

Rapport de bureaux d'étude

Apprentissage automatique embarqué et mobile
S9 INFO mso 3.4

Mohamed Amine MEJRI
Thomas PUCCI

Édité le 28/03/2017

Sommaire

1	Introduction	3
2	KNN	4
2.1	Knn_compute_distances_two_loops.m	4
2.2	Knn_predict_labels.m	5
2.3	Now lets try out a larger k, say k=5	6
2.4	knn_compute_distance_one_loop.m	6
2.5	knn_compute_distances_no_loops.m	7
2.6	Let's compare how fast the implementations are	7
2.7	Cross-validation	7
2.7.1	Image	8
3	SVM	9
3.1	svm_loss_naive.m	9
3.1.1	Gradcheck	10
3.1.2	Inline Question 1	10
3.2	svm_loss_vectorized.m	10
3.2.1	Loss	10
3.2.2	Gradient	11
3.2.3	Résultats	11
3.3	Stochastic Gradient Descent	11
3.4	linear_svm_predict.m	12
3.5	Configuration des hyperparamètres dans le script Run_svm.m	12
3.6	Visualisation des résultats	14
3.6.1	Inline Question 2	14
4	SoftMax	15
4.1	softmax_loss_naive	15
4.1.1	Gradcheck	16
4.2	svm_loss_vectorized.m	16
4.2.1	Loss	16
4.2.2	Résultats	17
4.3	Stochastic Gradient Descent	17
4.4	linear_softmax_predict.m	17
4.5	Configuration des hyperparamètres dans le script Run_softmax.m	18
4.6	Visualisation des résultats	18

5	Q-Learning	20
5.1	Configuration	20
5.2	Fonction récursive qLearn	21
5.3	Traces d'exécution	22
6	Conclusion	23
	Références	23

1 Introduction

Le présent rapport rend compte de nos travaux lors des séances de Bureau d'Étude et suit le cheminement des énoncés [1].

Une première partie de ce rapport concerne la programmation d'un classifieur KNN. Une seconde partie concerne l'apprentissage d'un algorithme SVM. Une troisième partie concerne l'implémentation d'un classifieur de type Softmax. Une quatrième partie concerne la programmation d'un algorithme de type réseau de neurones à deux couches. Enfin, une cinquième partie concerne l'implémentation de l'exemple simple de *QLearning* du dernier BE [2].

*NB: Ce Bureau d'étude est réalisé sous le logiciel **Matlab**.*

Mots-clés: *kNN, Support Vector Machine, Softmax, Réseau de neurones, Images features, QLearning.*

2 KNN

2.1 Knn_compute_distances_two_loops.m

Dans cette partie on commence par implémenter le code qui mesure la matrice distance entre tout les training et les tests exemples. Par exemple si on a N_{tr} training exemples et N_{te} exemples test on obtient une matrice de taille $N_{te} \times N_{tr}$ ou chaque élément (i,j) est la distance entre le i ème test et le j ème train et ceci via un double boucle for.

Ceci est le code :

```

1 function [dists] = knn_compute_distances_two_loops(model, X)
2 % Compute the distance between each test point in X and each training point
3 % in model.X_train using a nested loop over both the training data and the
4 % test data.
5
6 % Inputs:
7 % - model: KNN model struct, it has two members:
8 %     model.X_train : A matrix of shape (num_train, D) containing train data.
9 %     model.y_train : A matrix of shape (num_train, 1) containing train labels.
10 % - X: A matrix of shape (num_test, D) containing test data.
11 % Returns:
12 % - dists: A matrix of shape (num_test, num_train) where dists[i, j]
13 %     is the Euclidean distance between the ith test point and the jth training
14 %     point.
15
16 num_test = size(X,1);
17 num_train = size(model.X_train, 1);
18 dists = zeros(num_test, num_train);
19 for i = 1:num_test
20     for j = 1:num_train
21 %         #####
22 %         # TODO:                                     #
23 %         # Compute the l2 distance between the ith test point and the jth      #
24 %         # training point, and store the result in dists[i, j]                  #
25 %         #####
26
27         dists(i,j)= (sum((X(i,:)-model.X_train(j,:)).^2)).^0.5;
28
29
30 %         #####
31 %         #                                     END OF YOUR CODE                                     #
32 %         #####
33     end
34 end
35 end

```

Après l'exécution on obtient une matrice de taille 500×5000 :

Imprime écran

2.2 Knn_predict_labels.m

On a implémenté ici la fonction qui prédit le label de chaque exemple test.

En prenant la matrice dists on trie cette matrice après on prend les k plus proches labels après en utilisant la fonction mod on obtient le label le plus commun càd le label qui se répète le plus .

Ceci est le code:

```

1 function [y_pred] = knn_predict_labels(model, dists, k)
2 %     Given a matrix of distances between test points and training points,
3 %     predict a label for each test point.
4 %
5 %     Inputs:
6 %     - model: KNN model struct, it has two members:
7 %         model.X_train : A matrix of shape (num_train, D) containing train data.
8 %         model.y_train : A matrix of shape (num_train, 1) containing train labels.
9 %     - dists: A matrix of shape (num_test, num_train) where dists[i, j]
10 %         gives the distance between the ith test point and the jth training point.
11 %     - k : number of nearest neighbors
12 %     Returns:
13 %     - y: A matrix of shape (num_test,) containing predicted labels for the
14 %         test data, where y[i] is the predicted label for the test point X[i].
15
16
17 % #####
18 % # TODO:                                     #
19 % # Use the distance matrix to find the k nearest neighbors of the ith      #
20 % # training point, and use model.train_labels to find the labels of these   #
21 % # neighbors. Store these labels in closest_y.                             #
22 % # Hint: Look up the function sort                                           #
23 % #####
24
25     [~,index] = sort(dists,2);
26     closest_y = model.y_train(index(:,1:k));
27
28
29
30
31
32 %
33 % #####
34 % # TODO:                                     #
35 % # Now that you have found the labels of the k nearest neighbors, you      #
36 % # need to find the most common label in the list closest_y of labels.     #
37 % # Store this label in y_pred[i]. Break ties by choosing the smaller        #
38 % # label.
39 % # Hint: Look up the function mode
40 % #####
41
42     y_pred = mode(closest_y,2);
43 %
44 % #####
45 % #                                     END OF YOUR CODE                       #
46 % #####

