W_1 x + b_1 \quad \text{et} \quad r z_1 = \max(0, z_1)$$

- **Couche de sortie** Cette dernière se contente de « rajouter » une régression à partir des résultats de la couche précédente.

$$s = W_2 * z_1 + b_2 = W_2 * (\max(0, W_1 x + b_1)) + b_2$$

Il nous reste alors à implémenter ceci dans la méthode *twoLayerNet_loss*.

```

1 hidden_layer = max(0,X*model.W1 + repmat(model.b1,N,1));
2 scores = hidden_layer*model.W2+repmat(model.b2,N,1);

```

Le parti pris dans ce code est de décomposer les couches une à une pour plus de clareté. Ainsi, le calcul pour la couche cachée est fourni par la première ligne (`hidden_layer`). Ce résultat est ensuite réutilisé pour calculer les scores finaux.

On remarquera la nécessité de répliquer les vecteurs b_1 et b_2 , puisque nous écrivons un code entièrement matriciel qui prend tous les individus en entrée d'un seul coup (ils sont au nombre de $N = 5$).

Nous vérifions alors nos résultats avec ceux proposés (pour mémoire, la graine pour l'aleatoire est fixée) :

```

1 Your scores:
2   0.3967    -0.1384    -0.4843
3   0.8566     0.4300     0.9894
4   0.3361    -0.2670    -0.5919
5   -0.1935    0.3000     0.3927
6   0.2673    -0.6787     0.0901
7
8 Correct scores:
9   0.3967    -0.1384    -0.4843
10  0.8566     0.4300     0.9894
11  0.3361    -0.2670    -0.5919
12  -0.1935    0.3000     0.3927
13  0.2673    -0.6787     0.0901
14
15 Difference between your scores and correct scores:
16 3.291676e-08

```

Pas de soucis pour cette étape à priori, l'écart de valeurs est négligeable.

5.2.2 Calcul de la perte

Utilisation de la fonction softmax Nous avons (comme pour le SVM et le Softmax), obtenu un score en sortie de notre réseau. Il nous faut alors exprimer la perte représentée par ce score, ie. une mesure de l'écart de ce dernier à la réalité.

Le choix pris ici est de réutiliser comme score un softmax. Pour mémoire, ce dernier est donné par :

$$\mathcal{L}_i = -\log \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

Où s_j désigne les scores obtenus en sortie du réseau et y_i la classe théorique de l'objet.

Dans la mesure où nous reprenons une implémentation déjà existante, le code est quasiment identique à celui vue dans la section sur le softmax, si ce n'est quelques changements de noms de variables.

```

1 sum_by_images = log(sum(exp(scores),2));
2 % Calcul des termes hors normalisation
3 prov_ = eye(max(y));
4 theoric_labels = prov_(y,:); % 1 si c'est le bon label, 0 sinon
5 theoric_scores = theoric_labels.*scores; % scores des labels théoriques
6 % On combine le tout ensemble
7 data_loss = sum(-sum(theoric_scores,2) + sum_by_images)/N;

```

Terme de régularisation La régularisation change elle légèrement, car il y a désormais deux termes à prendre en compte. Cette dernière provient des termes W_1 et W_2 dont on désire contraindre la norme \mathcal{L}_2 . Ainsi, le terme de régularisation est visé à contraindre la somme de ces normes, soit :

$$\frac{1}{2} \cdot \gamma \cdot (\|W_1\|_2^2 + \|W_2\|_2^2)$$

Comme pour le softmax et le SVM, la fraction sert à avoir un coefficient 1 pour les dérivées. Nous en désuisions donc l'expression complète de la perte pour notre réseau :

$$\mathcal{L}_i = -\log \left(\frac{e^{s_i}}{\sum_j e^{s_j}} \right) + \frac{1}{2} \cdot \gamma \cdot \left(\|W_1\|_2^2 + \|W_2\|_2^2 \right)$$

Du point de vue du code, ceci se traduit en réutilisant des éléments similaires à ceux déjà abordés :

```
1 12_reg = .5*reg*sum(sum(model.W1' .* model.W1')) + .5*reg*sum(sum(model.W2' .* model.W2'))
   );
2 loss = data_loss + 12_reg;
```

Résultats obtenus

En enchaînant ces deux parties, nous obtenons alors le calcul de notre perte. Ce dernier peut être comparé à une valeur de référence prise (puisque la situation n'est pas réellement aléatoire). Les résultats obtenus sont alors les suivants :

```
1 Difference between your loss and correct loss:
2 2.220446e-16
```

Le code obtient une différence négligeable à la valeur fournie, signe que le calcul est cohérent.

5.3 | Passe backward

Nous avons à cette étape obtenu une expression de la perte en sortie du réseau. Il nous faut alors propager cette information « dans l'autre sens » pour calculer les gradients. Ces gradients seront utilisés ultérieurement dans une descente stochastique pour mettre à jour les poids du réseau.

5.3.1 Principe de la rétropropagation du gradient

Règle de chaînage L'idée pour remonter le calcul du gradient est d'utiliser la règle de chaînage. Cette dernière indique que pour une fonction f qui dépend de q qui dépend elle-même de y , nous pouvons écrire (sous des hypothèses acceptables...) :

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

Il est ainsi possible de faire remonter l'information étape par étape dans le réseau jusqu'à arriver à son entrée. Il nous faut donc pour cela expliciter les fonctions présente dans le réseau.

Quelles dérivées partielles calculer? Nous reprendrons tout au long de cette partie les notations présentées à la Figure 9. Il est à noter pour la bonne compréhension des calculs qui suivent que la couche cachée réalise en fait deux opérations. Dans un premier temps elle réalise $z_1 = W_1 X + b_1$, puis dans un second temps applique la fonction ReLU $rz_1 = \text{ReLU}(z_1) = \max(z_1, 0)$ (en effet, ces deux étapes n'ont pas été explicitement séparées dans cette figure).

Nous allons dans la suite nous intéresser aux calculs pour une entrée x_i donnée.

Ainsi, dans ce réseau, remonter l'information jusqu'à l'entrée du réseau revient à calculer $\frac{\partial \mathcal{L}_i}{\partial x}$. Dans les faits, les éléments qui vont nous intéresser sont la mise à jour des poids du réseau, i.e. des vecteurs $W_1; W_2; b_1$ et b_2 . Cette opération va alors nécessiter de remonter la valeur de \mathcal{L}_i jusqu'à leurs niveaux respectifs dans le réseau, il nous faut donc calculer :

$$\frac{\partial \mathcal{L}_i}{\partial W_2} \quad \frac{\partial \mathcal{L}_i}{\partial b_2} \quad \frac{\partial \mathcal{L}_i}{\partial W_1} \quad \frac{\partial \mathcal{L}_i}{\partial b_1}$$

Calcul de $\frac{\partial \mathcal{L}_i}{\partial W_2}$ Cette dérivée est une des plus simples à calculer. En utilisant la règle de chaînage, nous pouvons écrire :

$$\frac{\partial \mathcal{L}_i}{\partial W_2} = \frac{\partial \mathcal{L}_i}{\partial s} \frac{\partial s}{\partial W_2}$$

La quantité $\frac{\partial \mathcal{L}_i}{\partial s}$ correspond en fait à la dérivée du softmax précédemment calculée. Nous pouvons donc écrire :

$$\nabla_s \mathcal{L}_i = \frac{\partial \mathcal{L}_i}{\partial s} = \frac{e^{s_{y_i}}}{\sum_{j=1}^L e^{s_j}} - [\mathbb{1}_{y_i=j}]_{1 \leq j \leq L}$$

Nous retrouvons à peu de chose près la relation (6), si ce n'est que la dérivée est calculée à partir du score et non pour un élément du vecteur de poids (d'où l'absence du x_i et l'utilisation d'une matrice d'indicatrice).

La deuxième dérivée partielle est immédiate :

$$\frac{\partial s}{\partial W_2} = \frac{\partial}{\partial W_2} (W_2 r z_1 + b_2) = W_2$$

Nous en déduisons donc que la dérivée cherchée est donnée par la relation (7).

$$\frac{\partial \mathcal{L}_i}{\partial W_2} = \left(\frac{e^{s_{y_i}}}{\sum_{j=1}^L e^{s_j}} [\mathbb{1}_{y_i=j}]_{1 \leq j \leq L} \right) \times W_2 = \nabla_s \mathcal{L}_i \times W_2 \quad (7)$$

Calcule de $\frac{\partial \mathcal{L}_i}{\partial b_2}$ Ce calcul est très similaire au précédent. Nous pouvons écrire :

$$\frac{\partial \mathcal{L}_i}{\partial b_2} = \frac{\partial \mathcal{L}_i}{\partial s} \frac{\partial s}{\partial b_2}$$

Le premier terme de la multiplication est déjà connu, il reste à estimer le second qui vaut :

$$\frac{\partial s}{\partial b_2} = 1$$

Le résultat est alors donné par la relation (8).

$$\frac{\partial \mathcal{L}_i}{\partial b_2} = \nabla_s \mathcal{L}_i \quad (8)$$

Calcul de $\frac{\partial \mathcal{L}_i}{\partial W_1}$ Il s'agit ici de remonté à la première couche cachée. Entre cette couche et celle dont nous venons de calculer les gradient, ce situent deux opérations : l'application de la fonction ReLU et la régression $W_1 x + b_1$. Ainsi, deux dérivées partielles s'ajoutent à notre chaînage :

$$\frac{\partial \mathcal{L}_i}{\partial W_1} = \frac{\partial \mathcal{L}_i}{\partial s} \frac{\partial s}{\partial r z_1} \frac{\partial r z_1}{\partial z_1} \frac{\partial z_1}{\partial W_1}$$

Dans l'ordre, les trois termes non encore calculés qui apparaissent sont :

$$\begin{cases} \frac{\partial s}{\partial r z_1} = W_2 \\ \frac{\partial r z_1}{\partial z_1} = \mathbb{1}_{z_1 > 0} & \text{cf. les explications avec le SVM} \\ \frac{\partial z_1}{\partial W_1} = x_i \end{cases}$$

Le résultat final (9) s'obtient en multipliant ces expressions :

$$\frac{\partial \mathcal{L}_i}{\partial W_1} = \nabla_s \mathcal{L}_i \times W_2 \times \mathbb{1}_{z_1 > 0} \times x_i \quad (9)$$

Calcul de $\frac{\partial \mathcal{L}_i}{\partial b_1}$ Le calcul est similaire à celui pour W_1 :

$$\frac{\partial \mathcal{L}_i}{\partial b_1} = \frac{\partial \mathcal{L}_i}{\partial s} \frac{\partial s}{\partial r_{z1}} \frac{\partial r_{z1}}{\partial z_1} \frac{\partial z_1}{\partial b_1}$$

Seul le dernier terme n'est pas encore calculé, mais sa valeur est immédiate :

$$\frac{\partial z_1}{\partial b_1} = 1$$

Le résultat final s'écrit alors (10).

$$\frac{\partial \mathcal{L}_i}{\partial b_1} = \nabla_s \mathcal{L}_i \times W_2 \times \mathbb{1}_{z_1 > 0} \quad (10)$$

Régularisation Pour terminer, la dernière étape est de ne pas oublier les termes de régularisation. Nous devons ainsi ajouter ce terme sur les gradients appropriés, soit :

$$\begin{aligned} \frac{\partial \mathcal{L}_i}{\partial W_1} &= \nabla_s \mathcal{L}_i \times W_2 \times \mathbb{1}_{z_1 > 0} \times x_i + \gamma W_2 \\ \frac{\partial \mathcal{L}_i}{\partial W_2} &= \nabla_s \mathcal{L}_i \times W_2 + \gamma W_1 \end{aligned}$$

En effet, les termes de régularisation auront des dérivées nulles sur les autres dérivées partielles.

