Session 2

In this second session we will apply the Perceptron algorithm to some classification tasks. A simple implementation of the Perceptron algorithm and its application is provided. The final purpose of this session is to apply the Perceptron algorithm to MyDigits dataset to train a matrix of weights that will be used in an application for Handwritten Digit Classification.

You may need to run this code if this is the first time you are running this notebook.

In [ ]:
```
!pip install seaborn scikit-learn pandas pillow gradio matplotlib
```

Perceptron

In [8]:
```
#import warnings; warnings.filterwarnings("ignore")
import numpy as np
```

Perceptron Classification: classification of samples provided a weight matrix. Samples need to be prefixed by 1

$$c(\boldsymbol{x}) = \underset{c}{\operatorname{argmax}}\, g_c(\boldsymbol{x}), \text{ with } g_c(\boldsymbol{x}) = \boldsymbol{w}_c^t \boldsymbol{x} \text{ for all } c$$

where $\boldsymbol{x} = (1, x_1, \ldots, x_D)^t$, $\mathbf{W} = (\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_C)$ and $\boldsymbol{w}_c = (w_{c0}, w_{c1}, \ldots, w_{cD})^t$

In [9]:
```
def PerceptronClassification(X, W):
  Xh = np.hstack([np.ones((len(X), 1)), X])
  return np.argmax(Xh @ W, axis=1).reshape(-1, 1)
```

PerceptronTraining: Perceptron learns a matrix of weights $\mathbf{W}^*$ that minimizes the number of training errors (with margin b)

$$\mathbf{W}^* = \underset{\mathbf{W}=(\boldsymbol{w}_1,\ldots,\boldsymbol{w}_C)}{\operatorname{argmin}} \sum_n \mathbb{I}\left(\max_{c\neq y_n} \boldsymbol{w}_c^t \boldsymbol{x}_n + b \;>\; \boldsymbol{w}_{y_n}^t \boldsymbol{x}_n\right)$$

It returns weights in homogeneous notation, $\mathbf{W} \in \mathbb{R}^{(1+D)\times C}$; together with the number of errors and iterations executed

> **Input:** data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n)\}$ weights $\mathbf{W} = \{\boldsymbol{w}_c\}$ learning rate $\alpha \in \mathbb{R}^{>0}$ margin $b \in \mathbb{R}^{\geq 0}$
> **Output:** optimized weights $\mathbf{W}^* = \{\boldsymbol{w}_c\}^*$
> `repeat`
>
> > `for all` training sample $\boldsymbol{x}_n$
> >
> > > *err* = `False`
> > > `for all` class $c \neq y_n$
> > >
> > > > `if` $\boldsymbol{w}_c^t \boldsymbol{x}_n + b > \boldsymbol{w}_{y_n}^t \boldsymbol{x}_n$: $\boldsymbol{w}_c = \boldsymbol{w}_c - \alpha \boldsymbol{x}_n$; *err* = `True`
> >
> > `if` *err*: $\boldsymbol{w}_{y_n} = \boldsymbol{w}_{y_n} + \alpha \boldsymbol{x}_n$
>
> `until` no training sample is misclassified

Input: data $\mathcal{D} = \{(\boldsymbol{x}_n, y_n)\}$ weights $\mathbf{W} = \{\boldsymbol{w}_c\}$ learning rate $\alpha \in \mathbb{R}^{>0}$ margin $b \in \mathbb{R}^{\geq 0}$

Output: optimized weights $\mathbf{W}^* = \{\boldsymbol{w}_c\}^*$

`repeat`

> `for all` training sample $\boldsymbol{x}_n$
>
> > *err* = `False`
> > `for all` class $c \neq y_n$
> >
> > > `if` $\boldsymbol{w}_c^t \boldsymbol{x}_n + b > \boldsymbol{w}_{y_n}^t \boldsymbol{x}_n$: $\boldsymbol{w}_c = \boldsymbol{w}_c - \alpha \boldsymbol{x}_n$; *err* = `True`
> >
> > `if` *err*: $\boldsymbol{w}_{y_n} = \boldsymbol{w}_{y_n} + \alpha \boldsymbol{x}_n$