```

47 `end`

Le résultat obtenu après l'exécution de cette partie est le suivant avec une accuracy de 0.274000

Imprime écran

2.3 Now lets try out a larger k, say k=5

Dans cette partie on a utilisé un k=5 et on a effectué le test pour obtenir une accuracy égale à ...

Imprime écran

2.4 knn_compute_distance_one_loop.m

Dans cette partie on amélioré l'algorithme qui calcule la matrice dists en utilisant une vectorisation partielle avec un seul boucle for.

On peut vérifier maintenant qu'on obtient la même résultat que précédemment

```

1 function [ dists ] = knn_compute_distances_one_loops( model, X )
2 % Compute the distance between each test point in X and each training point
3 % in self.X_train using a single loop over the test data.
4
5 %   Inputs:
6 %   - model: KNN model struct, it has two members:
7 %       model.X_train : A matrix of shape (num_train, D) containing train data.
8 %       model.y_train : A matrix of shape (num_train, 1) containing train labels.
9 %   - X: A matrix of shape (num_test, D) containing test data.
10 %   Returns:
11 %   - dists: A matrix of shape (num_test, num_train) where dists[i, j]
12 %       is the Euclidean distance between the ith test point and the jth training
13 %       point.
14   num_test = size(X,1);
15   num_train = size(model.X_train, 1);
16   dists = zeros(num_test, num_train);
17   for i=1:num_test
18 %       #####
19 %       # TODO:                                     #
20 %       # Compute the l2 distance between the ith test point and all training #
21 %       # points, and store the result in dists[i, :].                         #
22 %       #####
23
24       dists(i,:) = (sum((model.X_train - repmat(X(i,:), num_train, 1)).^2, 2)).^0.5;
25
26 %       #####
27 %       #                                     END OF YOUR CODE                                     #
28 %       #####
29   end
30 end

```

Imprime écran

2.5 knn_compute_distances_no_loops.m

Maintenant on améliore mieux notre algorithme en calculant la matrice dists sans aucune boucle .

Ceci est le code:

```

1 function [ dists ] = knn_compute_distances_no_loops( model, X )
2 % Compute the distance between each test point in X and each training point
3 % in self.X_train using a single loop over the test data.
4 %   Inputs:
5 %   - model: KNN model struct, it has two members:
6 %       model.X_train : A matrix of shape (num_train, D) containing train data.
7 %       model.y_train : A matrix of shape (num_train, 1) containing train labels.
8 %   - X: A matrix of shape (num_test, D) containing test data.
9 %   Returns:
10 %   - dists: A matrix of shape (num_test, num_train) where dists[i, j]
11 %       is the Euclidean distance between the ith test point and the jth training
12 %       point.
13
14 % #####
15 % # TODO:                                     #
16 % # Compute the l2 distance between all test points and all training      #
17 % # points without using any explicit loops, and store the result in      #
18 % # dists.                                                                    #
19 % # HINT: Try to formulate the l2 distance using matrix multiplication     #
20 % #       and two broadcast sums.                                           #
21 % #####
22
23 num_test = size(X,1);
24 num_train = size(model.X_train, 1);
25 dists = (sum(X.^2,2)*ones(1,num_train) +
26         ones(num_test,1)*sum(model.X_train.^2,2)' - X*model.X_train'.*2).^0.5;
27 % #####
28 % #                                     END OF YOUR CODE                    #
29 % #####
30 end

```

On vérifie bien qu'on obtient le même résultat que précédemment

Imprime écran

2.6 Let's compare how fast the implementations are

Ici on a comparé la vitesse d'exécution de nos trois versions de calcul de la matrice dists et on remarque selon les résultats que la version knn_compute_distance_no_loops est la plus rapide

Imprime écran

2.7 Cross-validation

Nous avons implémenté le k-Nearest Neighbor classificateur, mais nous avons fixé la valeur $k = 5$ arbitrairement. Nous allons maintenant déterminer la meilleure valeur de cet hyperparamètre avec cross validation.

Nous avons divisé nos données en 5 folds ou à chaque fois on prend une valeur de k et on utilise 4 folds pour entraîner notre modèle et une pour le test .

Ceci est le code : `{../assets/classifier/Run_KNN.m}`

Pour chaque valeur de k on obtient les résultats suivants:

Imprime écran

En se basant sur ces résultats on remarque que la meilleure valeur de k est : ...

On met k = ... on entraîne notre modèle sur toutes nos données d'entraînement et on teste sur nos données de test on obtient le résultat suivant avec une précision égale à ...

