Homework 8

CS 559 - Neural Networks - Fall 2019

Matteo Corain 650088272

December 3, 2019

1 Question 1

1.1 Centroids generation

The centroids generation step is carried out using the provided implementation of the k-means algorithm, where $k = 10$ both for the positive class and the negative class (20 centers in total were generated). The `kmeans()` procedure has been coded for this purpose, receiving as arguments the value of k and the list of training samples x_i, $i = 1, \ldots, 100$, randomly generated in $[0, 1]^2$. This performs the following steps:

- It initializes the set of centroids by drawing k random points from the input space, storing them in the `centers` list;
- It computes the nearest centroid for each sample, storing the associations in the `classes` list;
- It starts a loop until the `unchanged` flag goes to zero; for each iteration:
  - It recomputes each centroid by averaging all the points in the associated Voronoi region, defined by the points that were assigned to that centroid in the previous iteration; if no points are associated to the considered centroid (operation that returns a division by zero error), a new centroid is randomly initialized;
  - It computes the new nearest centroid for each sample, storing the associations in the `new_classes` list;
  - It checks whether the `classes` and `new_classes` lists are the same and set the result of the comparison to the value of `unchanged`;
  - It copies back the new classes on the `classes` list.
- At the end of the loop, the procedure returns the list of centers and the index of the closest center for each sample.

The `kmeans()` procedure is applied separately to the list of points in the positive and negative classes; finally, an overall list of centroids is produced by concatenation of the returned results. The obtained clustering is shown in figure 1.

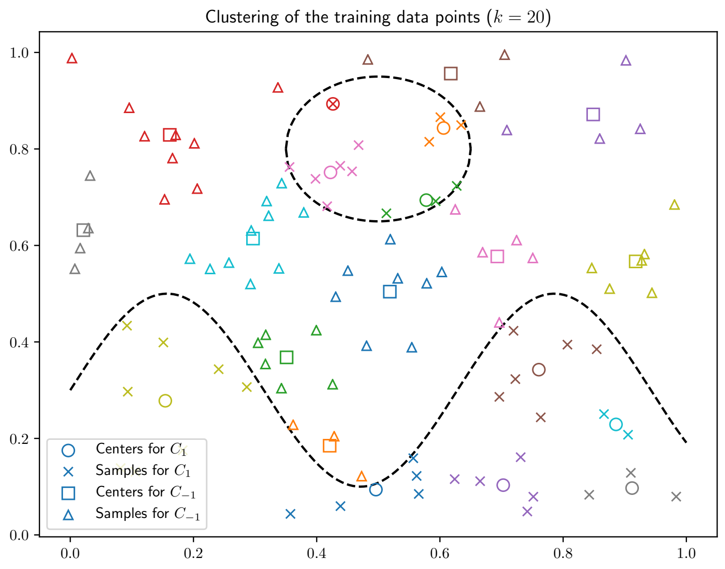

Figure 1: Clustering of the training data points ($k = 20$)

1.2 Radial basis function mapping

The centroids computed at the previous steps have been used to map the original data points into an appropriate higher dimensional space, by means of an opportune radial basis function $\phi(x, \mu)$. For the sake of the exercise, a Gaussian RBF has been used:

$$\phi(x, \mu) = e^{-\beta||x-\mu||^2}$$

Where β is a free parameter (sometimes defined in terms of the standard deviation of the data), x is one of the considered samples and μ is one of the computed centroids. In this case, each of the 2-dimensional samples was mapped on a 20-dimensional vector defined as:

$$x' = \begin{bmatrix} \phi(x, \mu_1) \\ \vdots \\ \phi(x, \mu_{20}) \end{bmatrix} = \begin{bmatrix} e^{-\beta||x-\mu_1||^2} \\ \vdots \\ e^{-\beta||x-\mu_{20}||^2} \end{bmatrix}$$

As for the parameter β, two possible values were tested ($\beta_1 = 1, \beta_2 = 10$) in order to analyze its effect on the overall results. In code, the mapping procedure is implemented in the `rbfmap()` function, which takes as arguments the sample to map, the list of centroids and the value of β.