5.3.2 Vectoriser les calculs

Comme nous l'avons précisé, les calculs précédents ont été montrés pour une entrée x_i . Cependant, l'objectif de notre implémentation étant l'efficacité, il nous faut vectoriser ces derniers.

Les dérivées partielles par rapport aux poids se vectorisent immédiatement, en multipliant juste par la matrice des inconnues à la place de prendre une seule d'entre elle (nous rajoutons juste des colonnes à la matrice utilisée pour multiplier, ce qui est sans incidence sur le calcul, le résultat ayant juste plus de colonnes).

Pour les dérivées partielles par rapport aux biais b_1 et b_2 cependant le calcul est moins immédiat. En effet, les dérivées donnaient des résultats unitaires, or nous ne pouvons garder une matrice de plus d'une colonne pour ces vecteurs. Le calcul exact revient alors à exécuter une sommation sur les lignes des matrices obtenues.

Enfin, la vectorisation va nous demander de diviser les résultats obtenus par le nombre de variables en entrée pour obtenir une moyenne sur nos résultats.

5.3.3 Implémentation de la rétropropagation

Nous disposons désormais de nos formules et de la manière de vectoriser le calcul. Nous rappelons que dans le code initial, nous avions nommé z_1 « *hidden_layer* » (terme nom repris pour les calculs ci-dessus car moins lisible). Nous allons alors construire le code pas à pas en partant de la dérivée de la perte.

Implémentation de la dérivée de la perte La dérivée de la perte est notée *delta_L* et reprend le code réalisé pour le softmax à quelques détails près pour s'adapter aux formats des matrices (cf. le dénominateur de la fraction). Nous rappelons également que le vecteur *theoric_labels* contient des 1 pour les bons labels des objets et des 0 ailleurs, il est défini au moment de calculer la perte. Le code est alors le suivant :

```

1 % Calcul du gradient sur le softmax (en sortie)
2 delta_L = exp(scores)./(sum(exp(scores),2)*ones(1,size(model.b2,2)));
3 delta_L = delta_L - theoric_labels;

```

Implémentation des gradients de W_2 et b_2 Les formules pour ces éléments étaient données en (7) et (8) et s'implémentent immédiatement :

```

1 % Gradient pour W2 et b2
2 grads.W2 = hidden_layer'*delta_L;
3 grads.b2 = sum(delta_L); % taille 1x3 (trois classes)

```

On remarquera la sommation pour b_2 qui avait été mentionnée.

Remontée de la fonction ReLU Ce terme correspond à la dérivée partielle $\frac{\partial \mathcal{L}}{\partial z_1}$ qui est utilisée dans les deux relations (9) et (10). Pour mémoire, ce terme est donné par :

$$\frac{\partial \mathcal{L}}{\partial z_1} = \nabla \mathcal{L} \times W_2 \times \mathbb{1}_{z_1 > 0}$$

Le code s'en déduit donc facilement, en se souvenant que z_1 est représenté par la variable `hidden_layer`.

```

1 % Back-propagation de ReLU
2 grad_hidden = delta_L*model.W2';
3 grad_hidden(hidden_layer<=0) = 0;

```

Implémentation des gradients de W_1 et b_1 On reprend ici les relations (9) et (10). L'implémentation suit la même idée que celle vue pour W_2 et b_2 (sommation pour b_1 et juste une multiplication matricielle pour W_1). Nous avons ainsi le code suivant :

```

1 % Gradient pour W1 et b1
2 grads.W1 = X'*grad_hidden;
3 grads.b1 = sum(grad_hidden);

```

Moyenne et régularisation Pour terminer, nous devons tout d'abord moyenner les valeurs obtenues. Ceci est réalisé ainsi :

```

1 % Moyenner les valeurs
2 grads.W2 = grads.W2/N;
3 grads.b2 = grads.b2/N;
4 grads.W1 = grads.W1/N;
5 grads.b1 = grads.b1/N;

```

Puis pour conclure ces calculs ajouter les termes de régularisation :

```

1 % Ajout de la régularisation
2 grads.W2 = grads.W2 + reg*model.W2;
3 grads.W1 = grads.W1 + reg*model.W1;

```

Ceci finit ainsi le codage de la rétropropagation du gradient dans le réseau.

5.3.4 Test de l'implémentation

Nous pouvons désormais tester l'implémentation du gradient sur les données pour vérifier si le code est conforme aux attentes.

Nous avons dans notre cas obtenu le résultat suivant :

```

1 W2 max relative error :1.819562e-02
2 b2 max relative error :7.103697e-04
3 W1 max relative error :7.464561e-01
4 b1 max relative error :7.352978e-02

```

Les résultats sont un peu moins bons que ceux proposés dans le sujet, nous n'avons cependant pas réussi à faire mieux... De plus, nous verrons pas la suite que malgré cela les résultats finaux sont plutôt satisfaisant (dans les intervalles annoncés).

5.4 | Entraînement du réseau

Si nous résumons le travail actuel sur le réseau jouet, nous avons les éléments suivants :

- La passe *forward* qui va calculer les scores et la perte en utilisant une fonction softmax.
- La passe *backward* qui va calculer les gradients de la perte en fonction des différents coefficients du modèles (W_1 ; W_2 ; b_1 et b_2).

L'étape suivant est donc d'utiliser ces résultats pour entraîner le modèle, ie. faire évoluer les valeurs des coefficients pour améliorer la perte d'étape en étape.

5.4.1 Descente stochastique du gradient

Comme précédemment, nous allons utiliser cette méthode pour faire évoluer les valeurs de nos paramètres. Le code réalisé est alors extrêmement similaire à ceux déjà vus.

Nous commençons pour cela par calculer un batch de données par itération via le code suivant :

```
1 random_indexes = randsample(1:num_train, batch_size);
2 X_batch = X(random_indexes,:);
3 y_batch = y(random_indexes,:);
```

Ensuite, nous calculons la perte et les gradients sur l'ensemble X_batch . Les résultats sont alors réutilisés pour mettre à jour les coefficients selon la relation (W représente un des quatre paramètres) :

$$W = W - \alpha \times \frac{\partial \mathcal{L}}{\partial W}$$

Dans cette relation, α représente la vitesse d'apprentissage, ie. l'importance de la mise à jour de W par rapport à sa valeur précédente.

Nous allons donc réutiliser ici les quatre gradients précédemment calculés et stockés dans le dictionnaire *grads*.

```
1 fields = fieldnames(grads);
2 for i = 1:length(fields)
3     grad = grads.(fields{i});
4     model.(fields{i}) = model.(fields{i}) - params.learning_rate*grad;
5 end
```

5.4.2 Test du réseau

Les paramètres proposés pour l'entraînement sont les suivants :

```
1 params.learning_rate = 1e-1;
2 params.reg = 1e-5;
3 params.num_iters = 100;
4 params.verbose = 0;
```

Nous remarquons un taux d'apprentissage qui semble petit vis-à-vis de nos données, de même pour la régularisation. On remarque que le modèle sera appris en 100 itérations.

Les résultats obtenus sont alors les suivants :

```
1 Final training loss: 0.083948
```

De plus, l'évolution de la perte au cours des itérations vous est proposée à la Figure 10.

On remarque une décroissance très rapide lors des premières itérations pour la perte. Le fait étonnant ici est de finir avec une allure de courbe qui est celle attendue, mais un *loss* final différent. La raison vient des calculs des gradients qui fournissaient un résultat différent. Nous n'en avons cependant pas trouvé la cause.

En particulier, nous avons cru à moment que cela pouvait venir d'instabilités numériques dans la division des exponentielles, et avons donc modifié cette dernière pour « limiter la casse » avec le code suivant :

```
1 max_score = max(max(scores));
2 delta_L = exp(scores-max_score*ones(size(scores)))./(sum(exp(scores-max_score*ones(
    size(scores))),2)*ones(1,size(model.b2,2)));
3 delta_L = delta_L - theoric_labels;
```

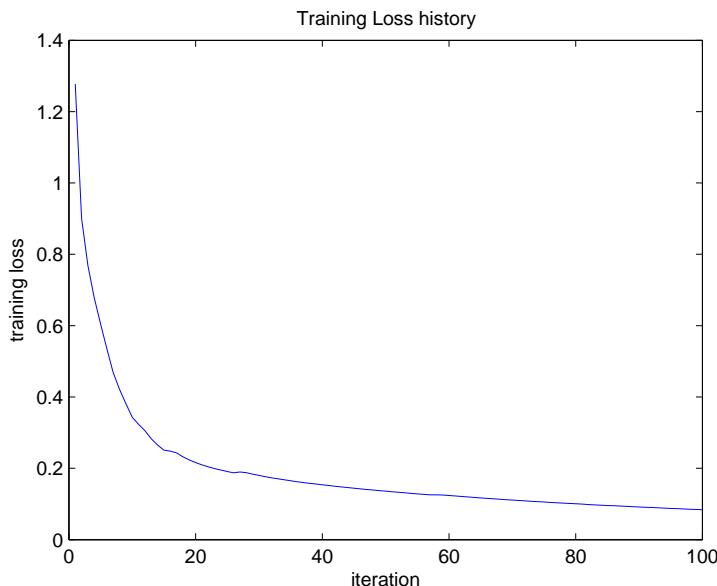


FIGURE 10 • Évolution de la perte au cours de l'entraînement de notre réseau

Le résultat est cependant resté inchangé (mais nous avons laissé cette version du code pour illustrer cette méthode présentée un peu plus tôt).