Imprime écran

2.7.1 Image

3 SVM

3.1 svm_loss_naive.m

Nous implémentons dans un premier temps la fonction *loss* ainsi que le gradient dans le fichier `svm_loss_naive.m`

```

1 function [ loss, dW ] = svm_loss_naive( W, X, y, reg )
2
3     dW = zeros(size(W));
4     num_classes = size(W,1);
5     num_train = size(X, 1);
6
7     loss = 0.0;
8     for i = 1:num_train
9         scores = W*X(i, :)' ;
10        correct_class_score = scores(y(i));
11        for j = 1:num_classes
12            if j == y(i)
13                continue;
14            end
15            margin = scores(j) - correct_class_score + 1; % note delta = 1
16            if margin > 0
17                loss = loss + margin;
18                %your code
19                dW(j,:) = dW(j,:) + X(i, :);
20                dW(y(i),:) = dW(y(i),:) - X(i, :);
21            end
22        end
23    end
24
25    % Right now the loss is a sum over all training examples, but we want it
26    % to be an average instead so we divide by num_train
27    loss = loss/num_train;
28
29    % Average gradients as well
30    %your code
31    dW = dW/num_train;
32
33    % Add regularization to the loss.
34    loss = loss + 0.5 * reg * sum(sum((W.*W)));
35
36    % Add regularization to the gradient
37    % your code
38    dW = dW + reg * W;
39
40 end

```

3.1.1 Gradcheck

Ce code permet la comparaison du calcul du *loss* analytique et numérique :

```
1 numerical: -11.624341 analytic: -11.623547, relative error: 3.416353e-05
2 numerical: -2.967959 analytic: -2.968469, relative error: 8.596690e-05
3 numerical: 1.370071 analytic: 1.370463, relative error: 1.429105e-04
4 numerical: -34.207994 analytic: -34.207484, relative error: 7.450883e-06
5 numerical: 15.514664 analytic: 15.515966, relative error: 4.194596e-05
6 numerical: -0.409555 analytic: -0.408745, relative error: 9.892837e-04
7 numerical: 15.763268 analytic: 15.764060, relative error: 2.509722e-05
8 numerical: -25.588662 analytic: -25.588890, relative error: 4.470214e-06
9 numerical: -4.197175 analytic: -4.193180, relative error: 4.760962e-04
10 numerical: -9.371776 analytic: -9.370036, relative error: 9.280094e-05
```

Avec le terme de régularisation nous obtenons :

```
1 numerical: 16.210153 analytic: 16.211038, relative error: 2.730251e-05
2 numerical: -1.609398 analytic: -1.615602, relative error: 1.923819e-03
3 numerical: -17.736330 analytic: -17.721728, relative error: 4.118093e-04
4 numerical: -11.510254 analytic: -11.493891, relative error: 7.113270e-04
5 numerical: -6.772364 analytic: -6.775551, relative error: 2.352329e-04
6 numerical: 9.015804 analytic: 9.005720, relative error: 5.595433e-04
7 numerical: -8.930279 analytic: -8.915413, relative error: 8.330575e-04
8 numerical: 2.049767 analytic: 2.052767, relative error: 7.311906e-04
9 numerical: 27.281231 analytic: 27.279314, relative error: 3.513422e-05
10 numerical: 26.358994 analytic: 26.371298, relative error: 2.333422e-04
```

Les erreurs sont extrêmement faibles, nous supposons que l'implémentation est juste.

3.1.2 Inline Question 1

It is possible that once in a while a dimension in the gradcheck will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? What is a simple example in one dimension where a gradient check could fail? Hint: the SVM loss function is not strictly speaking differentiable.

Le gradient n'est pas strictement différentiable. En considérant une dimension, un point peut avoir un gradient analytique nul et un gradient numérique positif par exemple. D'où la possibilité d'une erreur dans le *gradcheck* pour certaines dimensions de temps en temps. Ceci ne doit pas être préoccupant pour autant.

3.2 svm_loss_vectorized.m

Nous implémentons la version vectorisée de l'algorithme précédent :

3.2.1 Loss

```
1 num_classes = size(W,1);
2
3 scores = W * X';
4 correct_indexes = (0:num_train-1)*num_classes+double(y');
5 correct_class_score = ones(num_classes,1)*scores(correct_indexes);
```

```

6 L = scores - correct_class_score + 1; % delta = 1
7
8 L(L<0) = 0;
9 loss = sum(sum(L))- 1*num_train; % On retire les Lyi comptés en trop
10 loss = loss/num_train;
11
12 % Regularization
13 loss = loss + 0.5 * reg * sum(sum((W.*W)));

```

3.2.2 Gradient

```

1 L(L>0) = 1;
2 L(correct_indexes) = -sum(L);
3 dW=L*X;
4
5 dW = dW/num_train;
6 % Add regularization to the gradient
7 dW = dW + reg * W;

```

3.2.3 Résultats

Nous obtenons les résultats suivant :

```

1 Naive loss: 9.150210e+00 computed in 38.956417s
2 Vectorized loss: 9.150210e+00 computed in 0.780730s
3 difference: -0.000000
4
5 Naive loss and gradient: computed in 36.895610s
6 Vectorized loss and gradient: computed in 0.480518s
7 difference: 291.412469

```

La différence est numériquement nulle : nous considérons que l'implémentation est correcte. Nous remarquons l'efficacité de la version vectorisée par rapport à la version analytique.

3.3 Stochastic Gradient Descent

Nous implémentons le SGD dans le fichier `linear_svm_train.m`.

```

1 ...
2     rand_idx = randsample(num_train, batch_size);
3     X_batch = X(rand_idx, :);
4     y_batch = y(rand_idx);
5
6     %evaluate loss and gradient
7     [loss, grad] = self_loss(W, X_batch, y_batch, reg);
8     loss_hist(it) = loss;
9
10 %perform parameter update
11     W = W - learning_rate * grad;
12 ...

```

Nous obtenons la descente suivante (cf. figure).