1.3 Perceptron training

The RBF-mapped representations of the original samples have been used for the supervised training a perceptron with 20 inputs (21, if we consider also the bias-related input $x_0 = 1$), in the hope that they were linearly separable in the RBF space. For the training step, a mostly standard perceptron training algorithm has been used, with a few notable differences:

- Since in the original version of the PTA the considered perceptron use step activation (while here it uses signum activation), the shorthand formula for weight update was split into two separate update rules:
  - The first is applied when $d_i = 1$ but $y_i = \operatorname{sgn}(wx_i) = -1$:
  $$w \leftarrow w + \eta x_i$$
  - The other is applied when $d_i = -1$ but $y_i = \operatorname{sgn}(wx_i) = 1$:
  $$w \leftarrow w - \eta x_i$$

  In this way, we avoid the introduction of an additional factor of 2 which may cause convergence problems by effectively doubling the value of the configured learning rate.
- An update rule for η, which reduces the value of the learning rate of a 0.9 factor when the errors increase from an epoch to the other, has also been introduced for avoiding oscillations and allowing for faster convergence.

The perceptron training algorithm has been coded in the `pta()` function, receiving as arguments the value of the initial learning rate, the initial weights vector, the training set (whose elements have the first component set to 1 to account for the bias-related input), the expected outputs and the epoch limit value after which the algorithm terminates even if convergence has not been reached; the procedure returns the trained weights vector, the number of training epochs and the number of misclassifications per epoch.

1.4 Results

A first attempt was initially performed by setting parameters β to 1, η to 0.1, epoch limit to 5000 and by drawing the initial weights for the PTA from a uniform distribution in $[-1, 1]$. In this case, however, the algorithm was not able to converge within the configured epoch limit, remaining fairly stable at around 30 misclassifications after the first epochs. Consequently, the obtained boundary (shown in figure 2) was very different from the expected one, presenting a large number of misclassified items from both classes.

The conclusion we can draw from this is that our RBF mapping is not good enough to make our patterns separable in the RBF space. In particular, we can notice that, since our training samples are drawn from $[0, 1]^2$, the maximum distance between a point and a centroid can be no more than $\sqrt{2}$ (when they are in two opposite vertices of the considered region), and in practice it is generally much less than that (it is very unlikely for such a distribution of data to happen). When we map our data points into the RBF space, each feature x'_i of the mapped point will consequently satisfy the following inequality:

$$e^{-\max ||x-\mu||^2} = e^{-2} = 0.1353 \leq x'_i \leq 1 = e^0 = e^{-\min ||x-\mu||^2}$$

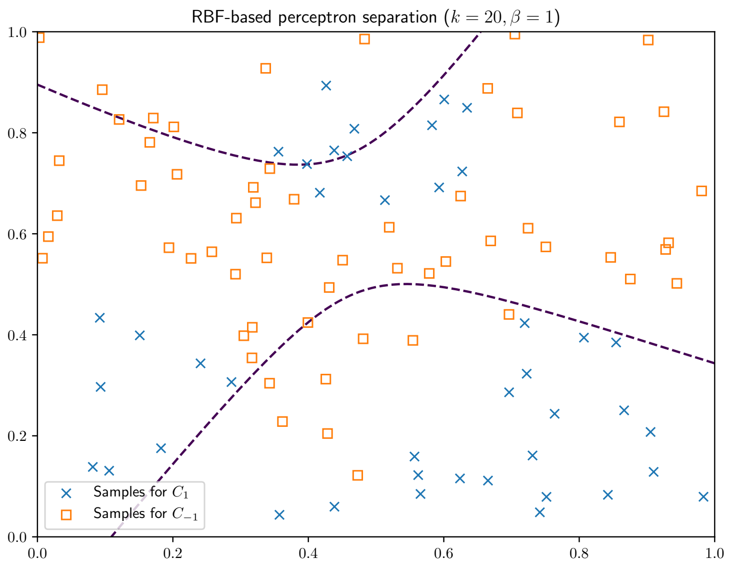

Figure 2: RBF-based perceptron separation ($k = 20$, $\beta = 1$)

In other words, the mapped data points be confined in the $[e^{-2}, 1]^{20}$ sub-region of the "complete" mapping space $(0, 1]^{20}$ (0 is excluded since we assume that $\max ||x - \mu||^2 < \infty$), effectively losing a very large portion of the RBF space. For this reason, we have to reconsider the way our patterns are mapped to the RBF space; an obvious choice of intervention is on parameter β, which in the second run was significantly increased to the value of 10. In this way, almost the entire mapping space can be used (since $e^{-20} = 2 * 10^{-9}$, leading to a very limited loss in terms of excluded volume); thus, we can expect that our patterns will be more likely to be linearly separable.

Running the script using $\beta = 10$ and leaving all the other parameters to their previous values, PTA was able to converge in a relatively small number of epochs (90), sign that, by exploiting almost the entirety of our mapping space, our patterns become clearly separable by a linear classifier. Results for this second run are shown in figure 3.