5.5 | Vraies données et vrai réseau

Notre réseau jouet fonctionnant, nous pouvons alors nous attacher à agrandir ce dernier et le tester sur nos images. Par agrandissement du réseau, nous entendons ici d'augmenter le nombre de neurones par couches, mais *pas* le nombre de couches.

Dans la mesure où nous données disposent désormais de $32 \times 32 \times 3 = 3072$ paramètres, la couche d'entrée sera directement plus imposante. Nous proposons alors d'utiliser 50 neurones pour la couche du milieu et 10 neurones en sortie (puisque il y a 10 classes).

La structure est donc la même que celle de la Figure 9, avec les valeurs suivantes pour les variables proposées :

`num_inputs : 3072` `hidden_size : 50` `num_classes : 10`

5.5.1 Chargement des données

Le chargement des données se fait à l'aide de la méthode `prepare_net_dataset`. Cette dernière va comme d'habitude charger les données de CIFAR10 en mémoire. Elle va également comme dans le cas du SVM soustraire l'image moyenne de toutes les images.

Il est noter que l'exécution de cette méthode sur mon ordinateur personnel est *longue* et à tendance à bloquer ce dernier pendant xx minutes sans rien pouvoir faire.

Quoiqu'il en soit, le résultats en terme d'ensemble de données fournis est le suivant :

```

1 Training data shape:
2     49000      3072
3
4 Training labels shape:
5     49000       1
6
7 Validation data shape:
8     1000      3072
9

```

```
10 Validation labels shape:  
11      1000           1  
12  
13 Test data shape:  
14      1000        3072  
15  
16 Test labels shape:  
17      1000           1
```

Nous retrouvons les ensembles que nous avions lors des parties précédentes.

5.5.2 Entraînement du réseau

Aucun code n'est à réalisé, dans la mesure où ce « gros » réseau fonctionne de la même manière que le réseau jouet des sections précédentes, la seule différence étant sa taille. Attardons-nous cependant sur les hyperparamètres proposés en entrée du réseau, dont les valeurs sont rappelées ci-dessous :

```
1 params.num_iters = 1000;  
2 params.batch_size = 200;  
3 params.learning_rate = 1e-4;  
4 params.learning_rate_decay = 0.95;  
5 params.reg = 0.5;  
6 params.verbose = 1;
```

Le réseau étant plus important, on remarque que 100 itérations auraient certainement été insuffisant pour l'entraîner proprement. Ainsi, nous utiliserons ici à la place 1000 itérations. La taille des batchs à chaque itération est elle assez faible comparée à la taille de l'ensemble d'apprentissage qui est de 49000 images. Ainsi, nous évitons le risque de faire régulièrement des tirages trop similaires lors des calculs de gradients, ce qui aide à consolider le modèle.

Le taux d'apprentissage est similaire à celui du petit modèle, ce qui n'est pas le cas de régularisation qui est elle bien plus importante (il y a plus de données, il faut donc que chaque donnée compte).

Enfin, nous remarquons l'apparition d'un nouveau paramètre, le taux de décroissance du taux d'apprentissage. L'idée est ici de multiplier le taux d'apprentissage par cette valeur pour faire baisser étape à étape sa valeur afin de ne pas trop apprendre en permanence et de finir par atteindre un équilibre.

Une fois l'entraînement lancé, nous obtenons la valeur de la précision sur l'ensemble de validation (ie. pas l'ensemble utilisé en apprentissage !) :

```
1 Validation accuracy: 0.305000
```

Cette valeur est assez faible, avec un modèle qui obtient le bon résultat uniquement 30% du temps, nous allons donc chercher à en savoir un peu plus.

5.5.3 Évaluer les problèmes

La solution proposée est d'afficher l'évolution au cours du temps de la perte, et des précisions sur les ensembles de tests et d'apprentissage.

Une première solution est de regarder les valeurs données par chaque itération par l'algorithme. Ces dernières sont proposées ci-dessous :

```
1 iteration 100 / 1000: loss 2.302552  
2 iteration 200 / 1000: loss 2.297252  
3 iteration 300 / 1000: loss 2.273685  
4 iteration 400 / 1000: loss 2.181215  
5 iteration 500 / 1000: loss 2.067076  
6 iteration 600 / 1000: loss 2.078757  
7 iteration 700 / 1000: loss 2.060523  
8 iteration 800 / 1000: loss 1.978652  
9 iteration 900 / 1000: loss 1.955659  
10 iteration 1000 / 1000: loss 1.887089
```

Nous remarquons déjà que la perte diminue au fur et à mesure du temps, ce qui est « rassurant ». D'un point de vue graphique, nous obtenons alors les résultats de la Figure 11.

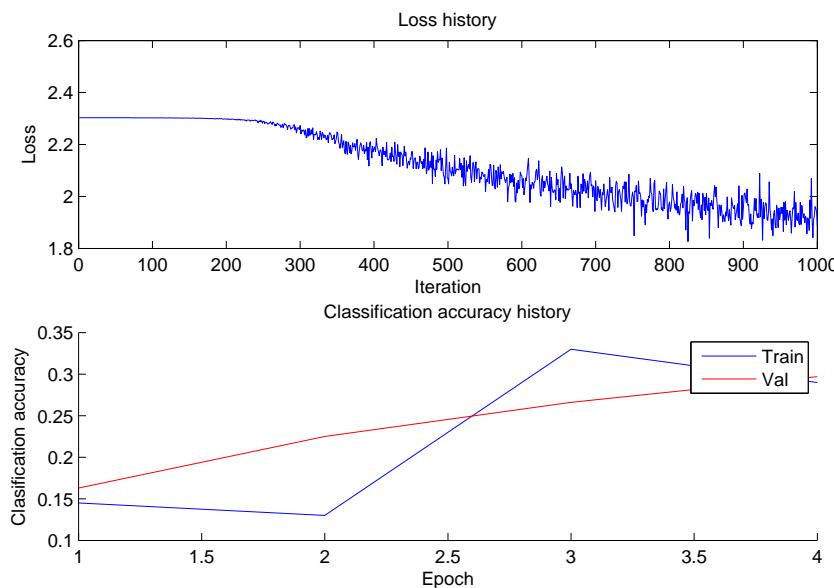


FIGURE 11 • Évolution de la perte et des précisions sur les ensembles d'apprentissage et de test en fonction des itérations

Cette Figure 11 nous apprend les éléments suivants :

- **Évolution du loss** Ce dernier ne décroît qu'au bout de 200 à 250 itérations ! Ainsi, sa décroissance n'est pas échelonnée sur tout l'apprentissage, ce qui peut impliquer le fait que ce dernier soit encore améliorable à la fin de l'exécution. On remarquera de même que la décroissance est bruitée, dans le mesure où la descente de gradient va permettre de s'avancer par itérations successives dans la bonne direction (mais les pas peuvent être trop importants à certains moments et empêcher le résultat, d'où les oscillations).
- **Évolution des précisions** La précision sur l'ensemble de test augmente au fur et à mesure des épisodes, ce qui est un bon point. Concernant celle sur l'ensemble d'apprentissage, nous remarquons une forte hausse pendant le premier temps de la baisse du *loss* avant une petite baisse. Cette baisse n'est pas très importante, mais pourrait être à corriger pour obtenir de meilleurs résultats. On remarque de plus que la différence test-apprentissage n'est pas trop importante, signe que le réseau ne fait pas sur-apprentissage.

Concernant les poids, ces derniers peuvent être observés à la Figure 12.

Nous remarquons qu'un certains nombre d'entre eux sont fortement similaires (par exemple ceux avec l'oval rouge qui font penser à des voitures). Si on compare avec les poids extraits du Softmax, nous retrouvons des motifs communs (ce qui peut sembler logique dans la mesure où nous utilisons la fonction softmax pour estimer la perte...).

Cependant, nous remarquons que certains types d'images semblent moins représentés dans cet apprentissage alors que d'autres semblent sur-représentées (la voiture comme nous l'avons dit par exemple). De plus, nous remarquons que les transformations vis-à-vis des images d'origine ne sont « pas trop importantes ».

5.6 | Optimisation des hyperparamètres

Pour obtenir de meilleurs résultats, nous allons chercher une bonne combinaison d'hyperparamètres. Dans la mesure où nous disposons au total de cinq paramètres, il sera difficile de faire une recherche croisée en temps raisonnable. Nous allons donc regarder l'influence de chaque paramètre et essayer de chercher un bon compromis « à la main ».

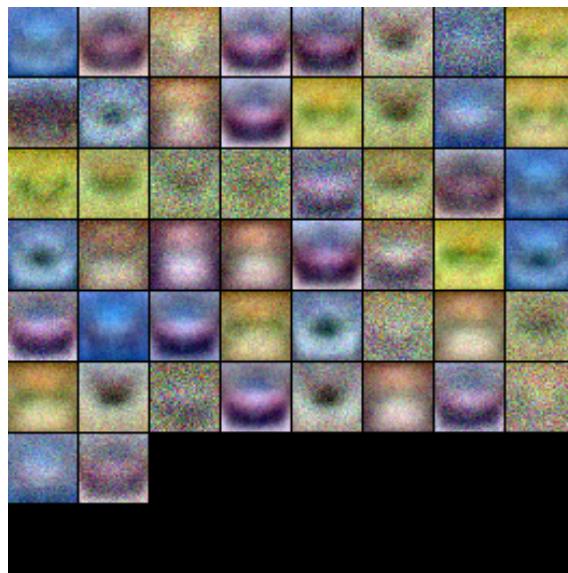


FIGURE 12 • Visualisation des poids appris par le réseau

5.6.1 Les hyperparamètres

Pour ce réseau, nous disposons des hyperparamètres suivants :

- `num_iters` Le nombre d'itérations pour apprendre le modèle.
- `batch_size` La dimension de chaque batch d'apprentissage.
- `learning_rate` Le taux d'apprentissage qui apparaît dans la mise à jour des poids :

$$W = W + \alpha \nabla W$$

- `learning_rate_decay` Le facteur influençant la décroissance du taux d'apprentissage au fur et à mesure des itérations pour essayer de faire converger le réseau.
- `reg` Le facteur de régularisation pour les poids du réseau.
- `hidden_size` Le nombre de neurones dans la couche cachée du réseau.

