Reminder: the course material is at : `https://gitlab.inria.fr/flandes/fpml`

Note: we won't do all exercises of the exercise sheets in the classroom. Many of the additional exercises proposed here won't be discussed, but you can **use these exercises to train yourself**. Typically the corrections won't be given, but if you do them and have questions about these additional exercises, you can ask us ! Having many examples of exercies is **usually useful to prepare for the exam**.

2 Classifications

2.1 Perceptrons

2.1.1 Perceptron – classics

This is the classic Perceptron with a ReLu activation function. The ReLU is

$$ReLU(z) = max(0, z) \tag{1}$$

0. Write down the **model** $f_\Theta(\vec{x})$? Write down the network's **output** $\hat{y}$. Write down the **cost function** $J(\Theta, X, T)$. You may read the slides. You should use the augmented X trick (where we add ones to X, see exercise 1.2.1

CORRECTION We have the model $f_\Theta(\vec{x}_n) = \vec{w}\vec{x}_n$*, the output* $\hat{y}_n = sign(f_\Theta(\vec{x}_n))$ *and the cost function is* $J = \frac{1}{N}\sum_n(ReLU(-\vec{w}\vec{x}_n t_n))$

How many parameters are there in this model ? (As a function of N and d, the number of samples and the dimensionality of each data point).

CORRECTION There are $d + 1$ *parameters in the model, if we count the bias separately.*

1. Compute $\vec{\nabla}_\Theta J$. If it's hard, you can start by deriving the case of the stupid cost function J_2 that was discarded in the lecture (it's bad but at least it's easy to compute). For the RELU case, there are two cases, depending on the sign of the argument in the ReLU function. Try to write the result also in pseudo-code, using the concept of boolean filters (or python *slices*), that numpy is pretty good at handling.

CORRECTION The result is

$$\vec{\nabla}_\Theta J = \vec{\nabla}_\Theta \frac{1}{N}\sum_n (ReLU(-\vec{w}\vec{x}_n t_n)) \tag{2}$$

$$= \frac{1}{N}\sum_{misclassified} (-\vec{x}_n t_n) \tag{3}$$

$$= -\frac{1}{N} T[\vec{m}] \cdot X[\vec{m}], \tag{4}$$

where $\vec{m}$ *is the vector of the indices of the misclassified data points, i.e. the* n*'s such that* $\vec{w}\vec{x}_n t_n \leq 0$*. Thus* $\vec{m}$ *can be built, in numpy's convention, like* $\vec{m} = (X.\vec{w}) * T \leq 0$*, where* $*$ *is the point-wise mutiplication and not a dot product.*

2. Derive the update rule of the parameters, assuming we perform a Gradient Descent with learning rate η, and a *full batch* strategy.

CORRECTION The GD goes: $\vec{w} \mapsto \vec{w} + \eta \frac{1}{N} T[\vec{m}] \cdot X[\vec{m}]$

3. Write down on *a piece of paper* (not code directly) the pseudo code of the learning algorithm (initialization, updates, stopping criterion, error computation).

Use as many matrix operations (or dot product) as you can, and boolean filters/slices.

CORRECTION First, let's recall that the data structures are:

$X_{(N,D)}$*: training data*

$w_{0(D)}$*: initial weight vector*

$w_{(D)}$*: weight vector (current value).*

Here is a numpy-style version, very comapct and close to the mathematical formulation. There is also another solution, much more compiled-langage oriented, easier to read for some people.

Algorithm 1 full Batch Perceptron, **numpy** style

```
procedure FullBatchLearning(X, y, η, w_0, maxIter)
    w = w_0
    for iter in [1,...,MaxIter] do
        Filter = (X.w) * T ≤ 0                ▷ boolean filter of misclassified examples
        NWronglyClassified = Filter.sum()     ▷ False=0, True=1
        if NWronglyClassified == 0 then return w   ▷ If no mistake, then exit
        w = w + (η/N) X[Filter].T[Filter]
```

Algorithm 2 Full Batch Perceptron, **C++** style

```
1: procedure FullBatchLearning(X, y, η, w_0, maxIter)
2:     w = w_0
3:     for iter in [1,...,MaxIter] do
4:         MissClassifList = [ ]                                  ▷ create empty list
5:         for n in [1,...,N] do
6:             if X[n] · w[n] ≠ T[n] then                         ▷ scalar product
7:                 MissClassifList.append(n)                      ▷ identify wrongly-classified examples
8:         if MissClassifList== empty list then return w          ▷ If no mistake, then exit
9:         for n in MissClassifList do                            ▷ Update weights
10:            w = w + η.X[n].T[n]
```