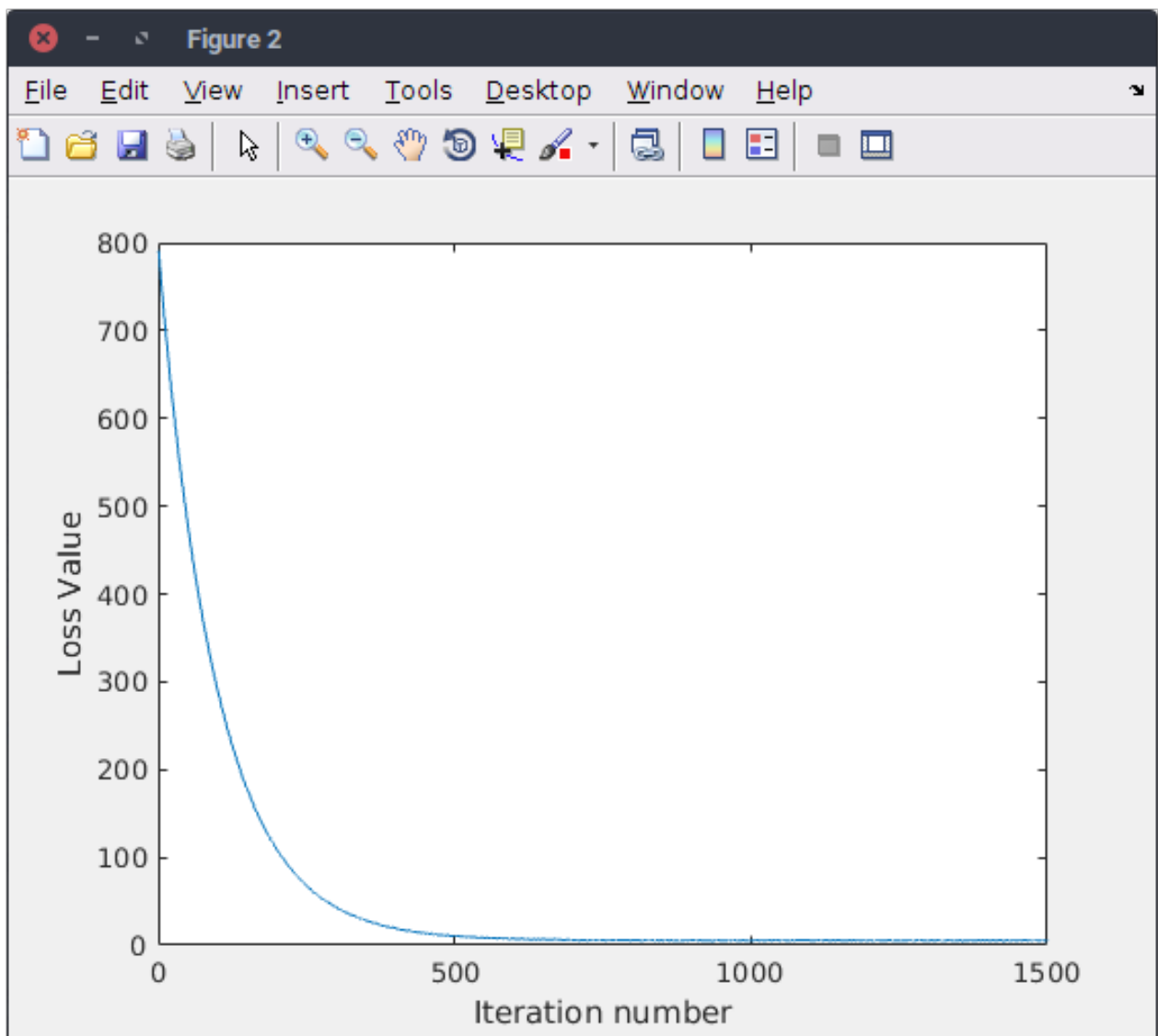


Figure 3.A: SGD

3.4 linear_svm_predict.m

La fonction de prédiction s'écrit simplement de la manière suivante :

```
1 function [ y_pred ] = linear_svm_predict( model, X )
2     y_pred = [];
3     scores = model.W * X';
4     [~,y_pred] = max(scores);
5 end
```

3.5 Configuration des hyperparamètres dans le script Run_svm.m

Afin de choisir les meilleurs paramètres, nous créons une boucle testant l'ensemble des paramètres choisis :

```
1 iter_num = 1000;
```

```

2 %iter_num = 100;
3 for i = 1:length(learning_rates)
4     learning = learning_rates(i);
5     for j = 1:length(regularization_strengths)
6         regularization = regularization_strengths(j);
7         [model, loss_hist] = linear_svm_train(imdb.X_train, imdb.y_train, learning,
8             regularization, iter_num);
9         y_train_pred = linear_svm_predict(model, imdb.X_train);
10        train_accuracy = mean(imdb.y_train == y_train_pred');
11        fprintf('training accuracy: %f', train_accuracy); % (train_accuracy)
12        y_val_pred = linear_svm_predict(model, imdb.X_val);
13        val_accuracy = mean(imdb.y_val' == y_val_pred);
14        fprintf('validation accuracy: %f', val_accuracy); % (val_accuracy)
15
16        if val_accuracy > best_val
17            best_val = val_accuracy;
18            best_svm = model;
19        end
20
21        results(i, j, :) = [train_accuracy, val_accuracy];
22    end
23 end

```

Visualisation des expériences :

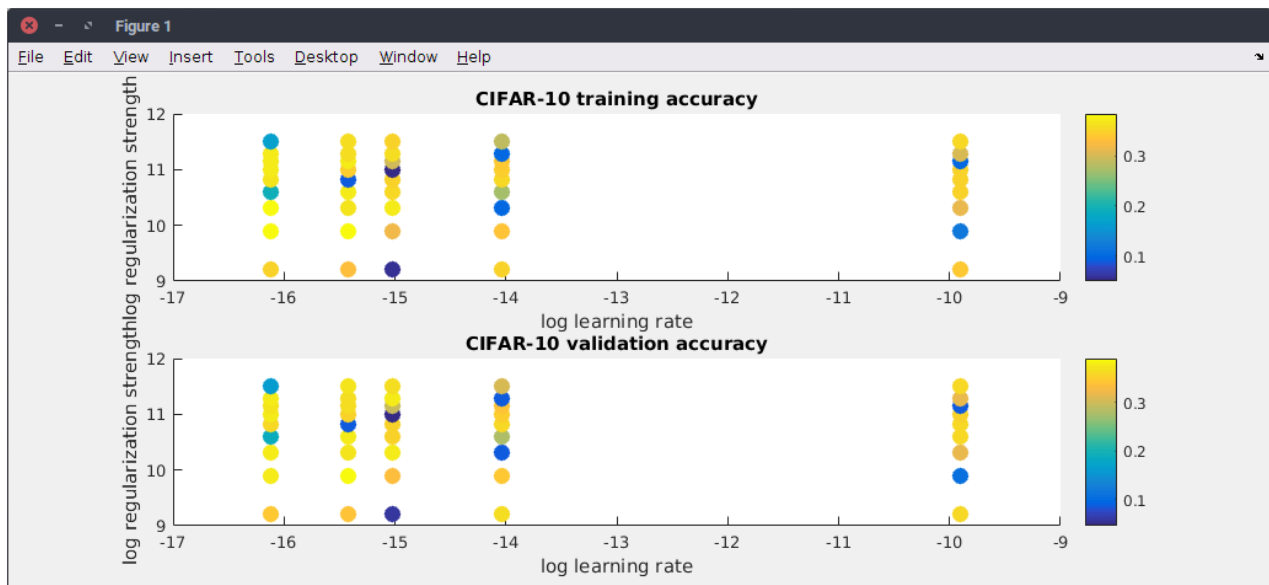


Figure 3.B: Hyperparamètres

Les meilleurs résultats retournés sont :

```

1 ...
2 lr 1.000000e-07 reg 3.000000e+04 train accuracy: 0.379776 val accuracy: 0.390000
3 ...
4 best validation accuracy achieved during cross-validation: 0.390000
5 linear SVM on raw pixels final test set accuracy: 0.382000

```

3.6 Visualisation des résultats

Voici les matrices images résultantes de l'apprentissage de l'algorithme pour chacune des classes d'images:

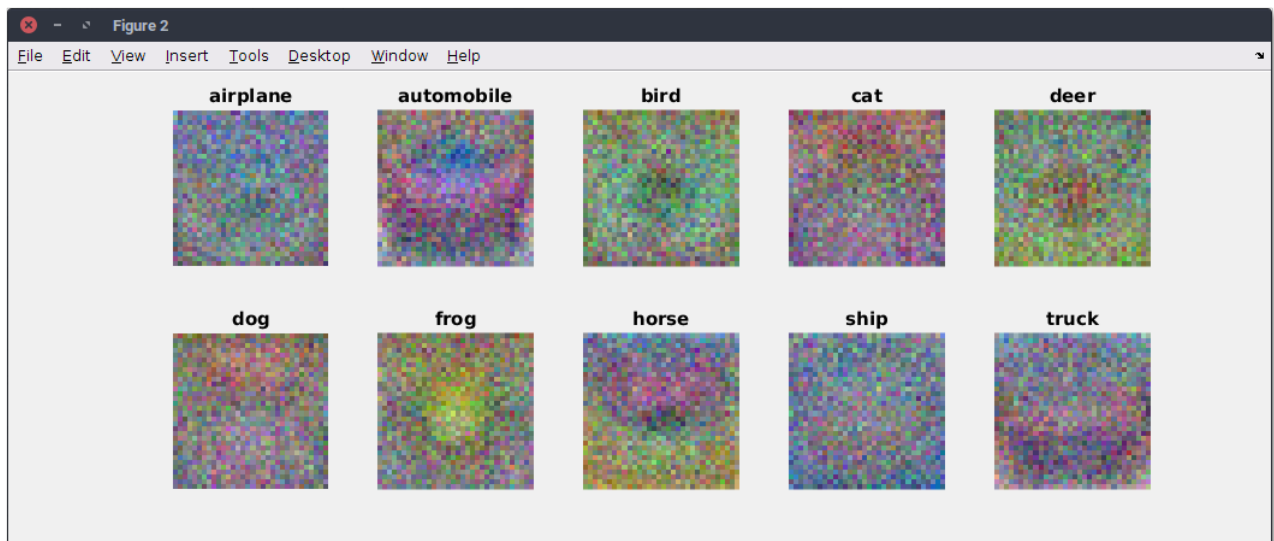


Figure 3.C: SVM weights images classes

3.6.1 Inline Question 2

Describe what your visualized SVM weights look like, and offer a brief explanation for why they look the way that they do.

Ces matrices images sont les représentations de chaque classe pour la SVM. Elles sont très distinctes en termes de couleurs par rapport à chaque région de l'image car la SVM sépare au mieux chaque classe, d'où ces représentations singulières.

4 SoftMax

L'algorithme SoftMax se différencie du classifieur SVM notamment par le retour d'une probabilité d'appartenance aux classes plutôt que le retour d'une classification simple à l'utilisateur. SoftMax généralise la régression logistique à plusieurs classes.