Nous allons regarder les influences de ces paramètres un à un avant de chercher notre compromis. Le code utilisé sera le même que pour le SVM et le softmax quand nous cherchions à optimiser les valeurs, avec des boucles pour tester les différentes valeurs de paramètres. Nous ne redétaillerons donc pas ce code qui n'est pas d'un grand intérêt ici contrairement aux résultats qu'il fournit.

Les valeurs par défaut utilisées sont les suivantes :

```
1 params.num_iters = 1000;
2 params.batch_size = 200;
3 params.learning_rate = 1e-4;
4 params.learning_rate_decay = 0.95;
5 params.reg = 0.5;
6 params.verbose = 0;
7 hidden_size = 50;
```

5.6.2 Influence de num_iters

Précisions obtenues Pour ce paramètre, nous avons testé les valeurs suivantes :

```
1 num_iterss = [100 500 1000 1500 2000 3000];
```

La limite a été donné par le temps de calcul que nous voulions raisonnable tout en pouvant observer une tendance. Les résultats obtenus sont les suivants :

```
1 Num_iters : 100.000000, validation accuracy: 0.220000
2 Num_iters : 500.000000, validation accuracy: 0.224000
3 Num_iters : 1000.000000, validation accuracy: 0.303000
4 Num_iters : 1500.000000, validation accuracy: 0.341000
5 Num_iters : 2000.000000, validation accuracy: 0.390000
6 Num_iters : 3000.000000, validation accuracy: 0.424000
```

Nous observons que les précisions en validation s'améliorent avec le nombre d'itérations. Ceci est logique dans la mesure où plus d'itérations implique plus de temps pour pouvoir affiner les poids du modèle. Concernant ce paramètre (dont la corrélation avec d'autres paramètre notamment la taille des batchs ou les deux paramètres liés au taux d'apprentissage), nous remarquons ici que plus la valeur est élevée, meilleurs sont les résultats. Cependant, un plateau se dessine, puisque l'on voit que le gain de précision entre 2000 et 3000 itérations est inférieur à celui entre 1500 et 2000.

Règles et valeurs dégagées Ainsi, pour notre meilleur modèle, il pourra être intéressant de s'approcher au mieux de ce plateau, avec par exemple un nombre de 4000 itérations. On remarquera également que comme l'augmentation du nombre d'itérations est monotone croissante, nous pouvons pour les derniers réglages garder la valeur de 1000 itérations et l'augmenter à la dernière exécution pour booster nos performances.

Aspect des courbes À titre indicatif, nous avons aussi calculé les courbes d'évolution de la perte et des précisions en fonction de l'itération lors de l'entraînement des modèles. Nous présentons ainsi trois courbes montrant leur évolution à la Figure 13.

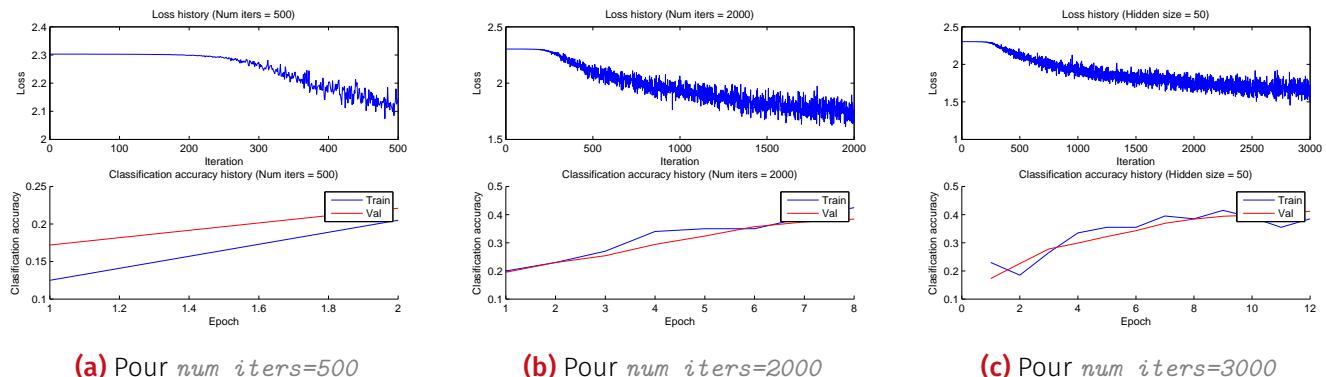


FIGURE 13 • Évolutions de la perte et des précisions en classification au cours de différents entraînements en faisant varier num_iters

5.6.3 Influence de batch_size

Précisions obtenues Pour ce paramètre, nous avons testé les valeurs suivantes :

```
1 batch_sizes = [100 200 500 750 1000];
```

La liste de valeurs a été définie en fonction de la valeur fournie par défaut et des valeurs qui nous semblaient « réalistes » pour ce paramètre (dans tous les cas inférieur au nombre de données). Les résultats obtenus sont alors les suivants :

```
1 Batch size : 100.000000, validation accuracy: 0.303000
2 Batch size : 200.000000, validation accuracy: 0.302000
3 Batch size : 500.000000, validation accuracy: 0.299000
4 Batch size : 750.000000, validation accuracy: 0.301000
5 Batch size : 1000.000000, validation accuracy: 0.300000
```

Une remarque qui ne peut se voir ici est la différence de temps d'exécution entre les différentes valeurs. En effet, plus `batch_size` est grand, plus les matrices d'individus à fournir en entrée du réseau sont importantes. Ainsi, nous obtenons un temps de calcul qui croît significativement avec l'augmentation de la valeur de ce paramètre.

Résultats retenus Pour les valeurs testées, nous remarquons des résultats plutôt homogènes, les « meilleures » valeurs étant obtenues vers 200. Cette valeur était celle proposée dans l'énoncé, que nous conserverons donc pour la suite, puisque changer sa valeur n'apporte pas d'amélioration sensible de la précision du modèle, mais peut avoir un impact significatif sur le temps d'exécution.

5.6.4 Influence de `learning_rate`

Précisions obtenues Pour ce paramètre, nous avons choisi de tester une liste de valeurs sensiblement proche de celles testées pour le softmax et le SVM. Les valeurs retenues sont ainsi les suivantes :

```
1 learning_rates = [1e-7, 5e-7, 1e-6, 5e-6, 1e-5, 1e-4, 2e-4, 3e-4, 5e-4, 8e-4, 1e-3, 5
e-3];
```

Nous avons alors obtenu les résultats suivants pour la précision sur l'ensemble de validation :

```
1 Learning rate : 0.000000, validation accuracy: 0.124000
2 Learning rate : 0.000000, validation accuracy: 0.100000
3 Learning rate : 0.000001, validation accuracy: 0.143000
4 Learning rate : 0.000005, validation accuracy: 0.231000
5 Learning rate : 0.000010, validation accuracy: 0.194000
6 Learning rate : 0.000100, validation accuracy: 0.299000
7 Learning rate : 0.000200, validation accuracy: 0.380000
8 Learning rate : 0.000300, validation accuracy: 0.419000
9 Learning rate : 0.000500, validation accuracy: 0.438000
10 Learning rate : 0.000800, validation accuracy: 0.476000
11 Learning rate : 0.001000, validation accuracy: 0.471000
12 Learning rate : 0.005000, validation accuracy: 0.087000
```

La première remarque est que ce paramètre a une influence extrêmement importante sur les résultats obtenus.

Règles et valeurs dégagées Pour mieux voir les tendances, un résumé graphique des données vous est fourni à la Figure 14.

Nous remarquons ainsi sur cette figure une croissance située pour des valeurs allant de 10^{-7} à 10^{-3} puis une chute brusque des résultats. Ceci peut s'expliquer par le fait que l'on dépasse un seuil au-delà duquel la valeur du taux d'apprentissage est tellement élevée que les sauts réalisés autours de l'optimum des valeurs deviennent trop important et empêchent la convergence. On noter que la précision en validation obtenue dans cette situation fait moins bien qu'un simple modèle aléatoire...

Du point de vue des valeurs, l'idée est donc de réussir à ce position vers la valeur au pic (ici environ pour 10^{-3}), qui est celle pour laquelle on obtiendra la meilleure précision.

Aspect des courbes Pour terminer avec ce paramètre, nous allons regarder l'évolution des courbes donnant l'évolution de la perte et des précisions lors d'un apprentissage. Ces courbes vous sont proposées pour différentes valeurs à la Figure 15.

Sur les différentes courbes de la Figure 15, les résultats sont assez visibles concernant les évolutions. Nous remarquons ainsi :

- **Pour 10^{-6}** On remarque que la décroissance de la perte au cours de l'apprentissage est quasiment inexiste. Ainsi, le modèle n'est jamais réellement optimisé, d'où ses résultats catastrophiques. On

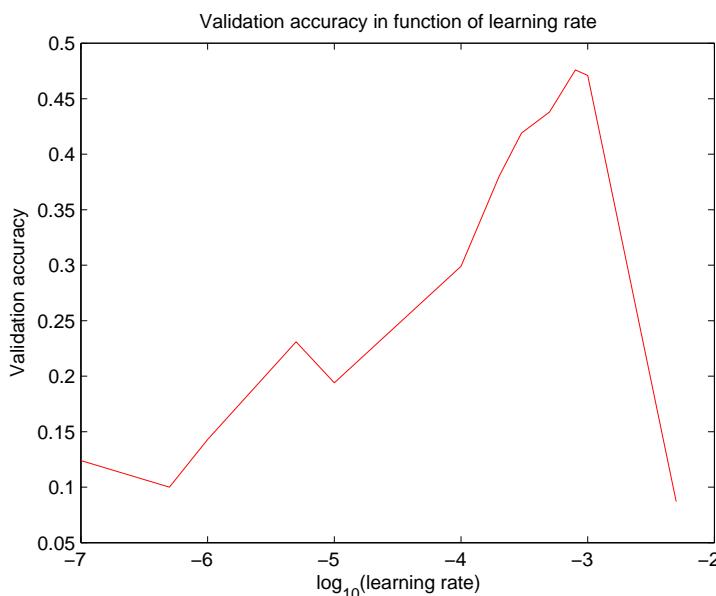


FIGURE 14 • Évolution de la précision en validation pour différentes valeurs de `learning_rate`

remarque une légère augmentation de la précision sur l'ensemble de validation, mais celle-ci n'est pas très représentative, surtout quand l'on s'aperçoit que celle d'apprentissage n'est pas améliorée, or la mise à jour des coefficients est censée la garantir.

