CS440: Intro to Artificial Intelligence, Fall 2017, Homework 4

Name: Nestor Alejandro Bermudez Sarmiento (nab6)

Worked individually

Introduction

Assignment 4 covers the topics of Perceptrons and Q-learning by implementing a digit classifier (from the same data as for Assignment 3) and a game of Pong, respectively. The first part is about the digit classification problem; k-nearest neighbor and perceptron classifiers will be implemented. In the second part I'll discuss the implementation using reinforcement learning to train the agent to play Pong.

As in the previous assignments, I'll be using Python 3.6.

Part 1

Since we will be using the same data set we used in Assignment 3 I'll reuse some of the classes I created previously; namely:

1. **Parser**: class to read the data files and create an internal representation.
2. **FeatureExtractors**: takes the original data and create features from it, be it: binarize it, group pixels or use a ternary feature.
3. **Utilities**: helper methods to print out matrices, create the color maps and scatter plots.

Part 1.1

For this part I have implemented a new classifier class called **PerceptronClassifier**. The interface is pretty much the same as for the previously developed classifiers: it has *train* and **evaluate** methods that create the training curve and confusion matrix respectively.
The classifier accepts the following parameters:

1. **epochs**: number of iterations until convergence of the weights.
2. **shuffle**: a boolean flag that indicates whether the training examples will be used in a fixed order or randomized.
3. **use_bias**: a boolean flag that indicates whether we should augment our features to generate a β_0 weight, a.k.a. bias. If the bias is used then the features are "augmented" with an additional dimension with value 1.

4. **zero_weights**: a boolean flag that indicates whether the weights should be initialized using all zeros or a random value between 0 and 1.

5. **learning_rate_decay**: a function that dictates how big should the step for the weight update should be. Two different functions were explored: time inverse and exponential decay. The idea for using these was inspired by the available decay functions in Tensorflow[1].

6. **learning_rate**: a constant that may be used by the decay function along with the epoch.

As per the implementation itself, I simply repeat the process for every epoch, on each iteration every example in the training data set is classified using the current weights and if it is misclassified then the corresponding weights are updated based on the decay function, learning rate constant and the example in question.

Multiple configurations of the parameters were tried and the best was chosen. The following table contains a summary of the different parameters and the corresponding results.

Perceptron parameter grid search

Epochs	Decay fn	Order	Bias	Zero w	Accuracy	Train time	Test time
10	Inverse	Random	Yes	Yes	80.6%	7.63	0.19
20	Inverse	Random	Yes	Yes	80.5%	10.85	0.19
50	Inverse	Random	Yes	Yes	81.5%	17.99	0.2
100	Inverse	Random	Yes	Yes	82.6%	33.92	0.2
100	Inverse	Fixed	Yes	Yes	82%	33.36	0.2
100	Inverse	Random	No	Yes	81.5%	30.04	0.19
100	Inverse	Random	Yes	No	82%	33.8	0.2
100	Exp	Random	Yes	Yes	81.6%	35.07	0.2
100	Inverse	Fixed	No	No	79.9%	29.67	0.19

Table 1: Different parameters for perceptron classifier

Now, lets look at the corresponding confusion matrix for the best accuracy obtained: 100 epochs, time inverse decay function, random process of examples, with bias and zero initialized weights.

[1] https://www.tensorflow.org/versions/r0.12/api_docs/python/train/decaying_the_learning_rate

		Predicted class									
		0	1	2	3	4	5	6	7	8	9
Class	0	94.44	0	1.11	0	0	0	1.11	0	2.22	1.11
	1	0	97.22	0.93	0	0.93	0	0.93	0	0	0
	2	0	1.94	83.50	3.88	1.94	0	3.88	2.91	1.94	0
	3	0	0	2	80	0	9	2	4	3	0
	4	0	0	3.74	0.93	82.24	0	2.8	2.8	0	7.48
	5	2.17	0	1.09	6.52	0	78.26	0	3.26	6.52	2.17
	6	1.1	1.1	2.2	0	2.2	2.2	86.81	2.2	2.2	0
	7	0.94	2.83	3.77	1.89	2.83	0	0	77.36	0.94	9.43
	8	0	1.94	6.8	7.77	1.94	4.85	2.91	0.97	68.93	3.88
	9	0	0	1	6	8	1	0	5	1	78

Table 2: Confusion matrix. Values are percentages.

Overall accuracy: **82.6%**. We can see that the performance of this classifier is better than the Naive Bayes classifier using single pixels as features that I created for Assignment 3. With Naive Bayes the best accuracy achieved was **77.6%**.
By looking at the confusion matrices we can see that the improvement comes from a better accuracy when classifying digits 0 to 7. The accuracy for digits 8 and 9 decreased when using the perceptron classifier.

While training the classifier, for every epoch, the classifier was evaluated using the same training data and the accuracy was captured. The following chart captures the progress in the accuracy of the model.

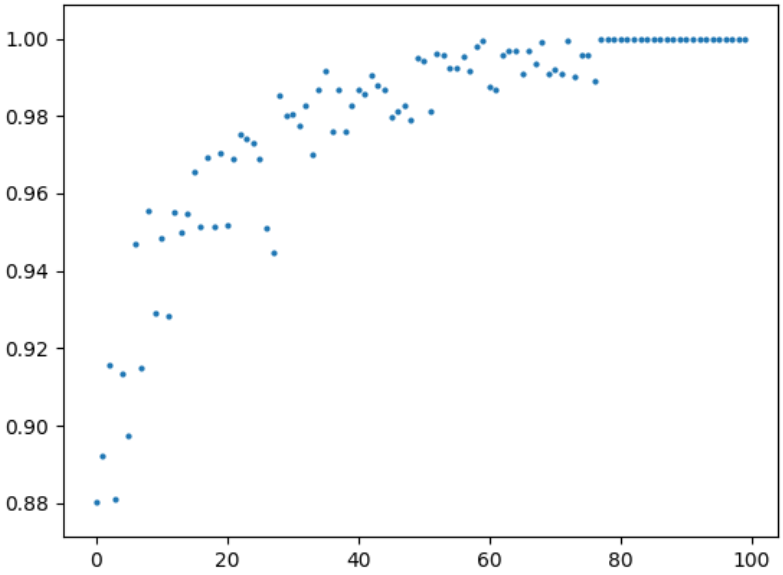

Figure 1: Accuracy as k increases

Note: the training curve and confusion matrix was capture for every configuration of parameters and they can be found under the results folder accompanying this report. They are not included in the report because the instructions indicated it was not necessary and for the sake of brevity.