`until` no training sample is misclassified

In [29]:

```python
def PerceptronTraining(X, y, b=0.1, a=1.0, K=200):
    N, D = X.shape; Y = np.unique(y); C = Y.size; W = np.zeros((1+D, C))
    for k in range(1, K+1):  # for K iterations
        E = 0
        for n in range(N):  # for every training sample
            xn = np.array([1, *X[n, :]])
            cn = np.squeeze(np.where(Y==y[n]))  # Mapping to class labels from 0 to C-1 (for algorithmic simplicity)
            gn = W[:,cn].T @ xn; err = False
            for c in np.arange(C):  # for every class
                if c != cn and W[:,c].T @ xn + b >= gn:
                    W[:, c] = W[:, c] - a*xn; err = True
            if err:
                W[:, cn] = W[:, cn] + a*xn; E = E + 1
        if E == 0:
            break
    return W, E, k
```

Perceptron applied to the Iris dataset

Reading the dataset: we also check that the data matrix and labels have the right number of rows and columns

In [11]:
```
from sklearn.datasets import load_iris
iris = load_iris(); X = iris.data.astype(np.float16)
y = iris.target.astype(np.uint).reshape(-1, 1)
print(X.shape, y.shape, "\n", np.hstack([X, y])[:5, :])
```

```
(150, 4) (150, 1)
 [[5.1015625  3.5        1.40039062 0.19995117 0.        ]
 [4.8984375  3.         1.40039062 0.19995117 0.        ]
 [4.69921875 3.19921875 1.29980469 0.19995117 0.        ]
 [4.6015625  3.09960938 1.5        0.19995117 0.        ]
 [5.         3.59960938 1.40039062 0.19995117 0.        ]]
```

Dataset partition: We create a split of the Iris dataset with 20% of data for test and the rest for training, previously shuffling the data according to a given seed provided by a random number generator. Here, as in all code that includes randomness (which requires generating random numbers), it is convenient to fix said seed to be able to reproduce experiments with accuracy.

In [12]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=True, random_state=23)
print(X_train.shape, X_test.shape)
```

```
(120, 4) (30, 4)
```

Learning a (linear) classifier with Perceptron:

In [13]:
```
W, E, k = PerceptronTraining(X_train, y_train)
print("Number of iterations executed: ", k)
print("Number of training errors: ", E)
print("Weight vectors of the classes (in columns and with homogeneous notation):\n", W)
```

```
Number of iterations executed:  200
Number of training errors:  2
Weight vectors of the classes (in columns and with homogeneous notation):
 [[  10.           85.         -142.        ]
 [ -49.421875    -68.19140625 -176.47265625]
 [  50.171875     -1.72460938 -181.06445312]
 [-189.91210938  -87.70507812   68.69726562]
 [ -86.40258789 -137.78149414  157.88415527]]
```

Calculation of test error rate:

In [14]:
```
y_test_pred = PerceptronClassification(X_test,W)
err_test = np.count_nonzero(y_test_pred != y_test) / len(X_test)
print(f"Error rate on test: {err_test:.1%}")
```

```
Error rate on test: 16.7%
```

Adjusting maximum number of iterations:

In [15]:
```
print(f"  alpha       b      K TrErr  TeErr")
print(f"------- ------- ------ ------ ------")
b = 0.1; a = 1.0
for K in (1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000):
    W, E, k = PerceptronTraining(X_train, y_train, b=b, a=a, K=K)
    y_test_pred = PerceptronClassification(X_test,W)
    err_test = np.count_nonzero(y_test_pred != y_test) / len(X_test)
    print(f"{a:.1e} {b:.1e} {k:6d} {E/len(X_train):6.1%} {err_test:6.1%}")
```