Ces courbes montrent donc clairement que l'apprentissage réalisé n'est pas bon.

- *Pour 2×10^{-4}* Pour cette valeur, on remarque que les courbes changent. Ainsi, la décroissance de la perte est plus marquée, bien que cette dernière soit de courte durée (elle met du temps à démarrer). Cependant, les précisions augmentent tout au long de l'exécution, signe que le réseau apprend bien des données à la différence de l'étape précédente.

Ces courbes montrent donc que l'apprentissage a lieu ici, il est rapide mais assez efficace avec environ 37% de précision pour l'ensemble de validation.

- *Pour 10^{-3}* Cette valeur correspond au meilleur point calculé. Comparé à la valeur 2×10^{-4} , on remarque que la décroissance de la perte commence beaucoup plus vite, et permet donc d'obtenir de bien meilleurs résultats pour la précision.

On remarque aussi que les courbes de précisions sont ici très nettement séparées, avec la précision d'apprentissage qui dépasse de presque 10% le taux obtenu en test. Cependant, ceci n'est pas inquiétant (il n'y a pas de réel effondrement des performances avec l'ensemble de test), et les taux obtenus sont très satisfaisants avec environ 45% en test.

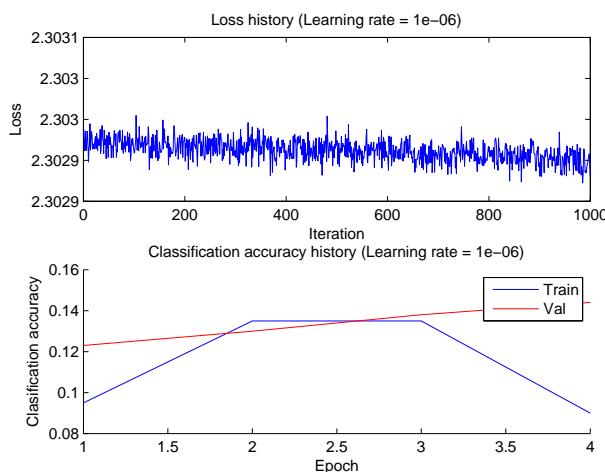
- *Pour 5×10^{-3}* Cette courbe permet de comprendre les performances désastreuses du modèle. En effet, on remarque qu'au lieu de diminuer, la perte s'est envolée... Avec les mains, nous voyons que la bascule se fait lorsque le plateau initial (celui que l'on voyait décroître sur les trois figures précédentes) passe le cap de zéro. Dans ce cas, les pas deviennent trop grand et il n'est plus de possible de bien viser en mettant à jour nos paramètres d'où ce constat.

Nous vous fournissons alors également les meilleurs poids obtenus à la Figure 16.

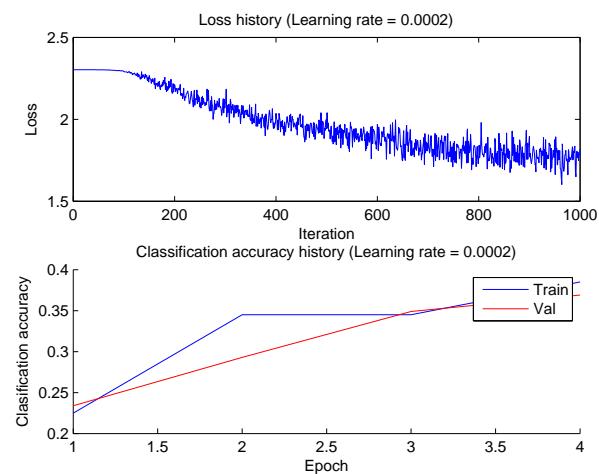
Nous remarquons sur cette figure qu'à la différence des poids vus sur l'apprentissage initial, ces derniers diffèrent plus fortement de ceux d'un Softmax. Ceci semble indiquer que pour les meilleurs `learning_rate`, le réseau arrive à extraire des *features* personnalisées par rapport à ce que ferait un Softmax. Il ne se comporte ainsi pas comme un « Softmax amélioré ».

5.6.5 Influence de `learning_rate_decay`

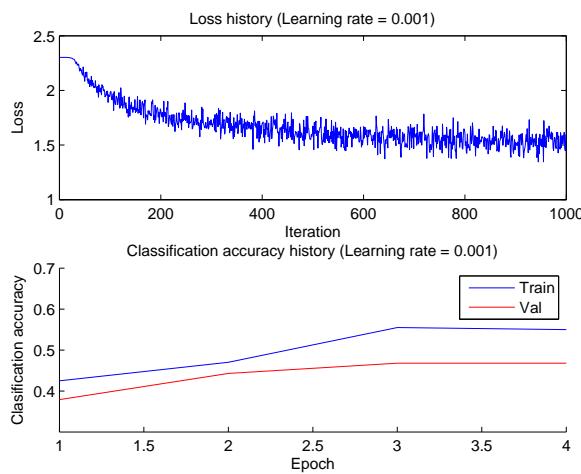
Précisions obtenues Pour ce paramètre, nous avons choisi nos valeurs en fonction de celle par défaut proposée pour voir les évolutions au voisinage de cette dernière :



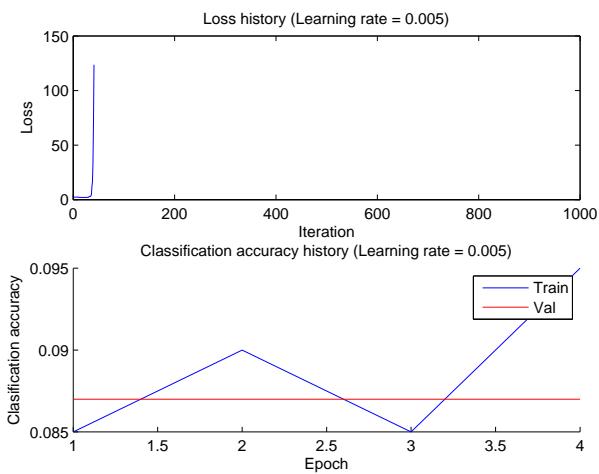
(a) Pour $learning_rate=1e-6$



(b) Pour $learning_rate=2e-4$



(c) Pour $learning_rate=1e-3$



(d) Pour $learning_rate=5e-3$

FIGURE 15 • Évolutions de la perte et des précisions en classification au cours de différents entraînements en faisant varier $learning_rate$

```
1 learning_rate_decays = [.5, .75, .85, .9, .95, .98];
```

Les résultats obtenus sont alors les suivants :

```
1 Learning rate decay : 0.500000, validation accuracy: 0.304000
2 Learning rate decay : 0.750000, validation accuracy: 0.304000
3 Learning rate decay : 0.850000, validation accuracy: 0.308000
4 Learning rate decay : 0.900000, validation accuracy: 0.303000
5 Learning rate decay : 0.950000, validation accuracy: 0.291000
6 Learning rate decay : 0.980000, validation accuracy: 0.292000
```

Nous remarquons que pour ce paramètre, prendre des valeurs faibles ne fournit pas de bons résultats. Ceci s'explique par le fait qu'une valeur trop faible fera trop vite décroître le taux d'apprentissage et qu'ainsi l'apprentissage ne se fera principalement que sur quelques itérations.

Règles et valeurs dégagées Nous observons un maximum vers 0,85, soit un peu moins que la valeur proposée initialement. Nous prendrons donc cette nouvelle valeur.

Aspect des courbes Les valeurs étant proches, les courbes que nous vous présentons habituellement sont sensiblement les mêmes pour toutes les valeurs testées. Nous obtenons ainsi des courbes ayant l'allure de la Figure 17.

Nous retrouvons des courbes où la décroissance du *loss* est lente à démarrer (plus de 200 itérations), ce qui fournit des résultats de l'ordre de 30% de précision en test.

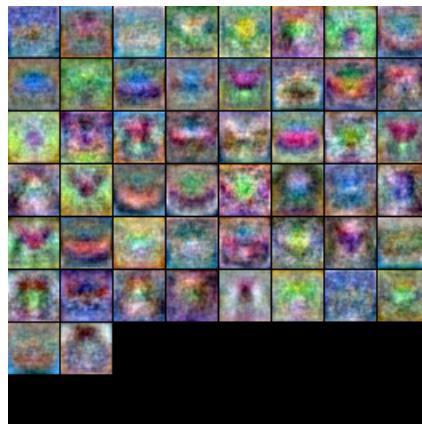


FIGURE 16 • Meilleurs poids obtenus pendant les essais de valeurs sur *learning_rate*

5.6.6 Influence de *reg*

Pour ce paramètre, nous allons d'abord commencer par déterminer un ordre de grandeur acceptable. Pour cela, nous avons testé les différentes puissances de dix sur des modèles :

```
1 regs = [.05, .5, 5, 50, 500, 5000];
```

Nous obtenons alors les résultats suivants :

Le constat est sans, appel, les valeurs doivent être prises inférieures à 1. Nous allons donc dans un second temps tenter d'affiner la détermination de ces valeurs en testant de nouvelles possibilités dans cet intervalle :

```
1 regulations = [.01, .05, .01, .1, .25, .5, .7, .8, .9, .92, .95];
```

Nous obtenons alors les résultats suivants sur des réseaux réinitialisés avant chaque valeur :

```
1 Reg : 0.010000, validation accuracy: 0.298000
2 Reg : 0.050000, validation accuracy: 0.302000
3 Reg : 0.100000, validation accuracy: 0.305000
4 Reg : 0.250000, validation accuracy: 0.298000
5 Reg : 0.500000, validation accuracy: 0.296000
6 Reg : 0.700000, validation accuracy: 0.293000
7 Reg : 0.800000, validation accuracy: 0.283000
8 Reg : 0.900000, validation accuracy: 0.294000
9 Reg : 0.920000, validation accuracy: 0.301000
10 Reg : 0.950000, validation accuracy: 0.290000
```

Nous observons alors des valeurs sensiblement similaires comprises entre 0,28 et 0,30. Les deux valeurs pour lesquelles semblent se concentrer les meilleurs résultats sont autour de 0,5 et autour de 0,92. Ces deux valeurs seront à essayer pour notre meilleur réseau.