Part 1.2

In this section we cover the implementation of a k-nearest neighbor classifier and compare the results with the previous classifiers.

For k-nearest neighbor I tuned two parameters: first the value of k and, second, the distance measure used.

The values of k were: 1, 2, 3, 5, 7, 13, 23, 43 and 91. And I tried three different distance measurements for each of those k. The distance functions are: Euclidean distance, Cosine similarity and Jaccard coefficient. The following charts show how the accuracy changes as a function of k for the different distances.

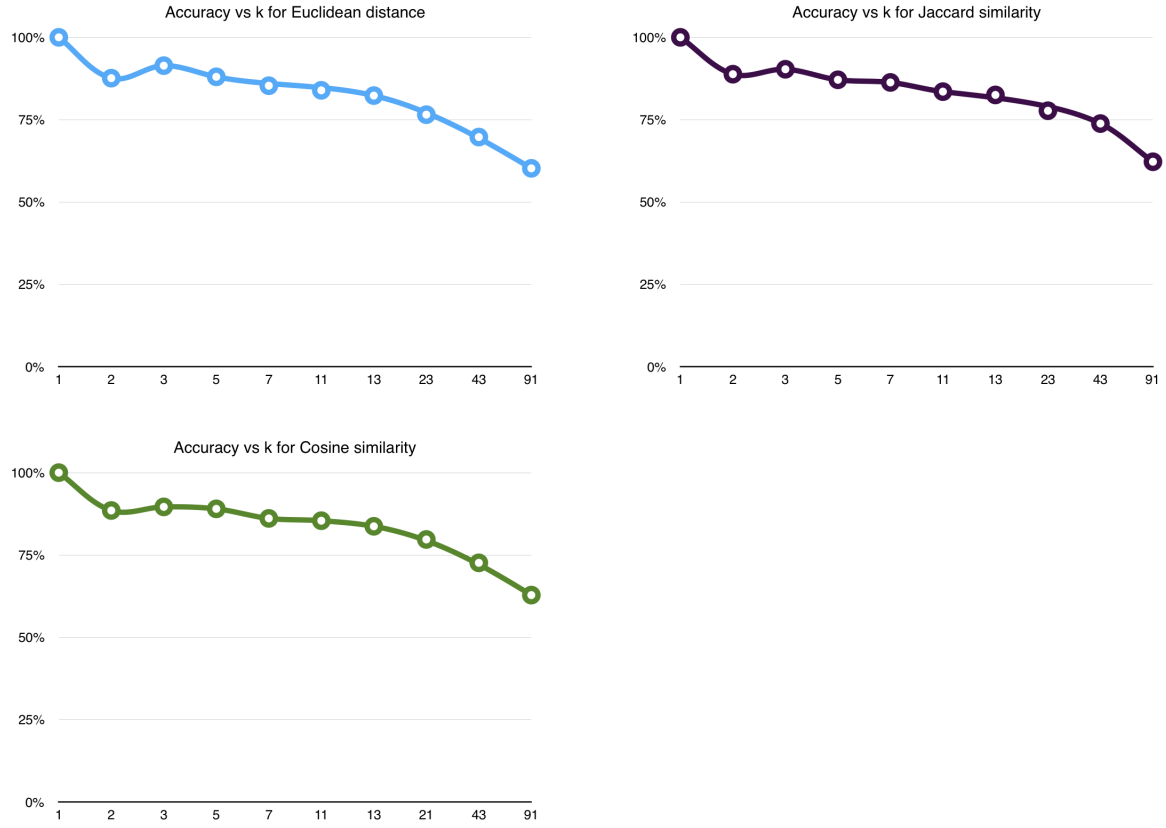

Figure 2: Accuracy vs k

Notice how the three plots have the same overall shape and they all reach a maximum when

using $k = 1$. To my surprise, it achieved a **100%** accuracy!

Now lets look at the time it takes to evaluate the classifier using the testing data set.

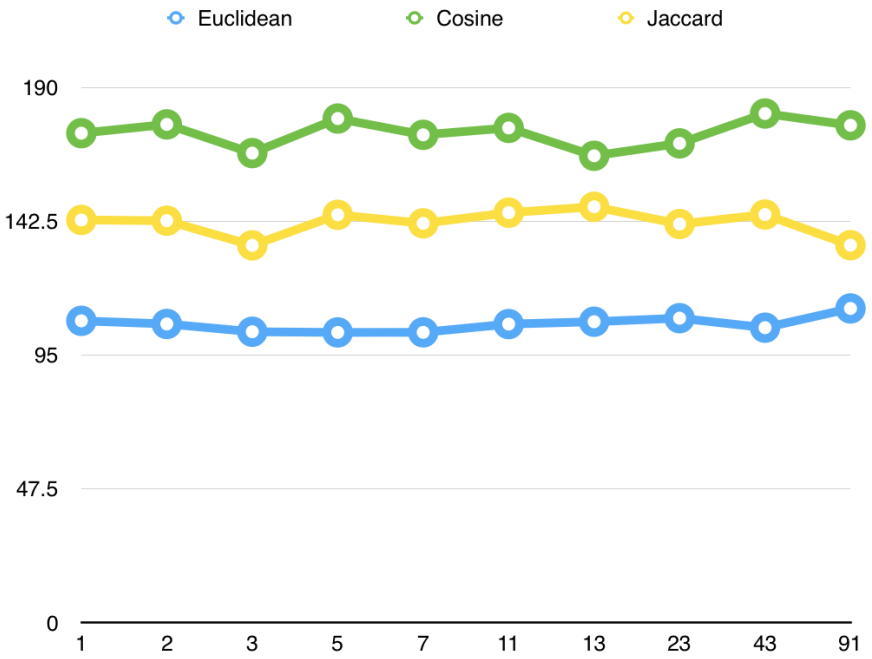

Figure 3: Testing time as a function of k

Out of the three distance measures Euclidean distance is the most efficient one when it comes to test time. Since all three of them achieve 100% I'll report the confusion matrix when using Euclidean distance. As expected, the confusion matrix would have 100 on its diagonal and 0 everywhere else.