```
  alpha       b      K TrErr  TeErr
------- ------- ------ ------ ------
1.0e+00 1.0e-01      1  49.2%  33.3%
1.0e+00 1.0e-01      2  31.7%  50.0%
1.0e+00 1.0e-01      5  14.2%  73.3%
1.0e+00 1.0e-01     10  12.5%  56.7%
1.0e+00 1.0e-01     20  14.2%  26.7%
1.0e+00 1.0e-01     50   8.3%  16.7%
1.0e+00 1.0e-01    100   9.2%  26.7%
1.0e+00 1.0e-01    200   1.7%  16.7%
1.0e+00 1.0e-01    500   2.5%   3.3%
1.0e+00 1.0e-01   1000   2.5%  13.3%
1.0e+00 1.0e-01   2000   5.0%   3.3%
1.0e+00 1.0e-01   5000   1.7%   6.7%
```

Adjusting the learning rate (alpha):

In [16]:
```
print(f"  alpha       b      K TrErr  TeErr")
print(f"------- ------- ------ ------ ------")
b = 0.1; K = 500
for a in (1e-3, 1e-2, 1e-1, 1e-0, 1e1, 1e2, 1e3):
    W, E, k = PerceptronTraining(X_train, y_train, b=b, a=a, K=K)
    y_test_pred = PerceptronClassification(X_test,W)
    err_test = np.count_nonzero(y_test_pred != y_test) / len(X_test)
    print(f"{a:.1e} {b:.1e} {k:6d} {E/len(X_train):6.1%} {err_test:6.1%}")
```

```
  alpha       b      K TrErr  TeErr
------- ------- ------ ------ ------
1.0e-03 1.0e-01    500   8.3%   3.3%
1.0e-02 1.0e-01    500   2.5%   3.3%
1.0e-01 1.0e-01    500   4.2%  16.7%
1.0e+00 1.0e-01    500   2.5%   3.3%
1.0e+01 1.0e-01    500   4.2%  16.7%
1.0e+02 1.0e-01    500   4.2%  16.7%
1.0e+03 1.0e-01    500   0.8%  16.7%
```

Adjusting the margin (b):

In [17]:
```
print(f"  alpha       b      K TrErr  TeErr")
print(f"------- ------- ------ ------ ------")
a = 1.0; K = 500
for b in (.0, .01, .1, 1, 10, 100):
    W, E, k = PerceptronTraining(X_train, y_train, b=b, a=a, K=K)
    y_test_pred = PerceptronClassification(X_test,W)
    err_test = np.count_nonzero(y_test_pred != y_test) / len(X_test)
    print(f"{a:.1e} {b:.1e} {k:6d} {E/len(X_train):6.1%} {err_test:6.1%}")
```

```
  alpha       b      K TrErr  TeErr
------- ------- ------ ------ ------
1.0e+00 0.0e+00    500   0.8%  16.7%
1.0e+00 1.0e-02    500   4.2%  16.7%
1.0e+00 1.0e-01    500   2.5%   3.3%
1.0e+00 1.0e+00    500   4.2%  16.7%
1.0e+00 1.0e+01    500   2.5%   3.3%
1.0e+00 1.0e+02    500   8.3%   3.3%
```

Interpretation of results: the training data does not appear to be linearly separable; it is not clear that a margin greater than zero can improve results, especially since we only have 30 test samples; with a margin $b = 0.1$ we have already seen that an error (in test) of 16.7% is obtained

Perceptron applied to the Digits dataset

Reading the dataset: we also check that the data matrix and labels have the right number of rows and columns

In [18]:
```
from sklearn.datasets import load_digits
digits = load_digits(); X = digits.images.astype(np.float16).reshape(-1, 8*8); X/=np.max(X)
y = digits.target.astype(np.uint).reshape(-1, 1)
print(X.shape, y.shape, "\n", np.hstack([X, y])[:4, :])
```