1.5 Differences with the case $k = 4$

A third run was finally performed, using the same value of $\beta = 10$ as before but setting the value of k to 2 for both runs of the k-means algorithm (for the positive and the negative classes). Again, the PTA was not able to reach convergence within the configured epoch limit, showing large oscillations in the number of misclassifications throughout the different epochs.

From this, we can conclude that, in this case as well, our mapping is not good enough to make the classes linearly separable; this is also reflected by the obtained separator, which appears to be significantly different from the original one and not able to divide the data points clearly. Results of the k-means clustering procedure and the separation obtained by running PTA are shown in figures 4 and 5.

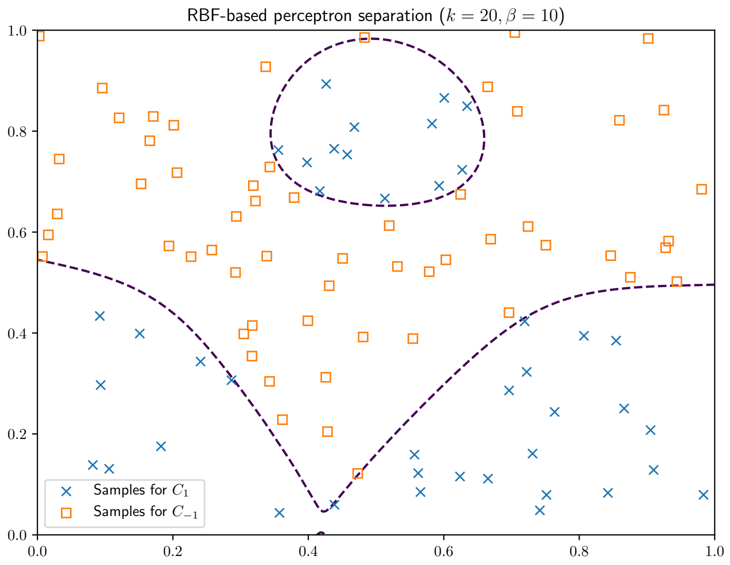

Figure 3: RBF-based perceptron separation ($k = 20$, $\beta = 10$)

1.6 Complete Python code

```
import numpy as np
import matplotlib.pyplot as plt

# Use TeX for text rendering
plt.rc('text', usetex=True)

# Set the random seed for reproducibility
np.random.seed(123)

def rbfmap(x, centers, beta):
    # Map x on its Gaussian RBF representation
    return np.array([np.exp(-beta * np.linalg.norm(x - centers[i]) ** 2) for i in range(len(centers))])

def classify(x, w):
    # Classify a data point from the given weights
    return np.sign(w @ x)

def count_mclass(x, d, w):
    # Count misclassifications with the given params
    return sum([1 if classify(x[i], w) != d[i] else 0 for i in range(len(x))])

def kmeans(k, x):
    print("K-Means started.")

    # Initialize random centroids
    centers = [np.random.uniform(0, 1, size=2) for i in range(k)]

    # Associate each point with its nearest centroid
    classes = [np.argmin([np.linalg.norm(xi - ci) for ci in centers]) for xi in x]
```

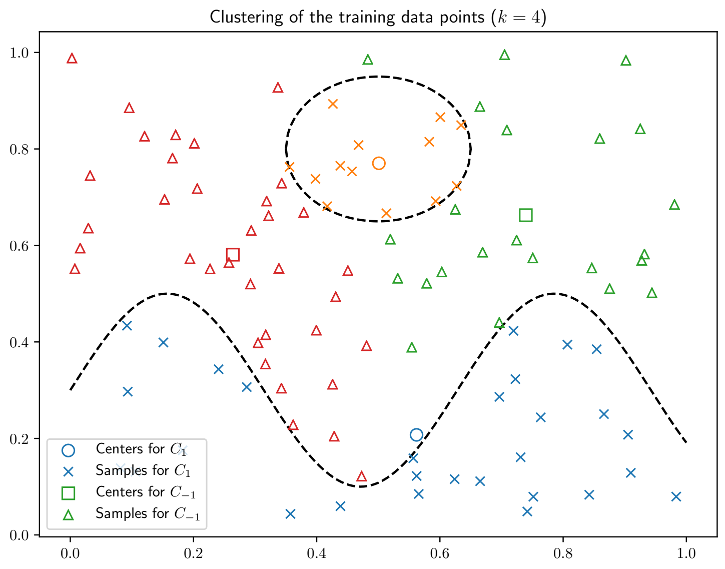

Figure 4: Clustering of the training data points ($k = 4$)

```
    # Loop until no changes
    i = 0
    unchanged = False

    while not unchanged:
        # Compute the next centroids
        for j in range(k):
            try:
                # Average all points in the Voronoi region
                centers[j] = sum([x[i] for i in range(len(x)) if classes[i] == j]) / len([x[i] for i in range(len
                    (x)) if classes[i] == j])
            except ZeroDivisionError:
                # If the Voronoi region is empty, initialize a random centroid
                centers[j] = np.random.uniform(0, 1, size=2)

        print(f"Iteration {i+1}: Centers = {[np.array2string(x) for x in centers]}")
        i = i + 1

        # Associate each point with its nearest centroid
        new_classes = [np.argmin([np.linalg.norm(xi - ci) for ci in centers]) for xi in x]

        # Check if classification is unchanged
        unchanged = classes == new_classes

        # Update classes
        classes = new_classes

    return centers, classes

def pta(eta, w0, x, d, limit):
    print("PTA started.")
```

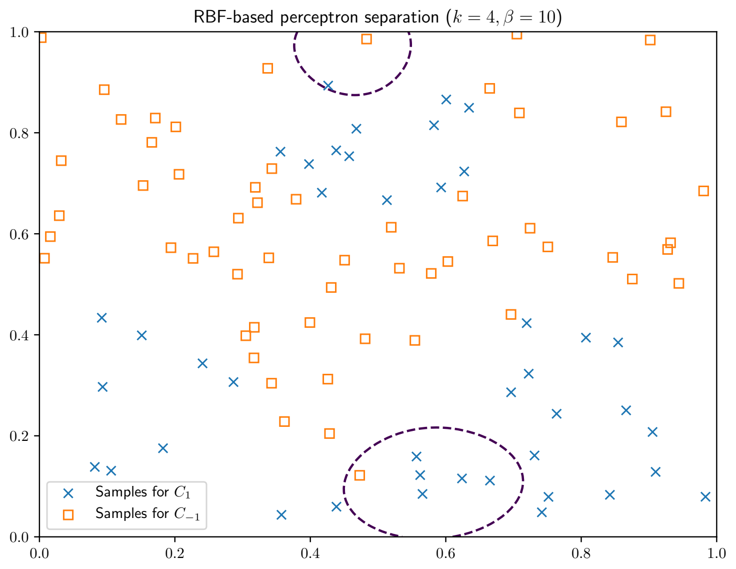

Figure 5: RBF-based perceptron separation ($k = 4$, $\beta = 10$)

```
    # Initialize PTA variables
    w = np.array(w0)
    epochs = 0

    # Initialize mclass array
    mclass = []
    mclass.append(count_mclass(x, d, w))

    # Run the PTA algorithm
    while mclass[-1] != 0 and epochs < limit:
        # Increase epochs
        epochs = epochs + 1

        # Loop through samples and update weights
        for i in range(len(x)):
            if d[i] == 1 and classify(x[i], w) == -1:
                w = w + eta * x[i]
            elif d[i] == -1 and classify(x[i], w) == 1:
                w = w - eta * x[i]

        # Count misclassifications
        mclass.append(count_mclass(x, d, w))

        # Reduce eta if errors are increasing
        if mclass[-1] > mclass[-2]:
            eta = 0.9 * eta

        print("Epoch {}: Weights = {} - Misclassifications = {} - Eta = {}".format(epochs, w, mclass[-1], eta))

    return w, epochs, mclass

def plot_data(n, xpos, xneg, d, k, pcenters, pclasses, ncenters, nclasses):
    # Create a new figure
    plt.figure(figsize=(8,6))

    # Plot ideal separator
    plt.plot(np.linspace(0, 1, 1000), [0.2 * np.sin(10 * x) + 0.3 for x in np.linspace(0, 1, 1000)], linestyle=
         "--", color="k")
    plt.plot(np.linspace(0.35, 0.65, 100), [np.sqrt(abs(0.15 ** 2 - (x - 0.5) ** 2)) + 0.8 for x in np.linspace
         (0.35, 0.65, 100)], linestyle="--", color="k")
    plt.plot(np.linspace(0.35, 0.65, 100), [-np.sqrt(abs(0.15 ** 2 - (x - 0.5) ** 2)) + 0.8 for x in np.
         linspace(0.35, 0.65, 100)], linestyle="--", color="k")

    # Plot positive K-means results
    for j in range(k):
        color = next(plt.gca()._get_lines.prop_cycler)['color']
        if j == 0:
            plt.plot([pcenters[j][0]], [pcenters[j][1]], linestyle="none", marker="o", fillstyle="none", color=
                 color, markersize=8, label="Centers for $C_{1}$")
            plt.plot([xpos[i][0] for i in range(len(xpos)) if pclasses[i] == j], [xpos[i][1] for i in range(len(
                 xpos)) if pclasses[i] == j], linestyle="none", marker="x", fillstyle="none", color=color, label
                 ="Samples for $C_{1}$")
        else:
            plt.plot([pcenters[j][0]], [pcenters[j][1]], linestyle="none", marker="o", fillstyle="none", color=
                 color, markersize=8)
            plt.plot([xpos[i][0] for i in range(len(xpos)) if pclasses[i] == j], [xpos[i][1] for i in range(len(
                 xpos)) if pclasses[i] == j], linestyle="none", marker="x", fillstyle="none", color=color)

    # Plot negative K-means results
    for j in range(k):
        color = next(plt.gca()._get_lines.prop_cycler)['color']
        if j == 0:
            plt.plot([ncenters[j][0]], [ncenters[j][1]], linestyle="none", marker="s", fillstyle="none", color=
                 color, markersize=8, label="Centers for $C_{-1}$")
            plt.plot([xneg[i][0] for i in range(len(xneg)) if nclasses[i] == j], [xneg[i][1] for i in range(len(
                 xneg)) if nclasses[i] == j], linestyle="none", marker="^", fillstyle="none", color=color, label
                 ="Samples for $C_{-1}$")
        else:
            plt.plot([ncenters[j][0]], [ncenters[j][1]], linestyle="none", marker="s", fillstyle="none", color=
                 color, markersize=8)
            plt.plot([xneg[i][0] for i in range(len(xneg)) if nclasses[i] == j], [xneg[i][1] for i in range(len(
                 xneg)) if nclasses[i] == j], linestyle="none", marker="^", fillstyle="none", color=color)

    # Decorate plot
    plt.title(f"Clustering of the training data points ($k = {2*k}$)")
    plt.legend()
    plt.savefig(fname="data.pdf")

def plot_sep(n, x, d, w, k, centers, beta, eval_pts):
    # Create a new figure
    plt.figure(figsize=(8,6))

    # Plot the training set
    plt.plot([x[i][0] for i in range(n) if d[i] == 1], [x[i][1] for i in range(n) if d[i] == 1], linestyle="
         none", marker="x", fillstyle="none")
    plt.plot([x[i][0] for i in range(n) if d[i] == -1], [x[i][1] for i in range(n) if d[i] == -1], linestyle="
         none", marker="s", fillstyle="none")

    # Evaluate the separator function on a eval_pts*eval_pts grid
    evals = np.zeros((eval_pts, eval_pts))
    for i in range(eval_pts):
        for j in range(eval_pts):
            evals[j][i] = w @ np.concatenate([[1], rbfmap([i / eval_pts, j / eval_pts], centers, beta)])
        print(f"Separator evaluation: {i+1}/{eval_pts}")
    # Plot the separator
    xx, yy = np.meshgrid(np.linspace(0, 1, eval_pts), np.linspace(0, 1, eval_pts))
    plt.contour(xx, yy, evals, levels=[0], linestyles="dashed")

    # Decorate plot
    plt.legend(["Samples for $C_{1}$", "Samples for $C_{-1}$"])
    plt.title(f"RBF-based perceptron separation ($k = {2*k}, \\beta = {beta}$)")
    plt.savefig(fname="sep.pdf")

# Generate the training set
n = 100
x = [np.random.uniform(0, 1, size=2) for i in range(n)]
d = [1 if x[i][1] < 0.2 * np.sin(10 * x[i][0]) + 0.3 or (x[i][1] - 0.8) ** 2 + (x[i][0] - 0.5) ** 2 < 0.15 **
     2 else -1 for i in range(n)]

# Define the two classes
xpos = [x[i] for i in range(n) if d[i] == 1]
xneg = [x[i] for i in range(n) if d[i] == -1]

# Run the K-means algorithm for the two classes
k = 10
pcenters, pclasses = kmeans(k, xpos)
ncenters, nclasses = kmeans(k, xneg)
centers = pcenters + ncenters

# Plot the training set
plot_data(n, xpos, xneg, d, k, pcenters, pclasses, ncenters, nclasses)

# Map training data on their RBF representation (adding bias component)
beta = 10
xrbf = [rbfmap(x[i], centers, beta) for i in range(len(x))]
xrbf_bias = [np.concatenate([[1], xrbf[i]]) for i in range(len(x))]

# Run PTA
eta = 0.1
w0 = np.random.uniform(-1, 1, size=(2*k + 1))
w, epochs, mclass = pta(eta, w0, xrbf_bias, d, 5000)

# Plot obtained separation
plot_sep(n, x, d, w, k, centers, beta, 250)
```