		Predicted class									
		0	1	2	3	4	5	6	7	8	9
Class	0	100	0	0	0	0	0	0	0	0	0
	1	0	100	0	0	0	0	0	0	0	0
	2	0	0	100	0	0	0	0	0	0	0
	3	0	0	0	100	0	0	0	0	0	0
	4	0	0	0	0	100	0	0	0	0	0
	5	0	0	0	0	0	100	0	0	0	0
	6	0	0	0	0	0	0	100	0	0	0
	7	0	0	0	0	0	0	0	100	0	0
	8	0	0	0	0	0	0	0	0	100	0
	9	0	0	0	0	0	0	0	0	0	100

Table 3: Confusion matrix. Values are percentages.

One important observation is that the testing time is relatively stable given the distance measurement; meaning it doesn't change much as the number k increases. This makes sense because, in order to find the best k, you still have to calculate the distance against *all* the training examples. The different in time depends exclusively on the distance function.

Finally, it is obvious that k-nearest neighbor outperforms both my Naive Bayes and Perceptron classifiers by a wide margin and, when using the same features, Naive Bayes is the one with the lowest accuracy.

Extra credits

Extra Credit 1 - Advanced features: as done in Assignment 3, I'll now use features that encode a group of pixels of various sizes. The same sizes were used: 2x2, 2x3, 3x2 and 3x3.

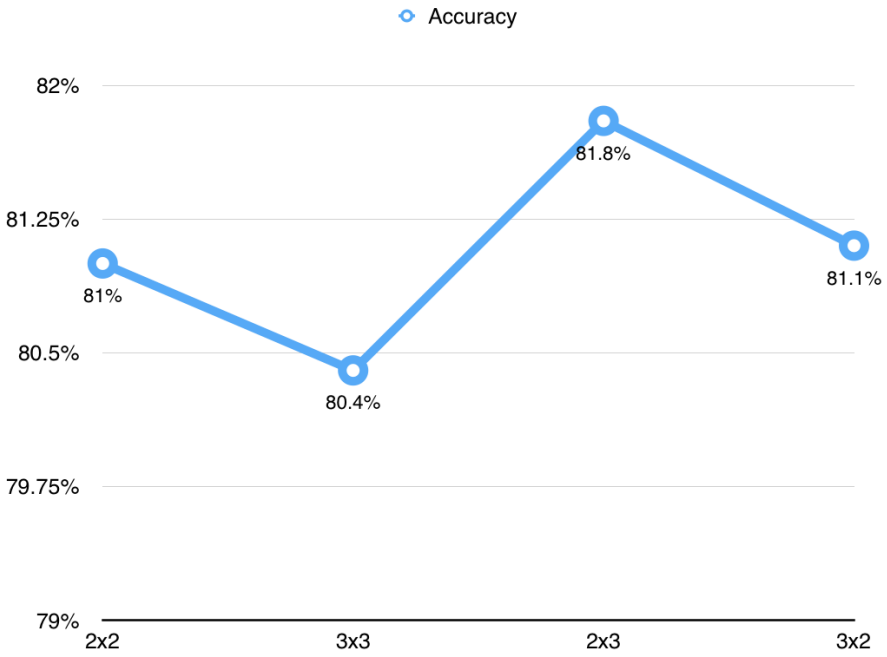

Figure 4: Accuracy for groups of pixels as features.

The highest accuracy when using these groups was **81.6%** which is still better than the accuracy obtained by Naive Bayes but does not improve the accuracy obtained by using single pixels. Training curves and confusion matrices can be found in the results folder.

Extra Credit 2 - Weights visualization: lets look at the color maps of the weights for each of the digits.

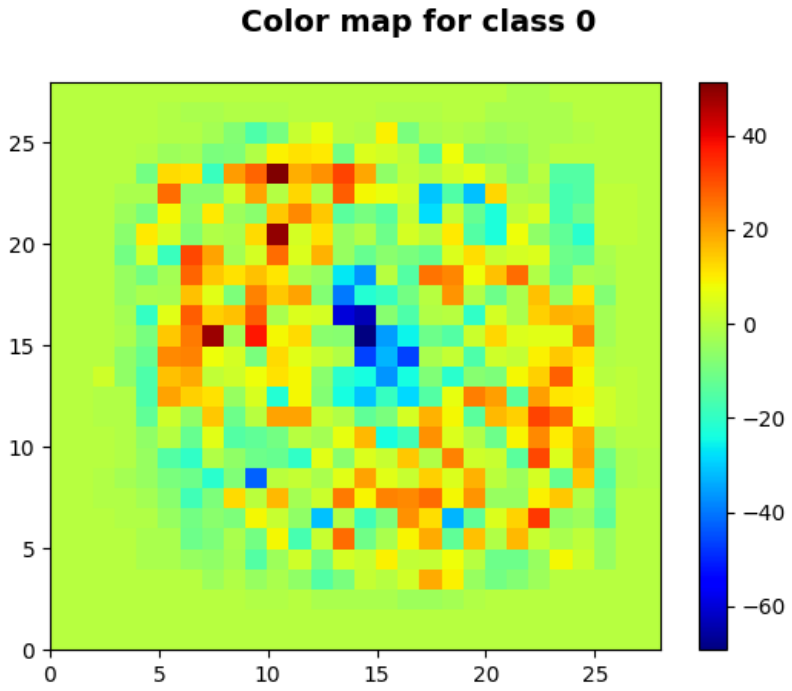

Figure 5: Color map for digit 0.