Aspect des courbes Nous allons nous intéresser au différents cas extrêmes. Nous avons pour cela utilisé quatre des valeurs précédentes, dont les courbes vous sont proposées Figure 19.

Nous remarquons sur ces figures que la décroissance met de plus en plus de temps à avoir lieu, et est également moins performance, avec des pertes beaucoup plus élevées pour des régularisation élevées. Ceci est logique dans la mesure où une forte régularisation va très fortement limiter les valeurs des coefficients du modèle mais « peu » se soucier de la réelle valeur à prévoir. Nous obtenons donc des modèles peu prédictifs. Pour la valeur de 500 (le cas est identique avec 5000), nous observons même que la décroissance de la perte n'est presque plus d'actualité, nous pouvons donc penser que l'apprentissage ne se préoccupe plus des classes pour ces valeurs, ce qui explique les précisions obtenues de l'ordre de 10%, ie. à presque comme le hasard.

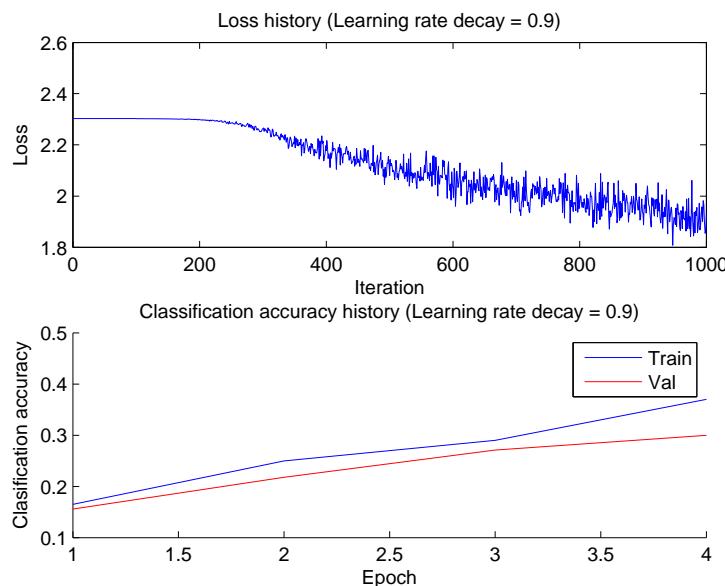


FIGURE 17 • Évolutions de la perte et des précisions en classification au cours de l'entraînement pour $learning_rate_decay=0.9$

```

1 Reg : 0.050000, validation accuracy:
      0.298000
2 Reg : 0.500000, validation accuracy:
      0.307000
3 Reg : 5.000000, validation accuracy:
      0.282000
4 Reg : 50.000000, validation accuracy:
      0.224000
5 Reg : 500.000000, validation accuracy:
      0.107000
6 Reg : 5000.000000, validation accuracy:
      0.112000

```

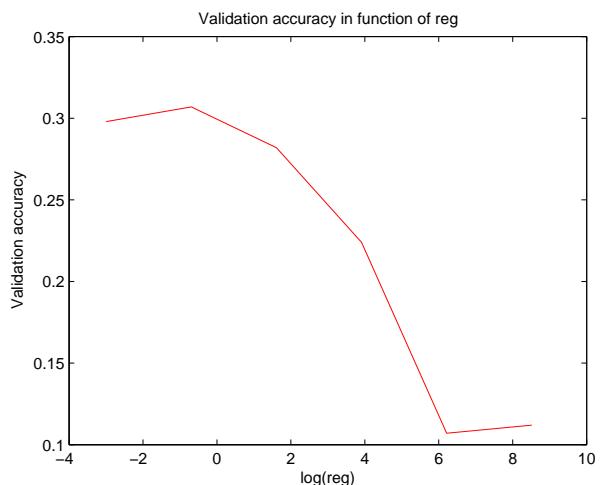


FIGURE 18 • Évolution de la précision en validation pour le paramètre reg sur des puissances de 10

Ainsi, on observe que trop contraindre les coefficients nous fait oublier notre tâche initiale. Mais justement, que deviennent ces coefficients ? Pour ce paramètre, il est alors intéressant de regarder les poids qui sont appris par le modèle, dans la mesure où ce dernier est censé les contraindre. Les résultats pour les mêmes valeurs de reg que la Figure 19 vous sont proposés à la Figure 20.

Le résultat de cette figure corrobore ce que nous avions vu avec la Figure 19. Pour de faibles valeurs de régularisation, nous trouvons des poids assez différents et distincts les uns des autres. Ceci apporte plus de diversité pour la classification, et montre que ces poids reflètent mieux les données.

On remarque alors que plus la régularisation augmente, plus les poids deviennent tous similaires, jusqu'à avoir une presque totale uniformité pour une valeur de 500. Ceci est cohérent dans la mesure où nous imposons une très forte contrainte sur la norme de ces vecteurs, ce qui ne laisse pas beaucoup d'espace et fait tendre leurs valeurs vers une même valeur commune.

Moralité, il ne faut pas prendre une régularisation trop forte si l'on veut que le modèle puisse s'appliquer réellement aux données et ne pas se concentrer uniquement sur la norme de ses coefficients. Il ne faut pas non plus prendre une valeur trop basse, au risque d'avoir une explosion des valeurs des coefficients.

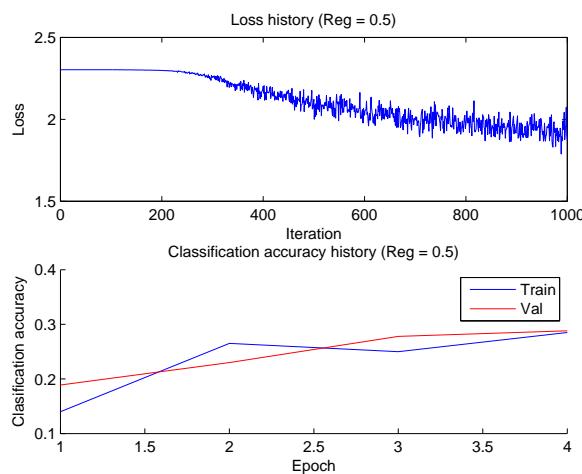
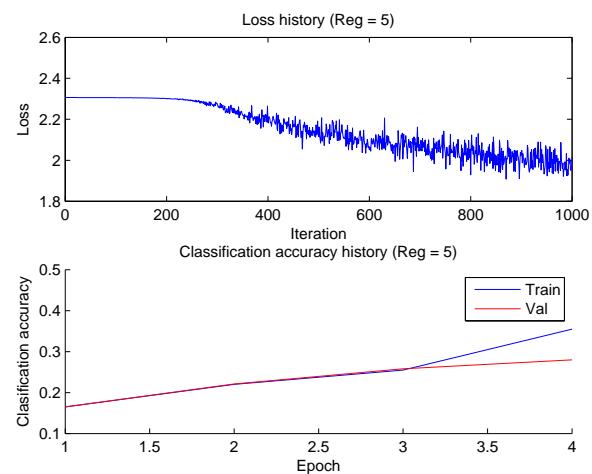
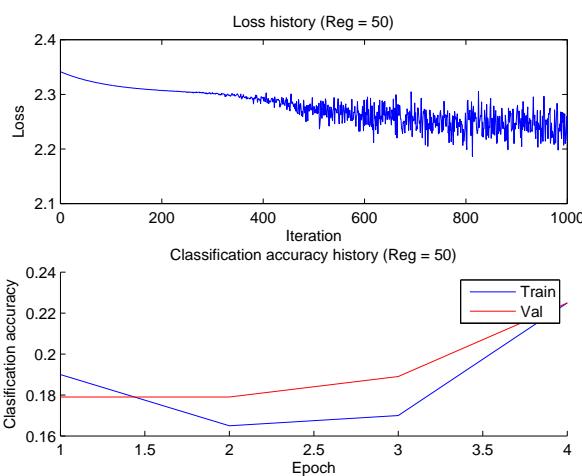
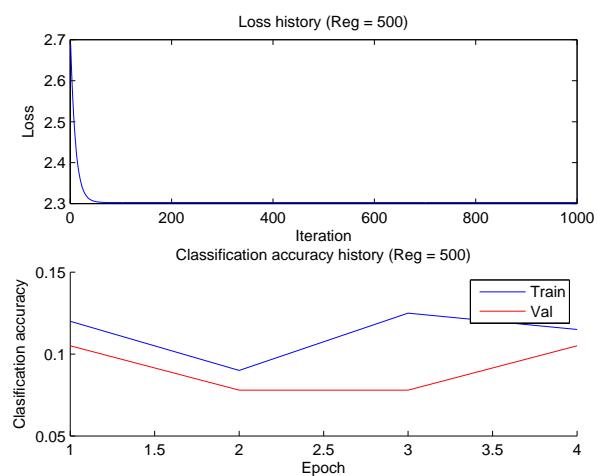
(a) Pour $reg=0.5$ (b) Pour $reg=5$ (c) Pour $reg=50$ (d) Pour $reg=500$

FIGURE 19 • Évolutions de la perte et des précisions en classification au cours de différents entraînements en faisant varier reg

5.6.7 Influence de `hidden_size`

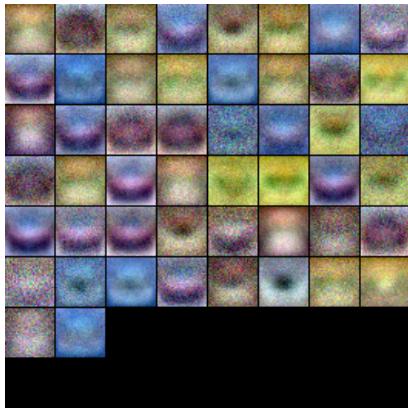
Résultats obtenus Pour ce dernier coefficients, nous avons testé les valeurs suivantes, pour conserver des temps d'exécution raisonnables.

```
1 hidden_sizes = [5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100];
```