4.1 softmax_loss_naive

Nous implémentons dans un premier temps la fonction *loss* ainsi que le gradient dans le fichier `softmax_loss_naive.m`

```

1 function [ loss, dW ] = softmax_loss_naive( W, X, y, reg )
2
3 % Initialize the loss and gradient to zero.
4 loss = 0.0;
5 dW = zeros(size(W));
6
7 num_class = size(W,1);
8 num_train = size(X,1);
9
10 for i = 1:num_train
11     scores = W*X(i, :);
12     scores = scores - max(scores);
13
14     sum_exp = sum(exp(scores));
15     loss = loss - log(exp(scores(y(i)))/sum_exp);
16
17     for j = 1:num_class
18
19         dW(j,:) = dW(j,:) + exp(scores(j))/sum_exp*X(i, :);
20
21         if j == y(i)
22             dW(j,:) = dW(j,:) - X(i, :);
23         end
24     end
25 end
26
27 % Right now the loss is a sum over all training examples, but we want it
28 % to be an average instead so we divide by num_train
29 loss = loss/num_train;
30
31 % Average gradients as well
32 dW = dW/num_train;
33
34 % Add regularization to the loss.
35 loss = loss + 0.5 * reg * sum(sum((W.*W)));
36
37 % Add regularization to the gradient

```



```

39 % your code
40 dW = dW + reg * W;
41
42 end

```

4.1.1 Gradcheck

Ce code permet la comparaison du calcul du *loss* analytique et numérique :

```

1 numerical: 1.460241 analytic: 1.460241, relative error: 4.602831e-08
2 numerical: -0.917689 analytic: -0.917689, relative error: 1.943096e-08
3 numerical: 1.647922 analytic: 1.647922, relative error: 4.257335e-08
4 numerical: -0.161815 analytic: -0.161815, relative error: 1.431171e-07
5 numerical: -0.201888 analytic: -0.201889, relative error: 4.207505e-07
6 numerical: -1.439165 analytic: -1.439165, relative error: 3.392779e-09
7 numerical: 2.334927 analytic: 2.334927, relative error: 2.989751e-08
8 numerical: 2.074058 analytic: 2.074058, relative error: 3.641542e-08
9 numerical: 1.029669 analytic: 1.029669, relative error: 5.111217e-08
10 numerical: 3.549438 analytic: 3.549438, relative error: 2.802475e-09

```

Avec le terme de régularisation nous obtenons :

```

1 numerical: -3.625634 analytic: -3.622718, relative error: 4.023212e-04
2 numerical: -1.654415 analytic: -1.634746, relative error: 5.979858e-03
3 numerical: 2.343703 analytic: 2.345992, relative error: 4.879353e-04
4 numerical: -3.047483 analytic: -3.060509, relative error: 2.132565e-03
5 numerical: 1.060127 analytic: 1.058400, relative error: 8.149228e-04
6 numerical: -0.253623 analytic: -0.250525, relative error: 6.145836e-03
7 numerical: 1.996955 analytic: 1.970200, relative error: 6.744259e-03
8 numerical: 1.490734 analytic: 1.479275, relative error: 3.858266e-03
9 numerical: -1.422064 analytic: -1.421236, relative error: 2.911050e-04
10 numerical: -0.208817 analytic: -0.207548, relative error: 3.046673e-0

```

Les erreurs sont extrêmement faibles, nous supposons que l'implémentation est juste.

4.2 svm_loss_vectorized.m

Nous implémentons la version vectorisée de l'algorithme précédent :

4.2.1 Loss

```

1 num_class = size(W,1);
2 num_train = size(X,1);
3
4 scores = W*X';
5 scores = scores - ones(num_class,1)*max(scores);
6 correct_indexes = (0:num_train-1)*num_class+double(y');
7
8 sum_exp = sum(exp(scores));
9 loss = sum(- log(exp(scores(correct_indexes)./sum_exp))));
10
11 L = exp(scores)./(ones(num_class,1)*sum_exp);

```

```

12 L(correct_indexes) = L(correct_indexes)-1;
13
14 dW=L*X;
15
16 % Right now the loss is a sum over all training examples, but we want it
17 % to be an average instead so we divide by num_train
18 loss = loss/num_train;
19
20 % Average gradients as well
21 dW = dW/num_train;
22
23 % Add regularization to the loss.
24 loss = loss + 0.5 * reg * sum(sum((W.*W)));
25
26 % Add regularization to the gradient
27 % your code
28 dW = dW + reg * W;

```

4.2.2 Résultats

Nous obtenons les résultats suivant :

```

1 Naive loss: 2.410764e+00 computed in 23.085370s
2 Vectorized loss: 1.036558e-01 computed in 0.779152s
3 Loss difference: 2.307108
4 Gradient difference: 0.000000

```

La différence est numériquement nulle : nous considérons que l'implémentation est correcte. Nous remarquons l'efficacité de la version vectorisée par rapport à la version analytique.

4.3 Stochastic Gradient Descent

Nous implémentons le SGD dans le fichier `linear_softmax_train.m`. L'implémentation est identique à celle du classifieur SVM.

```

1 ...
2     rand_idx = randsample(num_train, batch_size);
3     X_batch = X(rand_idx, :);
4     y_batch = y(rand_idx);
5
6     %evaluate loss and gradient
7     [loss, grad] = self_loss(W, X_batch, y_batch, reg);
8     loss_hist(it) = loss;
9
10 %perform parameter update
11     W = W - learning_rate * grad;
12 ...

```

4.4 linear_softmax_predict.m

La fonction de prédiction s'écrit simplement de la manière suivante, tout comme le classifieur SVM :

```

1 function [ y_pred ] = linear_softmax_predict( model, X )
2     y_pred = [];
3     scores = model.W * X';
4     [~,y_pred] = max(scores);
5 end

```

4.5 Configuration des hyperparamètres dans le script Run_softmax.m

Afin de choisir les meilleurs paramètres, nous créons une boucle testant l'ensemble des paramètres choisis. Il s'agit de la même boucle que celle utilisée pour le classifieur SVM :

```

1 iter_num = 1000;
2 %iter_num = 100;
3 for i = 1:length(learning_rates)
4     learning = learning_rates(i);
5     for j = 1:length(regularization_strengths)
6         regularization = regularization_strengths(j);
7         [model, loss_hist] = linear_softmax_train(imdb.X_train, imdb.y_train, learning,
8             regularization, iter_num);
9         y_train_pred = linear_softmax_predict(model, imdb.X_train);
10        train_accuracy = mean(imdb.y_train == y_train_pred');
11        fprintf('training accuracy: %f', train_accuracy); % (train_accuracy)
12        y_val_pred = linear_softmax_predict(model, imdb.X_val);
13        val_accuracy = mean(imdb.y_val' == y_val_pred);
14        fprintf('validation accuracy: %f', val_accuracy); % (val_accuracy)
15
16        if val_accuracy > best_val
17            best_val = val_accuracy;
18            best_softmax = model;
19        end
20        results(i, j, :) = [train_accuracy, val_accuracy];
21    end
22 end

```

Les meilleurs résultats retournés sont :

```

1 ...
2 lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.325959 val accuracy: 0.341000
3 ...
4 best validation accuracy achieved during cross-validation: 0.341000
5 linear Softmax on raw pixels final test set accuracy: 0.333000

```

4.6 Visualisation des résultats

Voici les matrices images résultantes de l'apprentissage de l'algorithme pour chacune des classes d'images :

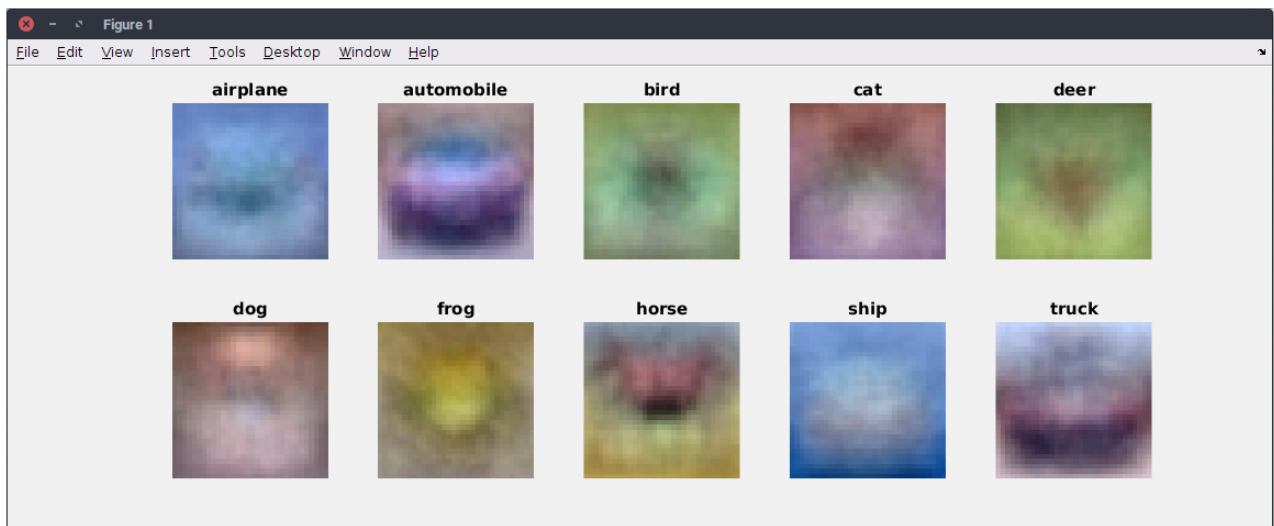


Figure 4.A: Softmax weights images

On remarque que les images sont bien plus lisses que celles générées par le classifieur SVM. C'est normal puisque SoftMax ne cherche pas à éloigner les classes le plus possibles les unes par rapport aux autres dans un hyperplan mais d'afficher les probabilités d'appartenance à des classes.

5 Q-Learning

Cet exercice est réalisé en suivant l'exemple de cours [2] dont nous reprenons ici l'illustration :

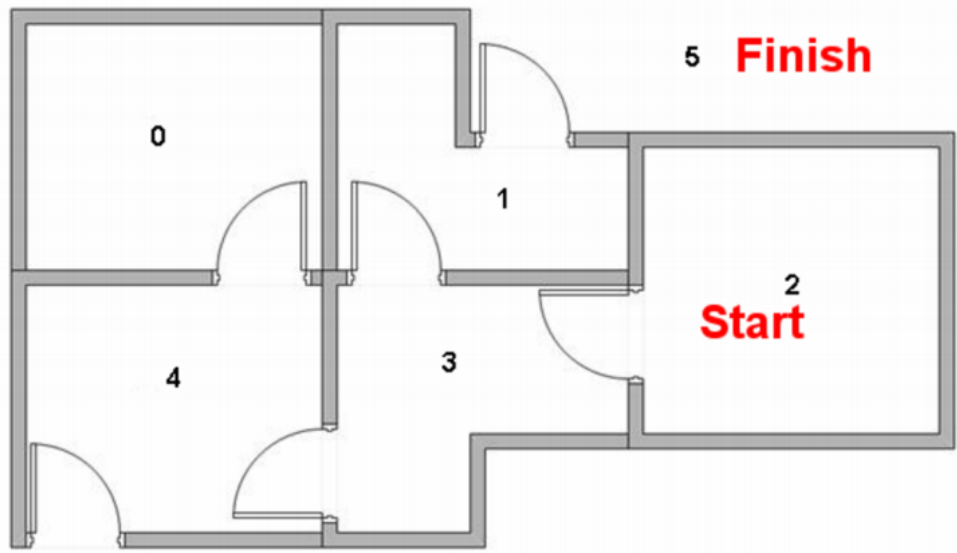


Figure 5.A: L'agent doit apprendre les meilleurs chemins vers la position 5

5.1 Configuration

Nous utilisons les données de l'énoncé pour configurer les variables de l'algorithme. Dans un premier temps, nous considérons la matrice *récompenses* suivante :

$$R = \begin{pmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{pmatrix}$$

Celle-ci est codée en créant une matrice ne comportant que les valeurs -1 (murs), puis en modifiant certaines valeurs à 0 (représentant les portes) et d'autres à 100 (représentant les changements de pièces gagnants).

```

1 R = -1*ones(6);
2 doors = [[0,4]; [4,3]; [4,5]; [2,3]; [1,3]; [1,5]];
3 wins = [[1,5]; [4,5]; [5,5]];
4
5 for i = 1:size(doors,1) % Création des portes
6     R(doors(i,1)+1,doors(i,2)+1) = 0;
7     R(doors(i,2)+1,doors(i,1)+1) = 0;
8 end
9
10 for i = 1:size(wins,1) % Chemins gagnants
```

```

11 R(wins(i,1)+1,wins(i,2)+1) = 100;
12 end

```

On règle ensuite les paramètres `alpha`, `gamma` et le nombre d'épisodes à réaliser :

```

1 alpha = 1;
2 gamma = .8;
3 nEpisodes = 100;

```

NB : Ces paramètres entraînent donc la formule d'apprentissage suivantes (annulation du terme $Q_t(s_t, a_t)$):

$$Q_{t+1}(s_t, a_t) = R_{t+1} + 0,8 * \max_a Q_t(s_{t+1}, a_t)$$

On initialise la matrice d'apprentissage `Q` et on choisit les états initiaux pour chacun des épisodes de manière aléatoire.

```

1 Q = zeros(size(R));
2 randomStates = randi([1 size(R,2)],1,100);

```

5.2 Fonction récursive qLearn

Dans un nouveau fichier `qLearn.m` nous programmons une fonction récursive définie de la manière suivante :

```

1 function Q = qLearn(Q,R,alpha,gamma,state,stopState)

```

Nous identifions dans un premier temps les états suivants possibles `possibleNextStates` étant donné l'état courant `state`. Nous choisissons ensuite aléatoirement l'état suivant `nextState` parmi les possibilités `possibleNextStates`. Nous identifions ensuite les états futurs possibles `possibleFutureStates` étant donné l'état suivant `nextState`. Puis nous appliquons la formule d'apprentissage et actualisons la valeur de `Q(state,nextState)` en fonction de `alpha`, `R(state,nextState)`, `gamma` et `max(Q(nextState,possibleFutureStates))`.

Si l'état suivant `nextState` correspond à l'état final, nous arrêtons la récursion, sinon nous rappelons la fonction récursive `qLearn`.

```

1 function Q = qLearn(Q,R,alpha,gamma,state, stopState)
2     possibleNextStates = find(R(state,:) >= 0);
3     nextState = possibleNextStates(randi(size(possibleNextStates)));
4     possibleFutureStates = find(R(nextState,:) >= 0);
5     Q(state,nextState) = Q(state,nextState) + alpha * (R(state,nextState) +
6         gamma*max(Q(nextState,possibleFutureStates)) - Q(state,nextState));
7     if nextState == stopState
8         return
9     else
10         Q = qLearn(Q,R,alpha,gamma,nextState,stopState);
11 end

```

Dans le script principal, nous appelons cette fonction sur chacun des épisodes :

```

1 for i = 1:nEpisodes % Boucle de nEpisodes
2     beginningState = randomStates(i);
3     Q = qLearn(Q,R,alpha,gamma,6, 5+1); % Appel de la fonction récursive qLearn
4 end

```

Enfin nous affichons le résultat :

```
1 QNormalized = round(Q./max(max(round(Q)))*100) % Affichage du résultat arrondi
```

5.3 Traces d'exécution

Nous exécutons l'ensemble du code Matlab à partir du fichier `Run_qlearning.m`. Voici le résultat:

```
1 >> Run_qlearning
2
3 QNormalized =
4
5     0     0     0     0    80     0
6     0     0     0    64     0   100
7     0     0     0    64     0     0
8     0    80    51     0    80     0
9    64     0     0    64     0   100
10    0    80     0     0    80   100
```

Nous obtenons effectivement la même matrice résultat que celle se trouvant dans l'énoncé du BE [2].

6 Conclusion

Ce bureau d'étude nous a permis de réaliser des classifieurs simples et complets. Nous avons mis à profit la vitesse de calcul de Matlab dans les cas vectorisés et entraîné des algorithmes en testant différents hyperparamètres.

Ce BE est la compilation de nombreuses méthodes de classification et permet de comprendre les différences entre ces dernières. kNN est un classifieur par “voisinage”, SVM est un classifieur par “distanciation des instances”, Softmax est un classifieur probabiliste et est la généralisation de la régression logistique au cas multiclasse. Le réseau de neurones copie le fonctionnement des cerveaux biologiques et donne également de bons résultats. L'algorithme Q-Learn permet l'entraînement supervisé d'une machine, si nous savons valoriser chacune de ses actions.

Ce BE est également un bon moyen de se remémorer la plupart des fonctions Matlab, essentielles pour paralléliser les calculs et permettre une exécution rapide des algorithmes.

Références

- [1] E.D. L. Cheng, TD – convolutional neural networks for visual recognition, (2016).
- [2] E. Dellandréa, Cours – apprentissage par renforcement, (2016).