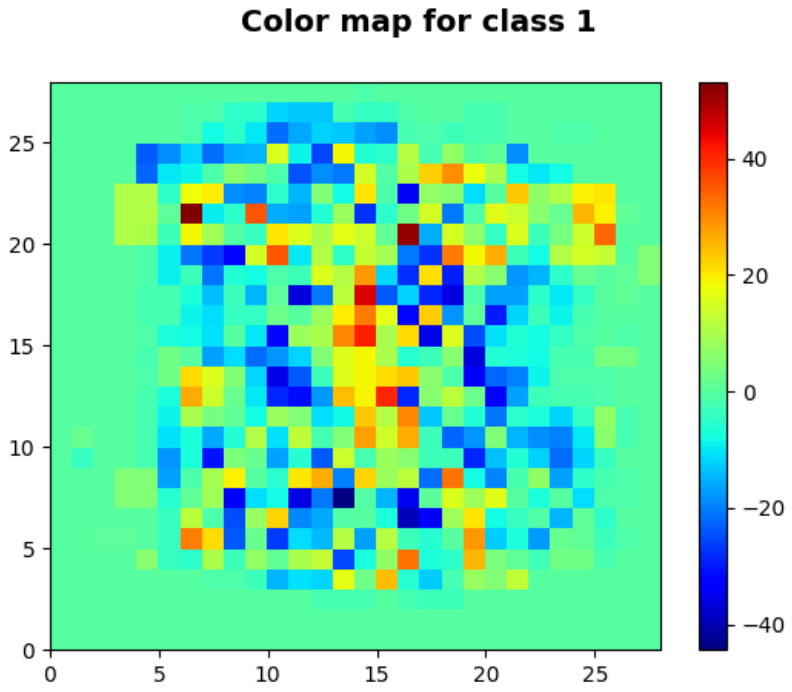

Figure 6: Color map for digit 1.

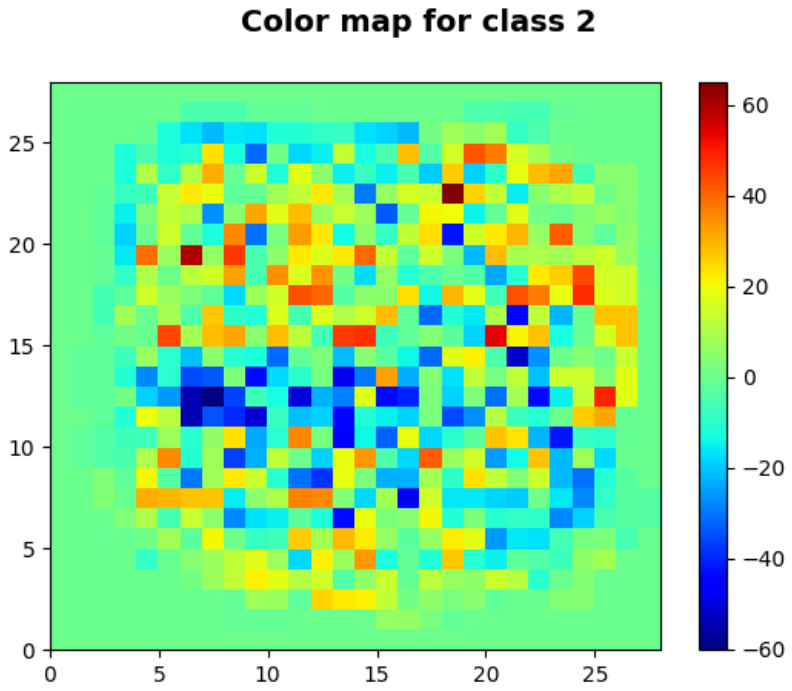

Figure 7: Color map for digit 2.

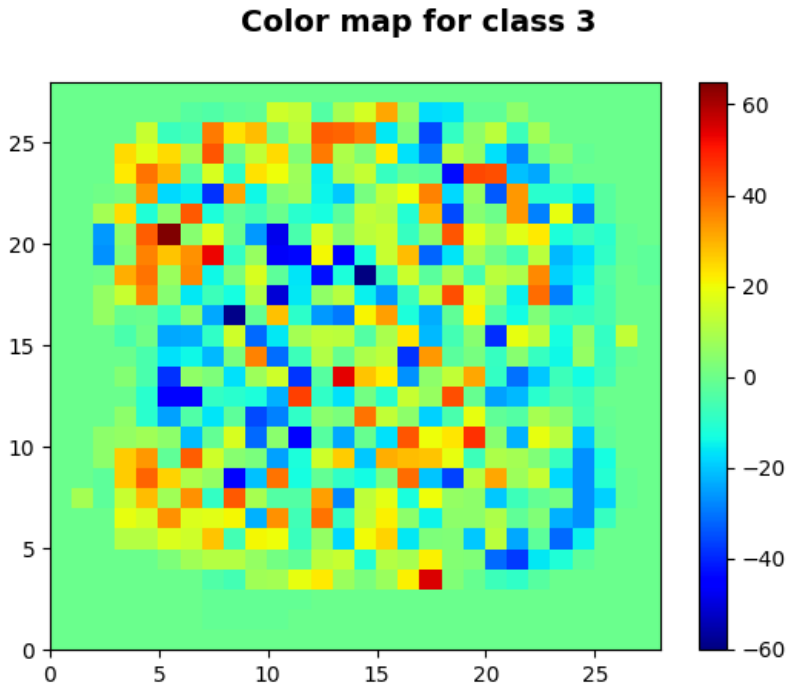

Figure 8: Color map for digit 3.

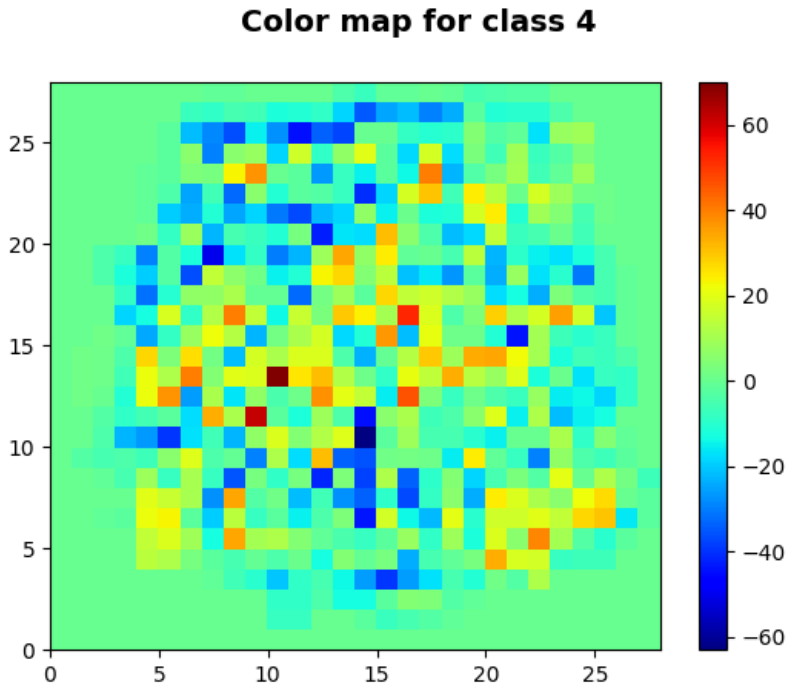

Figure 9: Color map for digit 4.

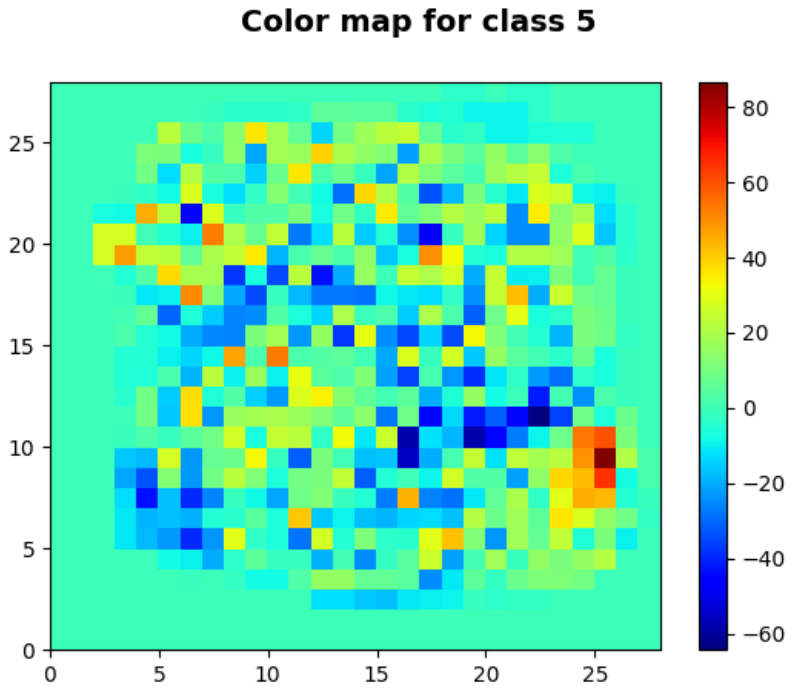

Figure 10: Color map for digit 5.

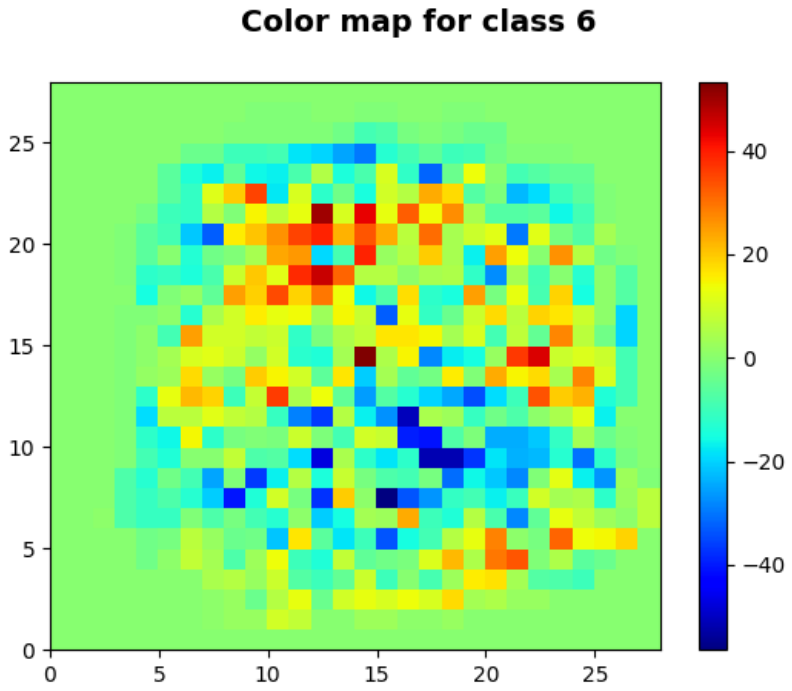

Figure 11: Color map for digit 6.

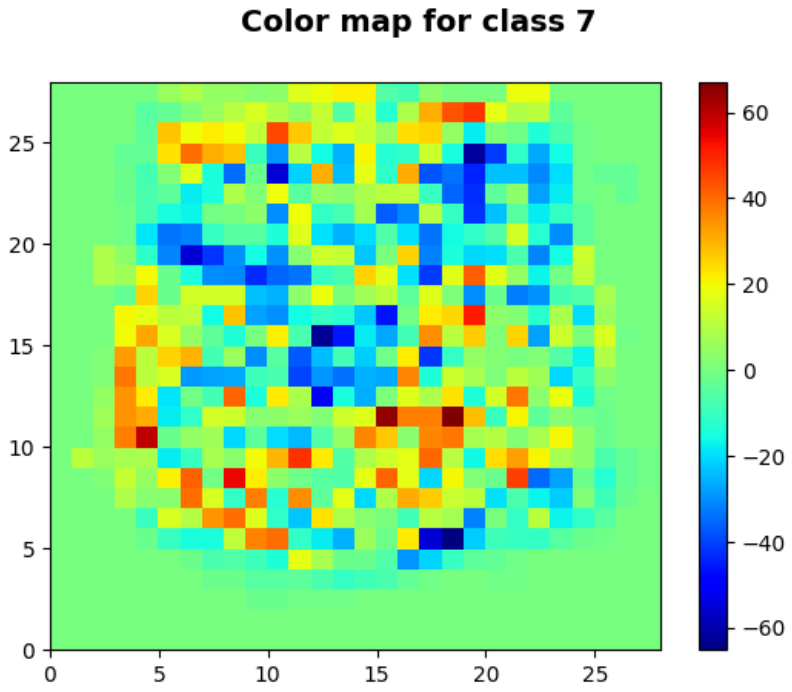

Figure 12: Color map for digit 7.

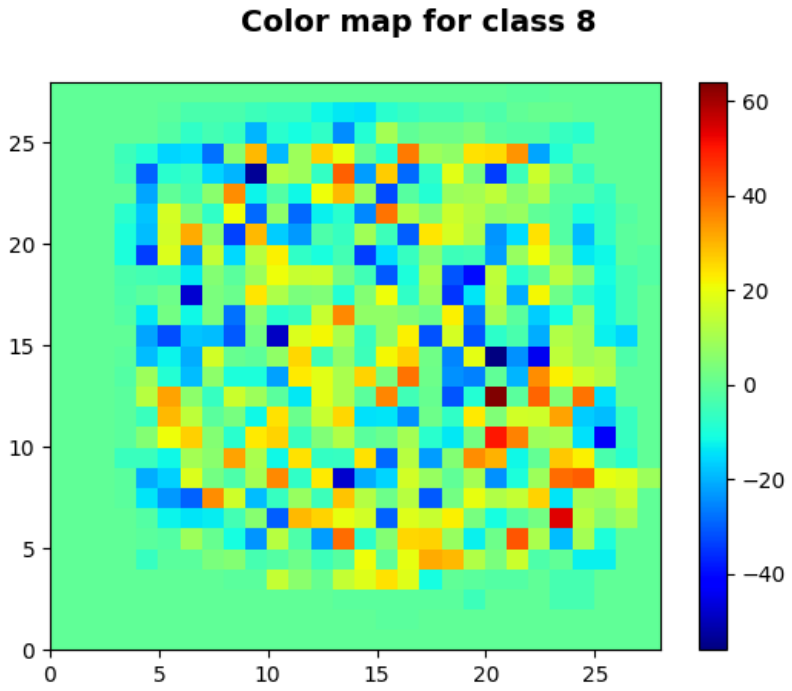

Figure 13: Color map for digit 8.

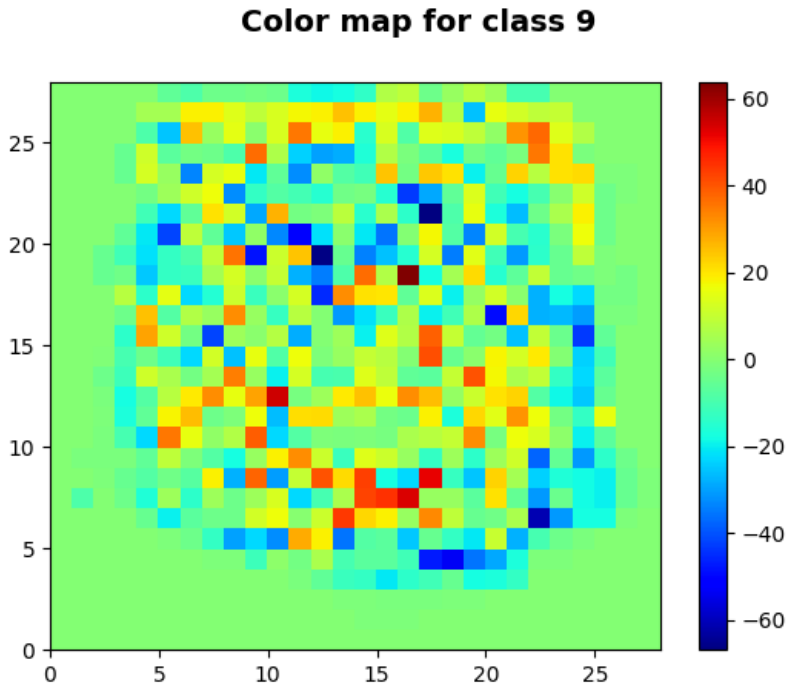

Figure 14: Color map for digit 9.

In this visualizations, the high heat spots represent features that when present are very important to classify an example as the expected class and, in contrast, the lowest spots (scales of blue) indicate features that when present can help determine that an example is NOT of the given class; it doesn't tell anything about the right class, simply says that it may not be an example of a particular class.

Extra Credit 3 - Differentiable weight update: I decided to use the sigmoid function and apply the weight update from the slides:

$$\vec{w} \leftarrow \vec{w} + \alpha(y - f(x))\sigma(\vec{w} \cdot \vec{x})(1 - \sigma(\vec{w} \cdot \vec{x})) \cdot \vec{x}$$

For calculating the σ function I used the **expit** function from SciPy[2].

A few things to note from the results obtained using gradient descent:

1. The training curve obtained using sigmoid is smoother than the non-differentiable perceptron. This is congruent with intuition because the derivative tells you the "right" way in which to change the weights. Below you will find the training curve for the sigmoid perceptron.
2. The accuracy on the training data is not as high when using sigmoid, compared to the Part 1.1 perceptron.

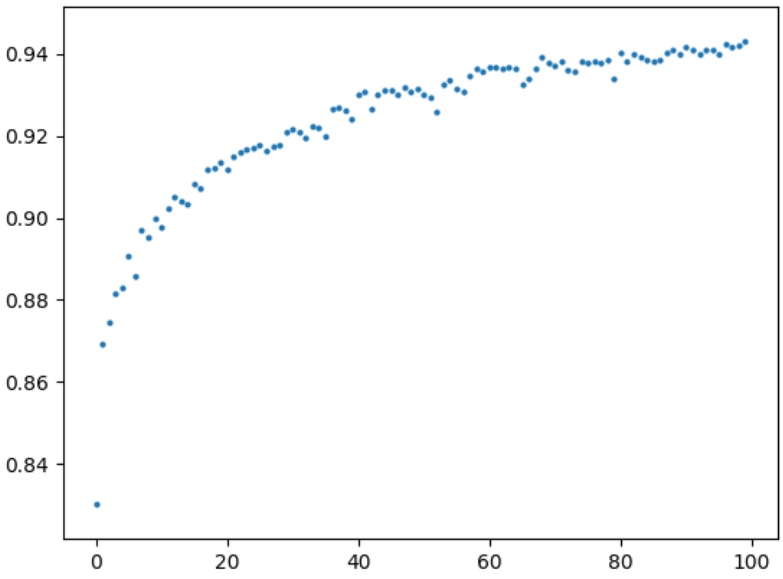

Figure 15: Training curve for the perceptron using σ.

[2] https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.special.expit.html

Finally, the best accuracy achived using sigmoid was **82%**, close to the best from Part 1.1. Additional experiments were performed trying to increase the accuracy without success. The attempts included increasing the number of epochs (this resulting in a higher accuracy for the training data but not considerable improvement in the test data), different α values and random weights initialization. The respective training curves can be found in the results folder but are not included here for brevity and because nothing more than a comparison was requested.

Extra Credit 4 - Additional classifiers: I decided to use scikit-learn[3] to implement an SVM classifier. In particular, I used the SVC method that implements the "one-vs-one" approach instead of the "one-vs-other" implemented in my Perceptron classifier. For reference you can see this. I tried different kernel functions to see how they impacted the accuracy of the model. There are four built-in kernel functions for SVC: rbf, sigmoid, linear and polynomial.

The following table summarizes the outcome of the different tries:

Kernel	Accuracy	Training time	Testing time
rbf	87.8%	9.29	2.36
sigmoid	86.1%	11.92	2.84
poly	17.3%	34.98	4.04
linear	87.5%	4.78	1.55

Figure 16: Summary of the accuracies for SVC classification.

Particularly interesting is the fact that a polynomial kernel gives a very bad accuracy while the other ones outperform my Perceptron classifier but not by a large margin.

Part 2

In this assignment we will create agents for the Pong game. First we will develop a trained agent using the Q-learning algorithm to learn a policy to best play the game. In the second part we will develop a second agent that tries to follow the ball as close as possible and we will compare it against the trained agent.

The code is structured in the following classes:

1. **PongMDP**: it receives the initial configuration of the game (ball position and velocity and paddle position), provides methods to discretize the state, find the best action to take given a state and carry out such action.

2. **QLearningAgent**: it contains the actual implementation of the Q-learning algorithm. It takes a **PongMDP** as argument and then provides methods to train and evaluate

[3] http://scikit-learn.org/stable/

it. Additionally it implements a *save* method that uses **pickle**[4] to create a file that can be later loaded.

3. **Util**: to plot training curve
4. **Action**: holds an enumeration of the three possible actions available to the agents.

Part 1.1

I tried two different exploration functions: the first was a function of the number of times a particular state-action pair had been seen; I never got it to work properly (the agent never learned anything) so I tried the second option: an ϵ-greedy method taken from Macello Restelli's lecture notes.

Report and justify parameters

My implementation includes the following hyper-parameters:

1. ϵ: a parameter for the exploration function.
2. **C**: the parameter for the learning rate decay function α of N_{sa}
3. γ: the discount factor
4. **n**: number of games until convergence.

The selected values were: $\epsilon = 0.05$, **C**=5, $\gamma = 0.8$, **n**=100,000.

The value selected for ϵ (**0.05**) for my exploration function was selected based on the discussion on this StackExchange thread.

The number of games was not much of a guess. First I tried 100,000 as recommended by the assignment; with this and the other parameters the performance was good enough. For the sake of experimentation I also tried 230,000 runs. The results can be found in the following sub-section.

Finally, the value for **C** and γ were found doing some basic grid search. For **C** 3 values were tested: 0.1, 1 and 5. The value **5** was chosen because it caused the average number of catches to increase more rapidly than the other two. For γ 4 values were tried: 0.5, 0.4, 0.6, 0.8, in that order. Finally **0.8** was chosen because it sped up the training.

The training reached an average of at least 9 hits after **26,000** games played.

[4] https://docs.python.org/3/library/pickle.html

Evaluation results

Every 1000 iterations I saved the model so that I could later evaluate the performance at different points. The following table summarizes the different evaluation points: 7,000; 26,000; 99,000; 100,000 and 230,000 games played. For evaluation the model was not learning and it would always pick the best Q-value action (no exploration). The evaluation was performed with 10,000 games.

		Metrics		
		Mean	Max	Min
# of games	7,000	9.7889	78	1
	26,000	14.5355	96	1
	99,000	15.0873	119	1
	100,000	14.4425	101	1
	230,000	13.8675	94	1

Table 4: Metrics (avg. number of hits) for different number of games during training.

The following graph shows the average number of hits while training:

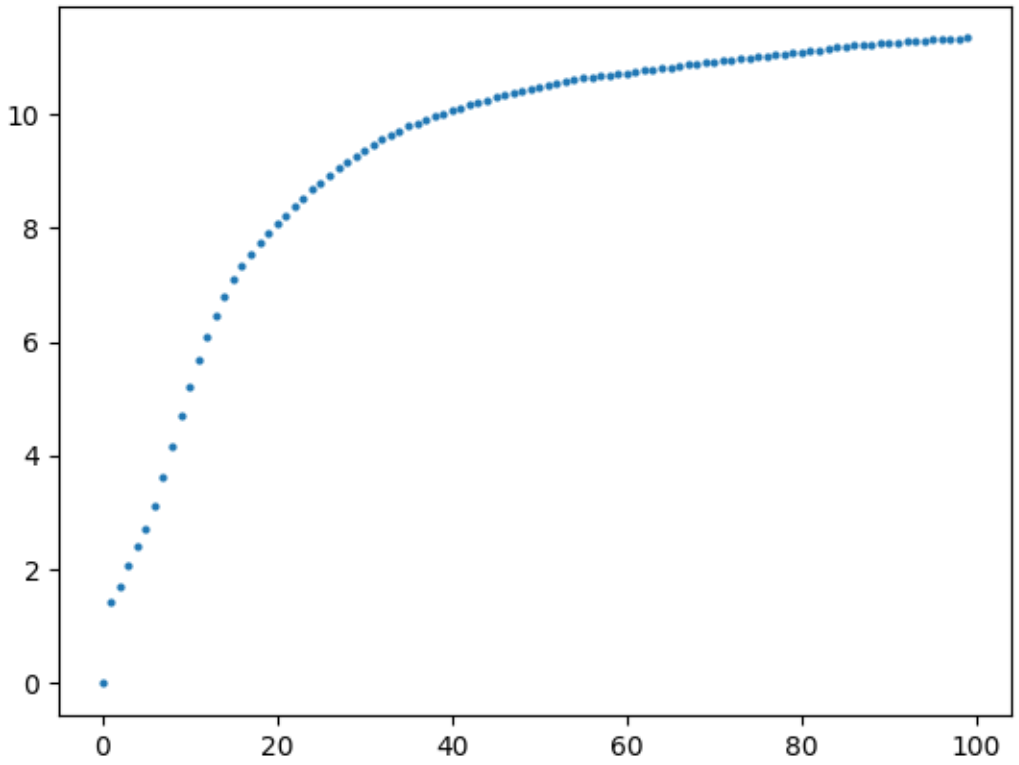

Figure 17: Avg. hits as the number of games for training increased. x-axis scaled by 1000.

Discretization

To implement the discretization procedure I simply added a new function, *as_discrete*, to the **PongMDP** class. The method takes the grid size as an argument. The process itself was simple since all the constraints were given in the assignment.

I tried multiple grid sizes over 10,000 games. The grid sizes used were: 8x8, 12x12, 24x24 and 144x144. The training time remained roughly constant at 32 seconds but the speed at which the avg. number of hits increased was greatly impacted: as the size of the grid grew the increment in the number of hits was slower. The following plots show this.

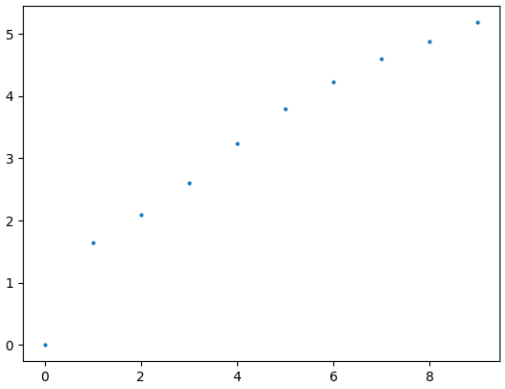

Figure 18: Hits vs # of games: 8x8.

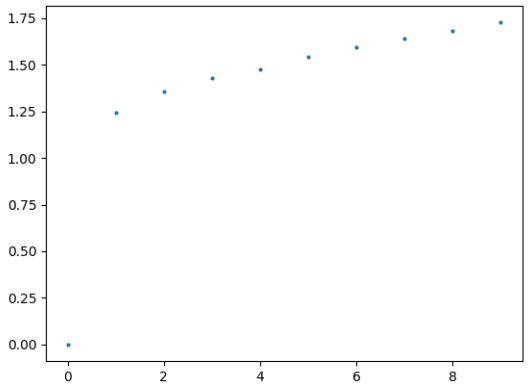

Figure 20: Hits vs # of games: 24x24.

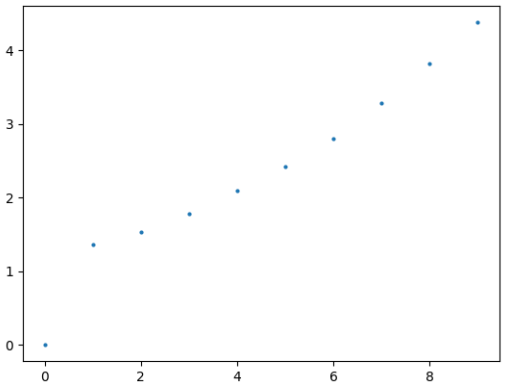

Figure 19: Hits vs # of games: 12x12.

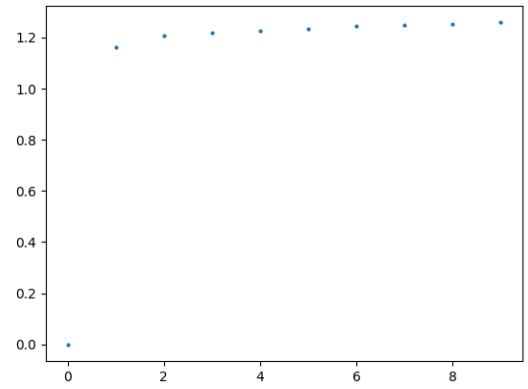

Figure 21: Hits vs # of games: 144x144.

Intuitively this makes sense since this is causing the state space to increase and therefore the changes of not having observed an action increase unless the number of games for training is increased.

144x144 grid size results

Using the grid size 144x144 I trained the agent for 100,000 games and then evaluated it over 10,000 games. During training the maximum average hit was 1.5 even after 100,000 games. During evaluation (exploration turned off) it only achieved 2.5 hits per game on average.

Part 2.2

Implementation changes

1. Update **QLearningAgent** to accept as a initialization parameter, that way **PongMDP** can be constructed outside and shared between the two agents.
2. Add a function *next_action* to the Agent classes to retrieve the best action without executing it.
3. Add a **Gameplay** class that instantiates a match between the two agents and determines when one of them have won.
4. The MDP *carry_out* function now expects two actions. One for each agent. That way it can transition correctly. For this to be possible the state now has an additional element called *paddle_y_b* and the MDP also has some other internal variables for the paddle step (0.02) and the x position of the paddle (0). The *carry_out* function was updated to consider these values, duplicate similar code for bouncing the ball.
5. The MDP *carry_out* now returns who the winner was, if there was one in the current move, instead of the reward of the move.

Note that the discretization process didn't change because it is only used by the **QLearningAgent** and, for it, there is no difference between the ball bouncing because there is a wall or because it hit the opponent's paddle. Even more, if we see this as a problem to use expected-minimax then the Q-learning agent can assume that that the hardcoded agent will always catch the ball and, therefore behaving like the wall from the viewpoint of the Q-learning agent.

Because of this, there is no side-effect if introducing the hard-coded agent.

Results

I performed the following tests (using the models saved every 1000 games):

1. agent trained for 2,000 games: It achieved a win rate of **48%**
2. agent trained for 4,000 games: It achieved a win rate of **62.5%**
3. agent trained for 8,000 games: It achieved a win rate of **92%**
4. agent trained for 100,000 games: It achieved a win rate of **96.78%**.
5. agent trained for 230,000 games: It achieved a win rate of **96.5%**