4. Ok, now, you may start TP2.1-Perceptron.ipynb

5. What is the time complexity of this algorithm ? And space complexity ?

CORRECTION Time complexity. Clearly $MaxIter$ comes as a prefactor. Tehn each iteration needs to compute N dot products and comparisons, $(\vec{X}[n] \cdot \vec{w}[n] \neq T[n])$, so that's $\sim O(ND)$ operations The exit criterion evaluation is negligible. Then, the update involves a sum of $N_{WronglyClassified}$ terms, each of dimension D, so that's at worst $O(ND)$ operations, although typically a bit less, by a factor of order 10-100, since the error will rarely be much less than 1%, but also, hopefully, will be less than 10%. In total, the number of operations is of the order $O(IterMax.N.D)$.

The space (memory) usage is small: we just need to store the data X and T, of size ND and N, so that's $O(ND)$. The parameters are of size D, and the list or filter are of size N at worst. So, in total, we have $ND + N + D + N$, which scales like $O(ND)$.

2.1.2 Roseblatt's online Perceptron

1. Write the pseudo code of the Online perceptron, i.e. the same as ex. 2.1.1 but using the *online* strategy, i.e. taking examples 1 by 1, in their (arbitrary) ordering in the data set.

Algorithm 3 Online Perceptron

```
1: procedure OnlineLearning(X, y, η, w_0, maxIter)
2:     w = w_0
3:     for iter in [1,...,MaxIter] do
4:         NumberOfMissclassified = 0
5:         for n in [1,...,N] do
6:             if X[n] · w[n] ≠ y[n] then                         ▷ scalar product (dot product)
7:                 NumberOfMissclassified += 1
8:                 w = w + η.(1/N) X[n].y[n]                      ▷ immediate weight update
9:         if NumberOfMissclassified ==0 then return w            ▷ If no mistake, then exit
```

2. In your opinion, when we do this kind of online learning, is it in general better to have first all examples of class 1, then all examples of class 2, or rather, to mix them randomly ?

CORRECTION It's better to mix, since otherwise we'll move the hyperplane only to classify class 1 correctly, and then when visiting examples of class 2, we'll only be correcting for these examples. This may lead to a kind of oscillation in the hyperplane position. The result will be more stable if we mix at random. Ideally, changing the order of visit of the examples is better. That strategy is called SGD (Stochastic Gradient Descent).

3. Ignoring this last comment, let's apply the online perceptron to the following data set:

$$\text{Data set 1}: \quad X = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 4 \\ 0 \end{pmatrix}, \begin{pmatrix} 4 \\ 1 \end{pmatrix} \right\}$$
$$T = \{-1, -1, 1, 1, 1, 1\}$$

a) Plot this data on your piece of paper (or notice the similarity with TP2.1 and cheat..)

b) Using the Online Perceptron algorithm (and updating only when an example is wrongly classified), perform a full epoch, *using only pen and paper! (like at the exam!)*. You may start from $\vec{w}_0 = (1, 0, 0)$, using $\eta = 1$. You could also quickly (using a picture, not detailing all computations), do all epochs until convergence. Note: When an example is exactly on the hyperplane, i.e. when $f_\Theta(\vec{x}) = 0$, consider it is not well classified. Help: at the end, you should find[1].

[1] You should find, after one epoch, (1 2 0). After a second epoch, we have $\vec{w} = (0\ 3\ 0)$, which, after a single correction, becomes correct, $\vec{w} = (-1\ 3\ 0)$ The vector $\vec{w} = (-1\ 3\ 0)$, yields no error, i.e. we have converged.

.

CORRECTION We recall the update: $\vec{w} \mapsto \vec{w} + \eta \vec{x}_n t_n$. *The intermediate* $\vec{w}$ *vectors are, in order :* $\vec{w} = (1\ 0\ 0), \vec{w} = (0\ 0\ 0), \vec{w} = (-1\ 0\ -1), \vec{w} = (0\ 1\ -1), \vec{w} = (1\ 2\ 0)$*, which closes the 1st epoch (N=6 checks for correctness and 4 updates.). Then,* $\vec{w} = (0\ 2\ 0), \vec{w} = (-1\ 2\ -1), \vec{w} = (0\ 3\ 0)$*, which cponcludes teh second epoch (N=6 checks, 3 updates), and then* $\vec{w} = (0\ 3\ 0)$ *after 1 check-update, and* $\vec{w} = (-1\ 3\ 0)$ *is correct for each data point.*

4. Check your computations using code.

2.1.3 Logistic Regression

Same questions as in ex. 2.1.1, but using the logistic function as activation function, i.e. using $\sigma(z) = \frac{1}{1+e^{-z}}$. Concretely, this means the output of the network is $\hat{y} = \sigma(f_\Theta(X))$.