```
(1797, 64) (1797, 1)
 [[0.     0.     0.3125 0.8125 0.5625 0.0625 0.     0.     0.     0.
  0.8125 0.9375 0.625  0.9375 0.3125 0.     0.     0.1875 0.9375 0.125
  0.     0.6875 0.5    0.     0.     0.25   0.75   0.     0.     0.5
  0.5    0.     0.     0.3125 0.5    0.     0.     0.5625 0.5    0.
  0.     0.25   0.6875 0.     0.0625 0.75   0.4375 0.     0.     0.125
  0.875  0.3125 0.625  0.75   0.     0.     0.     0.     0.375  0.8125
  0.625  0.     0.     0.     0.    ]
 [0.     0.     0.     0.75   0.8125 0.3125 0.     0.     0.     0.
  0.     0.6875 1.     0.5625 0.     0.     0.     0.     0.1875 0.9375
  1.     0.375  0.     0.     0.     0.4375 0.9375 1.     1.     0.125
  0.     0.     0.     0.     0.0625 1.     1.     0.1875 0.     0.
  0.     0.     0.0625 1.     1.     0.375  0.     0.     0.     0.
  0.0625 1.     1.     0.375  0.     0.     0.     0.     0.     0.6875
  1.     0.625  0.     0.     1.    ]
 [0.     0.     0.     0.25   0.9375 0.75   0.     0.     0.     0.
  0.1875 1.     0.9375 0.875  0.     0.     0.     0.     0.5    0.8125
  0.5    1.     0.     0.     0.     0.     0.0625 0.375  0.9375 0.6875
  0.     0.     0.     0.0625 0.5    0.8125 0.9375 0.0625 0.     0.
  0.     0.5625 1.     1.     0.3125 0.     0.     0.     0.     0.1875
  0.8125 1.     1.     0.6875 0.3125 0.     0.     0.     0.     0.1875
  0.6875 1.     0.5625 0.     2.    ]
 [0.     0.     0.4375 0.9375 0.8125 0.0625 0.     0.     0.     0.5
  0.8125 0.375  0.9375 0.25   0.     0.     0.     0.125  0.0625 0.8125
  0.8125 0.     0.     0.     0.     0.     0.125  0.9375 0.6875 0.0625
  0.     0.     0.     0.     0.     0.0625 0.75   0.75   0.0625 0.
  0.     0.     0.     0.     0.0625 0.625  0.5    0.     0.     0.
  0.5    0.25   0.3125 0.875  0.5625 0.     0.     0.     0.4375 0.8125
  0.8125 0.5625 0.     0.     3.    ]]
```

Dataset partition: We create a split of the Iris dataset with 20% of data for test and the rest for training, previously shuffling the data according to a given seed provided by a random number generator. Here, as in all code that includes randomness (which requires generating random numbers), it is convenient to fix said seed to be able to reproduce experiments with accuracy.

In [19]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=True, random_state=23)
print(X_train.shape, X_test.shape)
```

```
(1437, 64) (360, 64)
```

Adjusting maximum number of iterations:

In [20]:
```
print(f"  alpha       b      K TrErr  TeErr")
print(f"------- ------- ------ ------ ------")
b = 0.1; a = 1.0
for K in (1, 2, 5, 10, 20, 50, 100, 200):
    W, E, k = PerceptronTraining(X_train, y_train, b=b, a=a, K=K)
    y_test_pred = PerceptronClassification(X_test,W)
    err_test = np.count_nonzero(y_test_pred != y_test) / len(X_test)
    print(f"{a:.1e} {b:.1e} {k:6d} {E/len(X_train):6.1%} {err_test:6.1%}")
```

```
  alpha       b      K TrErr  TeErr
------- ------- ------ ------ ------
1.0e+00 1.0e-01      1  25.1%  14.7%
1.0e+00 1.0e-01      2  13.2%   8.1%
1.0e+00 1.0e-01      5   8.4%   8.1%
1.0e+00 1.0e-01     10   5.6%   7.5%
1.0e+00 1.0e-01     20   2.9%   6.7%
1.0e+00 1.0e-01     50   1.9%   5.8%
1.0e+00 1.0e-01    100   0.8%   4.7%
1.0e+00 1.0e-01    111   0.0%   4.4%
```