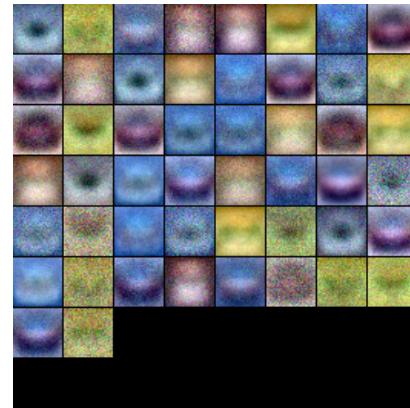
Les résultats en terme de précisions sont alors fournis ci-dessous.

```
1 Hidden size : 5.000000, validation accuracy: 0.269000
2 Hidden size : 10.000000, validation accuracy: 0.273000
3 Hidden size : 20.000000, validation accuracy: 0.281000
4 Hidden size : 30.000000, validation accuracy: 0.280000
5 Hidden size : 40.000000, validation accuracy: 0.287000
6 Hidden size : 50.000000, validation accuracy: 0.303000
7 Hidden size : 60.000000, validation accuracy: 0.292000
8 Hidden size : 70.000000, validation accuracy: 0.303000
9 Hidden size : 80.000000, validation accuracy: 0.309000
10 Hidden size : 90.000000, validation accuracy: 0.309000
11 Hidden size : 100.000000, validation accuracy: 0.302000
```

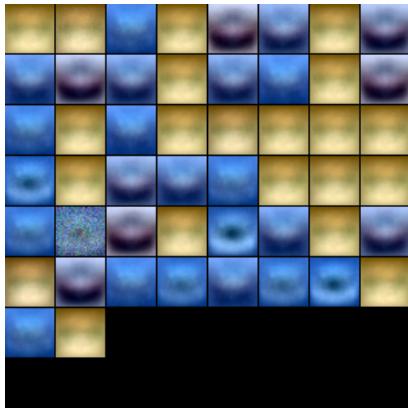
Graphiquement, la tendance générale est un peu plus visible, comme le montre la Figure 21.



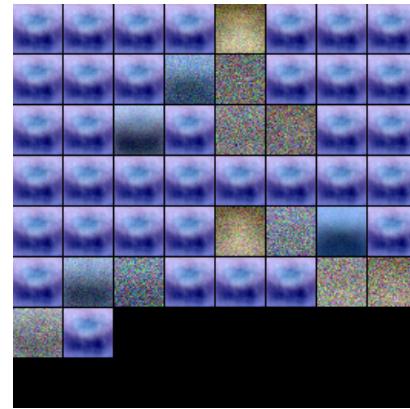
(a) Pour $reg=0.5$



(b) Pour $reg=5$



(c) Pour $reg=50$



(d) Pour $reg=500$

FIGURE 20 • Évolution des poids appris pour différentes valeurs de reg

Le pic au milieu de la courbe peut être assimilé à une valeur aléatoire qui a aidé un modèle pendant l'apprentissage (les différences restant de l'ordre de 1% de précision). Nous remarquons ici qu'augmenter le nombre de neurones dans la couche permet de meilleurs résultats. Cependant, on remarque que les valeurs semblent stagner passé les 70 neurones dans cette couches, signe qu'une valeur trop élevée n'apportera pas nécessairement de bien meilleurs résultats.

Nous remarquons par contre, que prendre un nombre trop faible va diminuer les performances de l'ordre de 4% (ce qui n'est au final pas tant que cela quand on s'aperçoit que ce réseau n'utilise que 5 neurones pour sa couche cachée!).

Valeur retenue Pour notre modèle amélioré, nous pourrons considérer un nombre de 80 neurones dans la couche pour améliorer légèrement nos performances. Une valeur plus élevée ne semblerait pas apporter d'apports majeurs.

Visualisation des courbes Pour cet hyperparamètre, détailler les courbes n'est utile dans la manière où ces dernières vont avoir sensiblement le même aspect. Ce qui peut cependant être intéressant ici est de regarder l'aspect des poids qui ont été entraînés. Nous le proposons pour différentes valeurs de `hidden_state` à la Figure 22.

Nous remarquons sur ces figures que le cas avec cinq neurones dans la couche caché définit en fait les principaux « filtres » que l'on retrouve dans les autres configurations. Ainsi, ce premier réseau va apprendre

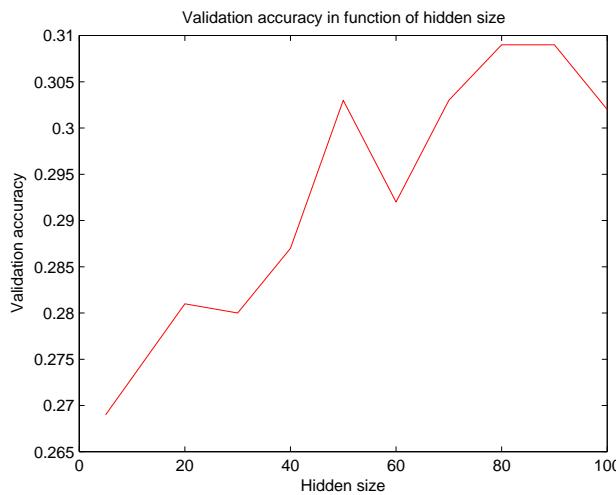


FIGURE 21 • Évolution de la précision pour différentes valeurs de *hidden_size*

les cinq filtres principaux qui permettent à eu seul d'apprendre (dans ce cas) 27% du modèle. Augmenter le nombre de neurone permet alors de dupliquer et affiner ces filtres pour les calculs, mais n'en ajoute pas un nouveau qui soit fondamentalement différent des cinq premiers appris (il y en a qui varient un peu, mais l'idée générale des nouveaux filtres reste la même).

Ceci explique donc pourquoi dans notre cas la forte augmentation du nombre de neurone ne révolutionne pas les performances : en l'état actuel notre réseau n'arrive pas à trouver d'autres filtres intéressant et se contente donc juste de faire varier légèrement ceux « basiques ». En regardant ces filtres pour le cas où il y a 100 neurones, nous pouvons peut être en déclarer 7 catégories distinctes, mais les 10 classes ne semblent pas aussi bien définies les unes que les autres (en particulier celle des voiture est particulièrement visible!).

5.6.8 Choix d'un compromis

Un premier modèle Après tous ces essais, nous pouvons nous lancer dans l'optimisation du modèle. Nous commençons avec le modèle suivant :

```

1 params.num_iters = 1000;
2 params.batch_size = 200;
3 params.learning_rate = 1e-3;
4 params.learning_rate_decay = 0.85;
5 params.reg = .92;
6 params.verbose = 0;
7 hidden_size = 80;
```

Ses résultats sont alors les suivants :

```
1 Validation accuracy: 0.483000
```

Concernant les deux représentations utiles, nous obtenons le résultat de la Figure 23.

En s'inspirant du sujet En regardant le sujet, nous pouvons nous apercevoir que le nombre de neurones de la couche intermédiaire est de 195. Nous avons donc essayé également cette valeur (que nous avions initialement exclu lors de notre étude de *hidden_params*). Nous obtenons alors les résultats suivants :

```
1 Validation accuracy: 0.470000
```

Il réalise donc une moins bonne performance que notre modèle, nous ne le retiendrons pas pour la suite. Pour le plaisir, nous pouvons cependant afficher les poids appris à la Figure 24, qui sont très proches (mais plus nombreux!) que ceux de la Figure 23.

Nous remarquons pour ces derniers qu'ils diffèrent fortement de ce qu'aurait appris un Softmax (même conclusion que pour l'influence de *learning_rate* que nous avions vue. On commence également à

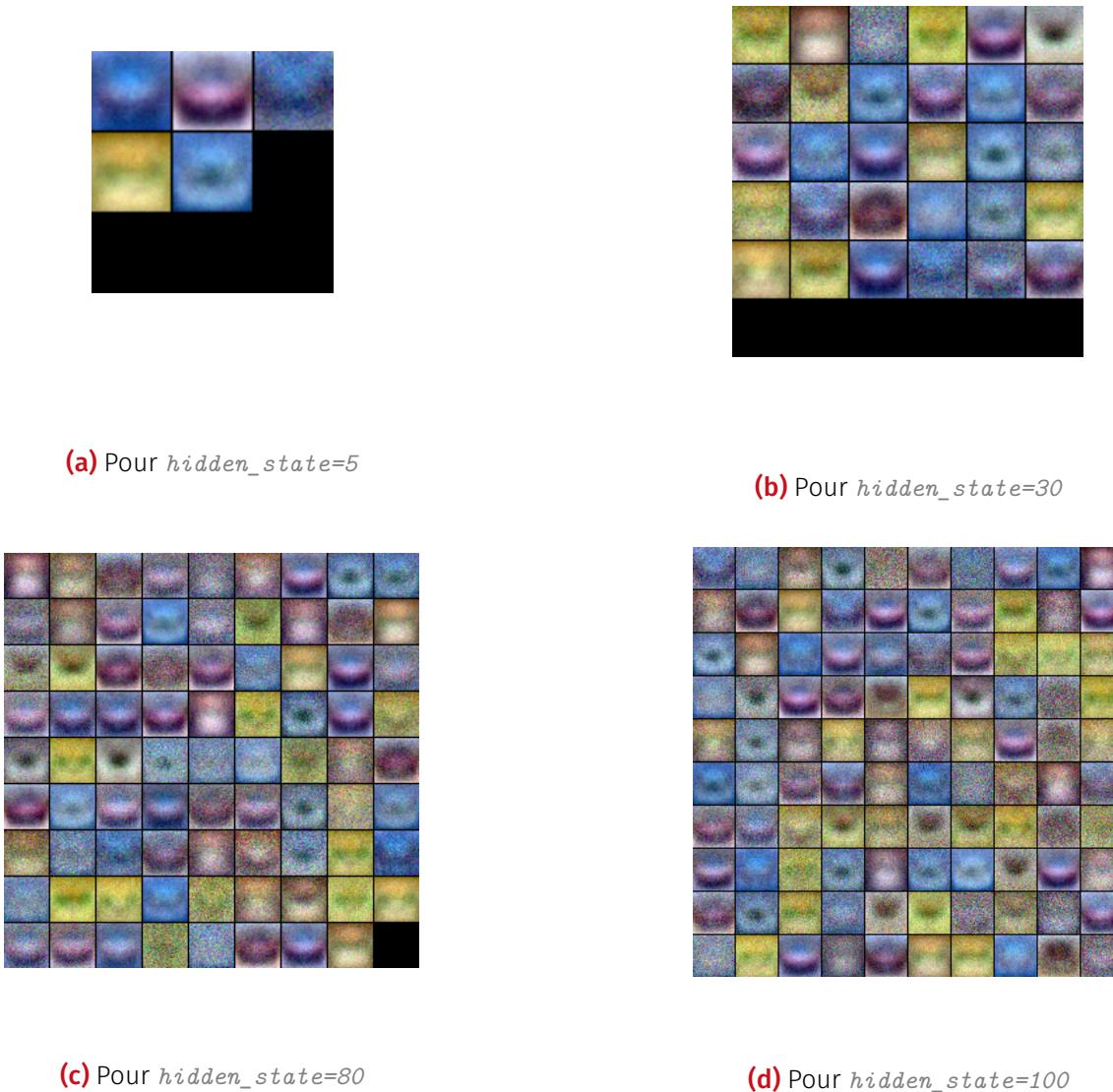


FIGURE 22 • Évolution des poids appris pour différentes valeurs de *hidden_state*

remarquer une plus grande diversité dans les filtres proposés par ces poids. Ceci laisse à penser que plus nous autorisons de neurones dans la couche cachée, et plus le réseau va se permettre d'apprendre des cas « particulier ».

De plus, il est peut être possible d'obtenir d'avoir plus de marge de progression avec ce modèle qu'avec le notre. Cependant, le temps de calcul devient ici plus important étant donné le nombre de coefficients, et nous n'avons malheureusement pas essayé de relancer ce modèle pour 4000 itérations (et peut être que 10000 aurait été plus censé dans la mesure où il y a plus de coefficients à affiner...).

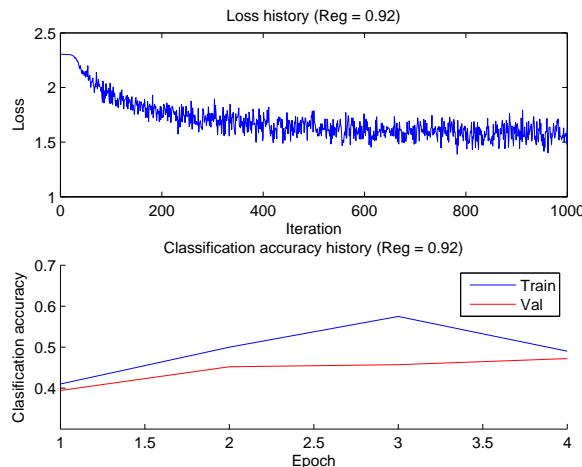
Augmenter le nombre d'itérations Nous reprenons alors notre premier modèle et réalisons comme nous l'avions dit dans la partie sur le nombre d'itérations une nouvelle simulation avec cette fois-ci 4000 itérations. Nous obtenons alors le résultat suivant :

```
Validation accuracy: 0.494000
```

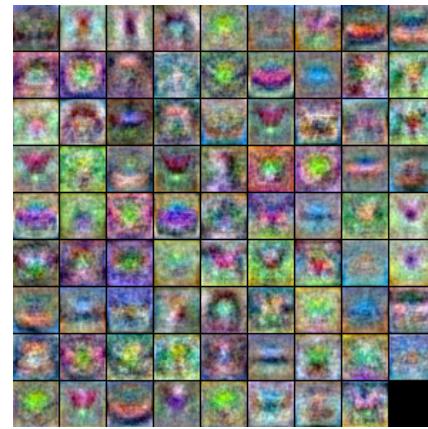
Nous avons donc gagné 1% de précision...Ceci n'est pas parfait, mais nous permet d'être dans les normes proposées. Les courbes obtenues sont alors celles de la Figure 25.

Nous voyons que la courbe d'évolution de la perte commence à décroître directement, nous ne devons donc pas être très loin du point de divergence, mais surtout de notre maximum pour le paramètre *learning_rate*.

L'évolution des pertes est quand à elle dans une phase plutôt stationnaire, ce qui laisserait penser que



(a) Évolution de la perte et des précisions lors de l'apprentissage



(b) Visualisation des poids obtenus pour le modèle optimisé

FIGURE 23 • Résultats obtenus pour le premier modèle optimisé

ce modèle ne pourra guère nous offrir mieux à moins de refaire varier des paramètres.

Enfin, en regardant les poids, nous pouvons faire la même conclusion que pour le modèle à 195 neurones : le réseau a appris plus de cas, et s'est démarqué d'un simple Softmax. Il se permet ainsi un apprentissage plus fin, d'où ses meilleurs résultats.

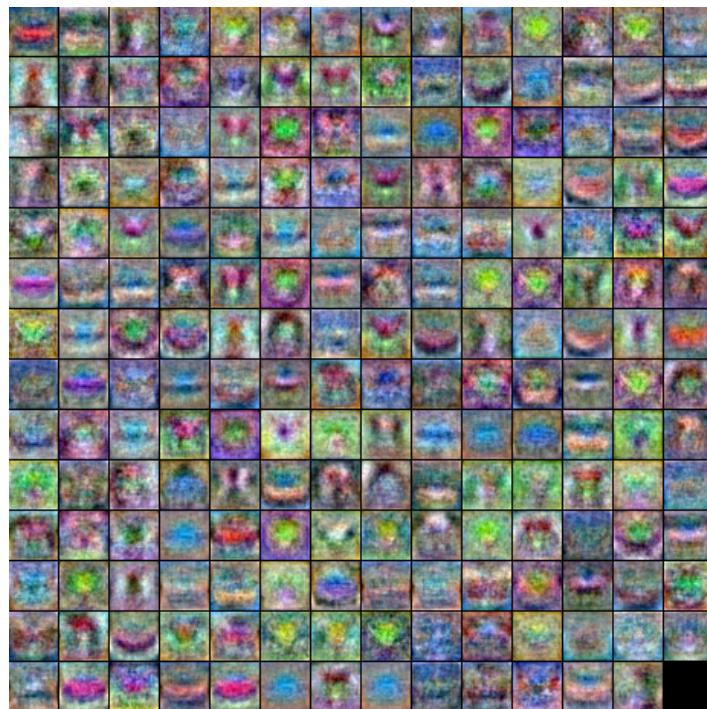
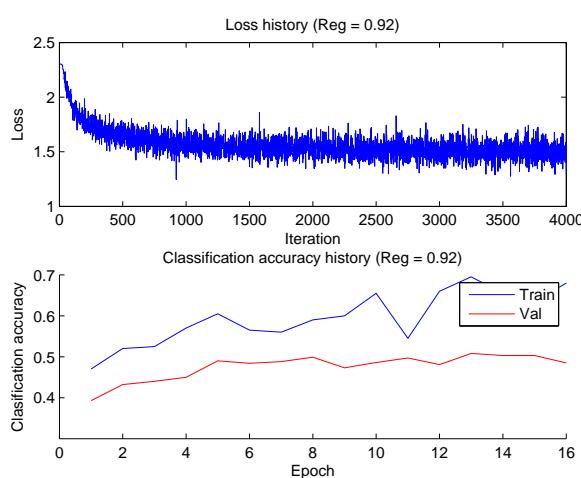


FIGURE 24 • Visualisation des poids obtenus pour un modèle optimisé similaire à celui de l'énoncé



(a) Évolution de la perte et des précisions lors de l'apprentissage



(b) Visualisation des poids obtenus pour le modèle optimisé

FIGURE 25 • Résultats obtenus pour le premier modèle optimisé avec 4000 itérations

Conclusion

Ce travail nous a permis de mieux comprendre la structure et les idées derrière les classifieurs de type KNN, SVM, Softmax et réseaux de neurones. Il a en particulier été l'occasion d'appliquer les résultats théoriques vus d'un point de vue pratique. Nous avons ainsi pu fournir une implémentation de ces quatre classifieurs.

Sans surprise, le classifieur le plus performant est ici le réseau de neurones, nos résultats étant montés jusqu'à 49,5% de précision sur l'ensemble de test. Il devance ainsi largement les résultats après optimisation du KNN (28,2%), du SVM (36,6%) et du Softmax (34%).

Cependant, plus que de montrer les bons résultats du réseau de neurones, ce travail nous a permis de comprendre comment ces autres modèles et les méthodes qui y sont associées ont pu conduire à la création des réseaux de neurones. Nous retrouvons ainsi des idées du Softmax avec l'utilisation de la fonction ReLU ou encore le calcul des scores en utilisant la fonction de coût d'un Softmax sur les sorties du réseau. De même, la méthode de descente stochastique du gradient est un élément commun entre ces trois classifieurs.

De plus, nous avons pu appréhender dans cet exercice l'importance de l'optimisation du code afin de pouvoir traiter des volumes importants de données. Nous avons ainsi vu que si une version itérative d'un algorithme était peut être simple et rapide à écrire, cette dernière ne fournit pas nécessairement de bonnes performances. Au contraire, la vectorisation complète du code, bien que parfois coûteuse du point de vue des calculs théoriques, permet de gagner en efficacité dans les langages le permettant. Nous avons ainsi pu réaliser des calculs qui n'auraient pas été possibles sans cette dernière en temps raisonnable.

Pour conclure, nous pouvons dire que si notre dernier réseau était plus performant que les autres classifieurs, ces résultats ne sont pas encore exceptionnels (un peu moins de 50% de précision pour les 10 classes). Une solution aurait pu être (comme suggéré dans l'énoncé) d'ajouter une PCA ou d'autres méthodes d'analyse en amont du réseau. Nous n'avons cependant pas pris le temps de le faire dans ce travail. Il s'agissait également de l'idée de la dernière partie non traitée : utiliser de nouveaux descripteurs plus haut niveau mais plus significatifs que les simples niveaux RGB des pixels de l'image. Une autre solution aurait pu être d'aborder la question de l'optimisation des hyper-paramètre non pas en regardant les paramètres un à un comme nous l'avons fait, mais en essayant de les combiner dans les essais pour balayer de possibles interactions (comme nous l'avons fait pour le SVM). Ceci n'a cependant pas été réalisé pour des questions de temps de test et de calcul étant donné la date de fin de rédaction du rapport.