Help: First, you should derive separately, in the general case, what is the derivative of $\sigma(u(\theta))$ with respect to θ, for a function u that is supposed to be sufficiently regular. More help: the result you should find is – see the footnote for a spoiler[2] (it's more honest to not look at the spoilers).

2.1.4 SGD

In ex. 2.1.1, change your learning strategy for the SGD strategy.

2.2 Multi class classification

This is a hard exercise. See TP2.2-MultiClass-Classification.ipynb for a much more detailed, step-by-step approach.

Same questions as in ex. 2.1.1, but using the cross-entropy Loss $\mathcal{L}_{CE}$, and recalling that the model is $\vec{y}$:

$$\vec{y}_n = softmax(W\vec{x}_n) \tag{5}$$

$$\mathcal{L}_{CE} = -\frac{1}{N}\sum_n^N \vec{t}_n \log(\vec{y}_n) = -\frac{1}{N}\sum_n^N \sum_k^K t_{n,k} \log(y_{n,k}) \tag{6}$$

Help: take some time to write down the shapes of all things.

2.3 Minimum distance classifier

This is in the spirit of "being able to write the code of an algorithm when it's described to you". This case is especially easy.

A minimum distance classifier works by defining a so-called *representative point* for each class of the data to classify. The representative of a class is chosen to be the barycenter (the average) of the points of this class (on the learning data). This point is supposed to "represent" the whole class (in a simplified way). For a given point, the prediction of its class corresponds to choosing the representative that is the most close to the point. So it is a *one shot* learning, it is done "at once".

In this exercise, we limit ourselves to two classes in a vector space of dimension $D = 2$, equipped with the scalar product $\langle x, x' \rangle$ (and thus the Euclidean distance).

1. Explain the logic of the algorithm: what are its parameters, how many are they, how do we train them? What are the hyper-parameters?

2. Explain the logic of the algorithm: how does one produce the prediction y for the class of a new data x_{test} that has not been observed during the training?

3. Write the pseudo code of the algorithm. We will specify the size of the vectors and matrices used, indicating the meaning of each variable. We will try to choose meaningful names for indices (for example $d = 1, 2, \ldots, D$).

4. You may fit the model on these 4 data sets (at home):

$$\text{Data set 1 :} \quad X_{-1} = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}$$

$$X_{+1} = \left\{ \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \begin{pmatrix} 3 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \end{pmatrix} \right\}$$

$$\text{Data set 2 :} \quad X_{-1} = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$$

$$X_{+1} = \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \begin{pmatrix} 3 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \end{pmatrix}, \begin{pmatrix} 4 \\ 0 \end{pmatrix}, \begin{pmatrix} 4 \\ 1 \end{pmatrix} \right\}$$

[2] $\partial_\theta \sigma\big(u(\theta)\big) = \sigma\big(u(\theta)\big)\Big(1 - \sigma\big(u(\theta)\big)\Big)\partial_\theta u$. More explictly, calling $y = \sigma\big(u(\theta)\big)$, we have the elegant formula: $\frac{\partial}{\partial\theta} y = y(1-y)\frac{\partial}{\partial\theta} u$

For each of these datasets:

4.a Compute a representative of each class

4.b Calculate the decision surface equation and derive the decision rule from it

4.c Tell if learning points are misclassified.

Help: in a sense, this algorithm is the supervised version of the classic K-means algorithm.

2.4 2020's Exam – O-V-O classification

In this problem, some parts are inter-dependent. However, to some extent, if you cannot do one of the parts, you can still do some of the later parts. If you don't find this easy, I would advise to keep this problem for the end.

We want to perform multi-class classification (K classes) on $(D-1)$-dimensional data. The extended data points (with a 1 as the value of the first component) will then be D-dimensional, which is very convenient. We have N examples in the training set. The data is the set of the training examples $X = \{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N, k_{true}^{(1)}, k_{true}^{(2)}, \ldots, k_{true}^{(N)}\}$, where $k_{true}^{(n)}$ is the label of example n (i.e. it takes values in $\{1, 2, \ldots, K\}$).

There are K classes. We encode the ground truth as one-hot vectors $\vec{t_n}$ of size K, or concretely $t_n = (0, ..., 0, 1, 0, ..., 0)^T$. This means that for the components t_{nk} we have $t_{nk} = \delta_{k,k_{true}^{(n)}}$, that is to say, we have $t_{nk} = \begin{cases} 1 \text{ if } k = k_{true}^{(n)} \\ 0 \text{ if } k \neq k_{true}^{(n)} \end{cases}$.

We plan to use a **one-vs-one classification scheme** (o-v-o). That means that for each pair of classes (k, k'), we will have a (binary) classifier $\vec{w}_{kk'}$. It will tell us if we are more likely to be in class k or class k'. For now, don't worry too much about the o-v-o scheme.

1. We start with the simpler $K = 2$ case, but keeping the one-hot encoding, so for class 1, $t_n = (1, 0)^T$, and for class 2, $t_n = (0, 1)^T$. The hyperplane characterized by the (extended) vector $\vec{w}_{12}$ points towards class 1: when $\vec{x}$ is on the "class 1" side of the hyperplane, then we have $\vec{w}_{12} \cdot \vec{x} > 0$. The loss function we consider has a term of the form

$$\sigma(\vec{w}_{12} \cdot \vec{x}_n) - (t_{n1} - t_{n2}) \tag{7}$$

   where $\sigma()$ is some non-linear activation function.

   (a) In the term above, look at what happens when example n is of class 1? And then what happens when example n is of class 2? Show that we can define a target label $t_{n,12}$ (to be read 1, 2, one-two, not twelve!) such that this expression looks more like what we're used to. (0.5 pt)

   (b) Explain how $\vec{w}_{21}$ depends on $\vec{w}_{12}$. How does $t_{n,21}$ depend on $t_{n,12}$? (0.25 pt)

   (c) Generalize the term of equation 1 to $K > 2$ classes. In particular, define our target label $t_{n,kk'}$. (0.75 pt)

2. In the $K > 2$ case, how many parameters are there in the model (what is the cardinal of Θ)? You may count "naively" and then count only independent parameters (making use of the relationship(s) found earlier). In the rest of the exercise, we may ignore this dependence and encode the parameters in a rather simple matrix, of simple shape. (0.5 pt)

3. We choose a Least-Squared Error kind of loss function, i.e. we want to define the loss J (or $\mathcal{L}$, as you prefer) as the appropriate sum over examples and classes, of the square of the term found in the previous question (generalization of equation 1 to $K > 2$ classes). Write down explicitly the full cost function of our model, $J(\Theta, X)$. Be careful to not forget some sums ($\sum_{...}$)! (0.5 pt)

4. (This question depends on question 3) The hyperbolic tangent function is defined by: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. It goes from -1 at $x \sim -\infty$ to $+1$ at $x \sim +\infty$. Its derivative is (for a generic smooth function u): $\partial_x \tanh(u(x)) = \frac{4u'(x)}{(\cosh(u(x)))^2}$, where $\cosh(x)$ is the hyperbolic cosine, $\cosh(x) = \frac{e^x + e^{-x}}{2}$. We know that $\cosh(x) \geq 1, \forall x$. We decide to use $\sigma(.) = \tanh(.)$, since $\tanh(.)$ nicely interpolates between -1 and $+1$.

   (a) Derive the elementary step of the Gradient descent algorithm for this $J(\Theta, X)$. If you cannot compute it, or you aren't sure of the result, in the rest of the exercise, you may simply write ∇J when it's convenient (and not explicit what it is). Explicit all indices. (1 pt)

   (b) Write the GD step in Matrix form, and give the shapes of all matrices or vectors involved, to make sure your computation is a legal computation. (0.5 pt)

5. We are interested in what will happen if we do updates taking points one by one (like in SGD or Online Perceptron). In particular, what happens to plane $\vec{w}_{kk'}$ when we visit a point of class k'', with $k'' \neq k$, and $k'' \neq k'$? (0.5 pt)

* We can now suppose the model has been learned. You may define some $f_\Theta(\vec{x})$ function, to lighten your notations. From now on, we assume the parameters have been learned.

6. For a test point $\vec{x}^{test}$ (for simplicity, $\vec{x}^{test} = \vec{x}$), what are the predictions of the hyperplanes $w_{kk'}$? How many do we have? We decide to take the max value. Write the corresponding `predict` function formally (mathematically). (0.75 point)

7. Choose a performance metric for this task, and write it in mathematical form for our model. (0.25 point)

8. Specify the list of our hyper-parameters (the main ones at least). (0.5 point)

Bonus Write down the `fit` and `predict` function in pseudo-code, sklearn-style. You should assume the data is given, but you should initialize the parameters and hyper-params yourself (as in real code). (up to 1 point? I don't know)