Adjusting the learning rate (alpha):

In [21]:
```
print(f"  alpha       b      K TrErr  TeErr")
print(f"------- ------- ------ ------ ------")
b = 0.1; K = 1000
for a in (1e-3, 1e-2, 1e-1, 1e-0, 1e1, 1e2, 1e3):
    W, E, k = PerceptronTraining(X_train, y_train, b=b, a=a, K=K)
    y_test_pred = PerceptronClassification(X_test,W)
    err_test = np.count_nonzero(y_test_pred != y_test) / len(X_test)
    print(f"{a:.1e} {b:.1e} {k:6d} {E/len(X_train):6.1%} {err_test:6.1%}")
```

```
  alpha       b      K TrErr  TeErr
------- ------- ------ ------ ------
1.0e-03 1.0e-01    742   0.0%   4.2%
1.0e-02 1.0e-01    188   0.0%   5.6%
1.0e-01 1.0e-01    113   0.0%   5.3%
1.0e+00 1.0e-01    111   0.0%   4.4%
1.0e+01 1.0e-01    130   0.0%   3.6%
1.0e+02 1.0e-01    112   0.0%   4.2%
1.0e+03 1.0e-01    112   0.0%   4.2%
```

Adjusting the margin (b):

In [22]:
```
print(f"  alpha       b      K TrErr  TeErr")
print(f"------- ------- ------ ------ ------")
a = 1e1; K = 1000
for b in (.0, .01, .1, 1, 10, 100):
    W, E, k = PerceptronTraining(X_train, y_train, b=b, a=a, K=K)
    y_test_pred = PerceptronClassification(X_test,W)
    err_test = np.count_nonzero(y_test_pred != y_test) / len(X_test)
    print(f"{a:.1e} {b:.1e} {k:6d} {E/len(X_train):6.1%} {err_test:6.1%}")
```

```
  alpha       b      K TrErr  TeErr
------- ------- ------ ------ ------
1.0e+01 0.0e+00    112   0.0%   4.2%
1.0e+01 1.0e-02    112   0.0%   4.2%
1.0e+01 1.0e-01    130   0.0%   3.6%
1.0e+01 1.0e+00    111   0.0%   4.4%
1.0e+01 1.0e+01    113   0.0%   5.3%
1.0e+01 1.0e+02    187   0.0%   5.0%
```

Interpretation of results: the training data is linearly separable with training error equal to zero. In this case, it seems that small margins provide similar results on the test set with a lowest value of 3.6%.

Perceptron applied to MyDigits dataset

Reading the dataset:

In [ ]:
```
# Execute this cell only when running in Google Colab
# You need to upload your images and labels files
from google.colab import files
uploaded = files.upload()
```

In [ ]:
```
from sklearn.model_selection import train_test_split

with open('images.npy', 'rb') as fd:
    X = np.load(fd)
    fd.close()

with open('labels.npy', 'rb') as fd:
    y = np.load(fd).astype(int)
    fd.close()
```

Dataset partition:

In [ ]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=True, random_state=23)
print(X_train.shape, X_test.shape)
```

Adjusting maximum number of iterations:

In [ ]:
```
print(f"  alpha       b      K TrErr  TeErr")
print(f"------- ------- ------ ------ ------")
# Complete
```

Adjusting the learning rate (alpha):

In [ ]:
```
print(f"  alpha       b      K TrErr  TeErr")
print(f"------- ------- ------ ------ ------")
# Complete
```

Adjusting the margin (b):

In [ ]:
```
print(f"  alpha       b      K TrErr  TeErr")
print(f"------- ------- ------ ------ ------")
# Complete
```

Final classifier: Training final classifier with best parameters, saving and loading to test it

In [ ]:
```
b = 0.1; a = 1.0; K = 200 # Replace with the best configuration obtained in the previous experiments
W, E, k = PerceptronTraining(X, y, b=b, a=a, K=K)
np.save("MyDigitsWeights.npy",W)
```

In [ ]:
```
with open('MyDigitsWeights.npy', 'rb') as fd:
    W = np.load(fd)
    fd.close()
y_test_pred = PerceptronClassification(X_test,W)
err_test = np.count_nonzero(y_test_pred.flatten() != y_test) / len(X_test)
print(f"Test error of final classifier: {err_test:.1%}")
```

In [ ]:
```
# Execute this cell only when running in Google Colab
# You need to download MyDigitsWeights.npy
files.download('MyDigitsWeights.npy')
```

Classify your own handwritten digits

The following simple application allows you to classify your own handwritten digits. When you run this application, it shows a basic graphical interface containing a panel on which you can draw your own handwritten digits.

Before you can draw a digit, you need to click on the *pen* locate on the left vertical. Then you can draw on the panel. If you need to erase what you have drawn on the panel, just click on *bin* located on the top menu.

You can classify the image on the panel by clicking on the bottom bar labeled with *Classify image".

In [ ]:
```
# Execute this cell only when running in Google Colab
# You need to upload DigitClassifyGradioApp.py
from google.colab import files
uploaded = files.upload()
```

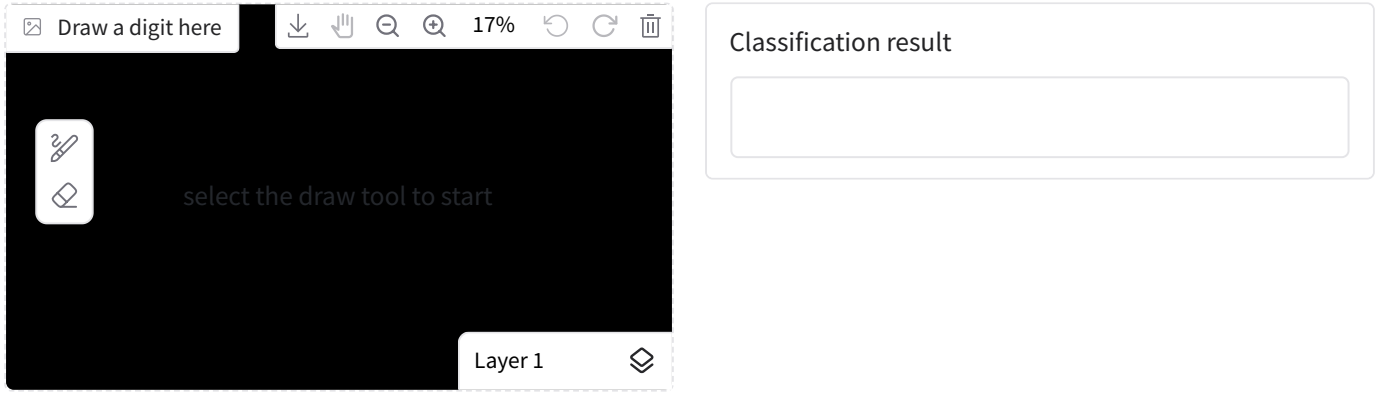
In [26]:
```
from DigitClassifyGradioApp import create_interface

fn = input("Please provide filename for weight matrix:")
with open(fn, 'rb') as fd:
    W = np.load(fd)
    fd.close()

demo = create_interface(W, PerceptronClassification)
demo.launch()
```

```
* Running on local URL:  http://127.0.0.1:7863
* To create a public link, set `share=True` in `launch()`.
```

Handwritten Digit Classification



Classify image

Utilitzar via API · Construït amb Gradio · Configuració

Out[26]: