Lab 7: Classification and the Perceptron

Tutorial objectives:

- **Revise and solidify your understanding of the Perceptron learning rule (in matrix form)**
- **Implement the Perceptron learning rule using Numpy library**
- **Practice working with Sklearn's API by porting your implementation of the learning rule into Sklearn framework**
- **Reinforce your understanding of the geometric representation of the Perceptron through visualisations of its decision function**
- **Train the Perceptron on various datasets**

Classification concepts

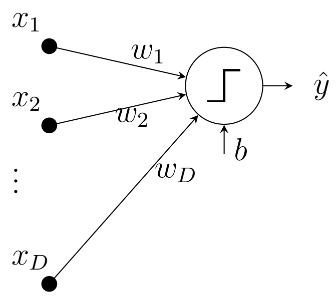
1. What's a half-space?
2. How would 4-class labels be encoded in one-hot encoding?
3. How is classification accuracy computed?

Perceptron

Let's go over the matrix-based calculation for the Perceptron learning rule, starting with a single-neuron perceptron connected to input of D attributes. The output of the neuron (including the activation function) is defined as:

$$\hat{y} = \begin{cases} 1 & \sum_{i=1}^{D} w_i x_i + b > 0 \\ 0 & \text{otherwise,} \end{cases}$$

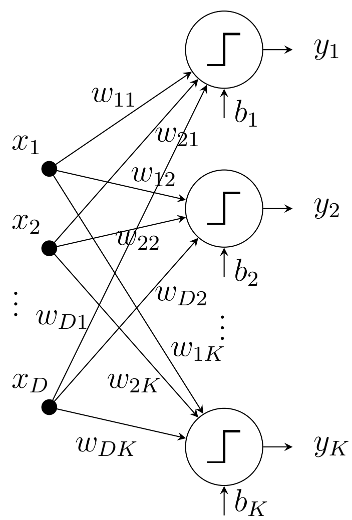
where x_i and w_i are the i^{th} input and weight respectively, b is the bias. We have one neuron, D inputs, D weights, single bias, and a single output.



We extend this to multiple neurons, each connected to D inputs. The output of the k^{th} neuron is:

$$\hat{y}_k = \begin{cases} 1 & \sum_{i=1}^{D} w_{ik}x_i + b_k > 0 \\ 0 & \text{otherwise,} \end{cases}$$

where w_{ik} is the weight on the connection between input i and neuron k, b_k is the bias of neuron k. We've got K neurons, D inputs, K times D weights, and K biases.



A dataset for a K-class classification problem consists of a set of N training points $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_N)$. An input point $\mathbf{x}_n - \begin{bmatrix} x_{n1} & \ldots y_{nD} \end{bmatrix}$ consists of D percepts/attributes and $\mathbf{y}_n$ provides the label information in the format $\mathbf{y}_n = \begin{bmatrix} y_{n1} & \ldots y_{nK} \end{bmatrix}$, where label corresponding to class k consists of all 0's expect for $y_{nk} = 1$ (so-called one-hot encoding). In other words, we have a machine learning problem with training data as in the table below:

n	$\mathbf{x}_n$				$\mathbf{y}_n$			
	x_{n1}	x_{n2}	$\ldots$	x_{nD}	y_{n1}	y_{n2}	$\ldots$	y_{nK}
1	.	.	.	.	.	.	.	.
2	.	.	.	.	.	.	.	.
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
N	.	.	.	.	.	.	.	.

A sample n of the input consists of D attributes. We can denote the output of the k^{th} neuron for the n^{th} input sample as

$$\hat{y}_{nk} = \begin{cases} 1 & \sum_{i=1}^{D} w_{ik} x_{ni} + b_k > 0 \\ 0 & \text{otherwise,} \end{cases}$$

where x_{ni} is the i^{th} attribute of input sample n.

Why make this all so complicated with all of this indexing? Well, consider the following matrix multiplication:

$$\begin{bmatrix} x_{11} & \dots & x_{1D} \\ \vdots & \ddots & \vdots \\ x_{N1} & \dots & x_{ND} \end{bmatrix} \begin{bmatrix} w_{11} & \dots & w_{1K} \\ \vdots & \ddots & \vdots \\ w_{D1} & \dots & w_{DK} \end{bmatrix} = \begin{bmatrix} \sum_{d=1}^{D} w_{d1} x_{1d} & \dots & \sum_{d=1}^{D} w_{dK} x_{1d} \\ \vdots & \ddots & \vdots \\ \sum_{d=1}^{D} w_{d1} x_{1N} & \dots & \sum_{d=1}^{D} w_{dK} x_{Nd}. \end{bmatrix}$$

Note that matrix multiplication of a $N \times D$ input matrix (where rows correspond to different training points and columns different attribute of a given point) times a $D \times K$ weight matrix (where each column is a set of D weights of a neuron) gives an $N \times K$ matrix with entry at row n and column k being equal to $\sum_{i=1}^{D} w_{ik} x_{ni}$. Once we add appropriate bias to each column of the $N \times K$ matrix, we have the activity of K neurons in response to N inputs. Then, all we need to do is to turn every positive activity to 1, negative to 0, and we have the outputs of a K-neuron perceptron for N samples.

Let's turn our attention to the learning rule. Given n^{th} training sample, the update to the weight connecting input attribute i to neuron k is:

$$\begin{aligned} e_{nk} &= y_{nk} - \hat{y}_{nk} \\ w_{ki} &:= w_{ki} + \alpha e_{nk} x_{ni}, \end{aligned}$$

where y_{nk} is the desired output, and $\hat{y}_{nk}$ is the computed output of neron k for the n^{th} training sample. The value e_{nk} is the error of neuron k on training point n and α is the learning parameter that you set to a small value (less than 1). For batch learning, we update the weights based on the average of the errors from all N training points. Thus we compute[1]:

$$\begin{aligned} \Delta w_{ki} &= \sum_{n=1}^{N} e_{nk} x_{ni} \\ w_{ki} &:= w_{ki} + \alpha \frac{1}{N} \Delta w_{ki} \end{aligned}$$

[1] Note, the Δ in Δw and Δb is not an operator on w and b, but part of the symbol denoting the "change in w" and "change in b".

Similarly, the update for bias k comes out to:

$$\Delta b_k = \sum_{n=1}^{N} e_{nk}$$

$$b_k := b_k + \alpha \frac{1}{N} \Delta b_k$$

Once again, with matrix arithmetic we have:

$$\begin{bmatrix} x_{11} & \dots & x_{N1} \\ \vdots & \ddots & \vdots \\ x_{1D} & \dots & x_{ND} \end{bmatrix} \begin{bmatrix} e_{11} & \dots & e_{1K} \\ \vdots & \ddots & \vdots \\ e_{N1} & \dots & e_{NK} \end{bmatrix} = \begin{bmatrix} \sum_{n=1}^{N} e_{n1}x_{n1} & \dots & \sum_{n=1}^{N} e_{nK}x_{n1} \\ \vdots & \ddots & \vdots \\ \sum_{n=1}^{N} e_{n1}x_{nD} & \dots & \sum_{n=1}^{N} e_{nK}x_{nD} \end{bmatrix} = \begin{bmatrix} \Delta w_{11} & \dots & \Delta w_{1K} \\ \vdots & \ddots & \vdots \\ \Delta w_{D1} & \dots & \Delta w_{DK} \end{bmatrix},$$

where

$$\begin{bmatrix} e_{11} & \dots & e_{1K} \\ \vdots & \ddots & \vdots \\ e_{N1} & \dots & e_{NK} \end{bmatrix} = \begin{bmatrix} y_{11} & \dots & y_{1K} \\ \vdots & \ddots & \vdots \\ y_{D1} & \dots & y_{DK} \end{bmatrix} - \begin{bmatrix} \hat{y}_{11} & \dots & \hat{y}_{1K} \\ \vdots & \ddots & \vdots \\ \hat{y}_{D1} & \dots & \hat{y}_{DK} \end{bmatrix}.$$

That's the computation for computing the updates to the weights of the pereptron in matrix form.

Let's recap the whole matrix based computation for the Perceptron model. The parameters of the peceptron can be specified by $\mathbf{W}$, a $D \times K$ weight matrix, and $\mathbf{b}$, a K-dimensional vector (both initialised to random values at first). Given $\mathbf{X}$, an $N \times D$ matrix of inputs the output of the model can be computed like so

$$\hat{\mathbf{Y}} = \sigma_{\text{hardlim}}(\mathbf{X} \cdot \mathbf{W} + \mathbf{b}), \tag{1}$$

where $\mathbf{b}$ is added column-wise to the $N \times K$ result of the matrix dot product $\mathbf{X} \cdot \mathbf{W}$; the hardlim function

$$\sigma_{\text{hardlim}}(v) = \begin{cases} 1 & v > 0 \\ 0 & v \leq 0 \end{cases} \tag{2}$$

is applied individually to every element of the matrix in the function argument. Given $\mathbf{Y}$, an $N \times K$ matrix of target labels, the Perceptron training rule in matrix forms is as follows:

$$\mathbf{E} = \mathbf{Y} - \hat{\mathbf{Y}} \tag{3}$$

$$\Delta\mathbf{W} = \mathbf{X}^T \cdot \mathbf{E} \tag{4}$$

$$\Delta\mathbf{b} = \begin{bmatrix} \sum_{n=1}^{N} e_{n1} & \sum_{n=1}^{N} e_{n2} & \dots \sum_{n=1}^{N} e_{nK} \end{bmatrix} \tag{5}$$

$$\mathbf{W} := \mathbf{W} + \frac{\alpha}{N}\Delta\mathbf{W} \tag{6}$$

$$\mathbf{b} := \mathbf{b} + \frac{\alpha}{N}\Delta\mathbf{b}, \tag{7}$$

where $\mathbf{E}$ is a $N \times K$ matrix, $\mathbf{X}^T \cdot \mathbf{E}$ is a dot product of transpose of $\mathbf{X}$ and $\mathbf{E}$, $\Delta\mathbf{W}$ is a $D \times K$ matrix, $\Delta\mathbf{b}$ is a K-dimensional vector and α is a scalar value less than 1.

Exercise 1: Implement the matrix-based perceptron learning rule (equations 1-7) using the Numpy library.

Start by creating a new PyCharm project (select Anaconda's cosc343 environment for the interpreter) and create a new script, adding the following code:

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

X = np.array([[-1, -1],
              [-1, 1],
              [1, -1],
              [1, 1]]).astype('float32')

y = np.array([0,
              0,
              0,
              1]).astype('uint8')
```

The code above loads the following dataset:

X		y
x_1	x_2	
-1	-1	0
-1	1	0
1	-1	0
1	1	1

Recall that a perceptron can only classify linearly separable problem. Since we are working with a 2-attribute input, you can visualise it on a plot. Add the following code to your script:

```
plt.scatter(X[:3,0],X[:3,1],c='red')
plt.scatter(X[3,0],X[3,1],c='blue')
plt.xlabel('x_1')
plt.ylabel('x_2')
plt.show()
```

The first line plots the first three points (those that have label 0) from the table, coloured in red. The syntax `X[:3,0]` selects the first three rows and the first column from `X` (attribute x_1 of the first three samples from `X`) and the syntax `X[:3,1]` selects the first three rows and the second column from X (attribute x_2 of the first three samples from X).

The second line plots the fourth point (one that has label 1) from the table, coloured in blue. The syntax X[3,0] selects the fourth row and the first column from X (attribute x_1 of the fourth samples from X) and the syntax X[3,1] selects the fourth row and the second column from X (attribute x_2 of the fourth sample from X). The following two lines add labels to the plot and the last line shows the plot.

Next, you need to infer the number of training samples and the dimensionality of input from the data. Since X is a numpy array, you can do:

```
N,D = np.shape(X)
```

For this problem y has only two labels, so it's a binary classification problem, and so we can use only one neuron.

```
K = 1
```

In the matrix equations you have been given, the label matrix Y needs to be in the $N \times K$ format. Since y is a 4-dimensional vector, we need to convert it to a 4×1 matrix. You can do this using Numpy's expand_dims function:

```
Y = np.expand_dims(y,axis=1)
```

The code above converts an N-dimensional vector y to an $N \times 1$ matrix (no data is added, just the structure of the Numpy array is reconfigured).

Next set the maximum number of epochs to train for and the learning rate. In Sklearn's terminology epochs are refereed to as iterations, so we'll stick to this conventions.

```
max_iter = 100
learning_rate = 0.1
```

Now, create the randomly initialised values for the weight matrix W and bias vector b. Since D=2 and K=1 it will be a 2×1 weight matrix and a 1-dimensional array. But using variables D and K makes this code generic for arbitrary D and K.

```
W = np.random.randn(D,K)
b = np.random.randn(K)
```

This is all that you should need to be able to create a training loop implementing matrix version of the batch version of the Perceptron learning rule given in equations 1-7:

```
for i in range(max_iter):

        # Implement the perecptron
        # learning rule
        # .
```

Hints!

- Numpy's dot function computes the matrix dot product of two matrices
- Syntax `X.T` gives you the transpose of Numpy array `X`;
- Numpy array allows addition with the '+' operator of a $N \times K$ matrix and a K-dimensional vector (it will add the vector column-wise to the matrix);
- For hardlimiting function you can use the following Numpy syntax `Yhat[Yhat < 0]=0`, which selects all indices of `Yhat` array with values less than zero and overwrites the values of `Yhat` at those indices with 0;
- To get a sum of each column of Numpy variable `E` you can do `np.sum(E, axis=0)`
- You might want to print/keep a track of the total number of errors per iteration; assuming `E` is a Numpy array of 0's, 1's and -1's, you find the number of elements that are not equal to zero using the following syntax: `np.sum(E!=0)`;
- You might as well break out of the loop once the total number of errors is zero...since at this point, the learning rule will ceased to updated the weights and biases.

Make sure that the learning rule works (that number of errors decreases as the training proceeds).

Exercise 2: Port your code into the Sklearn style Perceptron model.

Download `Perceptron.py` and `helper.py` from Blackboard and place them inside your project folder.

`Perceptron.py` implements `Perceptron` class in Sklearn library style. It provides initialisation method that allows you to specify the hyper-parameters of the model, the `predict` method that computes model output, and parts of the `fit` method where the model is to be trained.

The `fit` method implements all the logic for inferring the number of classes and input attributes from the data as well as converting the class labels to one hot encoding. It also initialises the weight and bias values of the perceptron on the first call of the `fit` method. But it is missing the actual learning part. The contents of the loop starting at line 137 are empty – that's where the Perceptron learning rule needs to go. The code from the previous exercise, which iterates over the epochs and updates the weight and biases, should work fine here. In the `fit` method the following variables are already defined for you: N, D, K, an $N \times D$ matrix X, an $N \times K$ matrix Y in on hot encoding format, and an $D \times K$ weight matrix `self.W` and a K-dimensional bias vector `self.b`. So the port should require only minimal changes. If you are printing to the console information about the number of errors as the model is training, you might want to condition this on the `self.verbose` variable of the `Perceptron` object.

Once you're finished with the learning rule in the `fit` method, the `Perceptron` class should be ready (meaning everything else is already implemented inside). Test how it all works with the following code:

```python
import helper
from Perceptron import Perceptron

X, y = helper.load_and()

model = Perceptron(learning_rate=0.1,max_iter=1,verbose=False)

for epoch in range(200):
    model.fit(X,y)
    model.plot_classified_regions(X,y,titleStr="Epoch %d" % (epoch+1))
```

The `helper.py` script provides various methods for loading data; in this case you're loading the same AND problem you used in the previous exercise. You could set `max_iter=200` when initialising the Perceptron and invoke `fit` just once. However, we want to display after every epoch a visualisation of the training that `Perceptron` class provides, so set `max_iter=1` on initialisation and then call the `fit` method 200 times (this should work fine as the `fit` only initialises `self.W` and `self.b` values on its first invocation, afterwards the values from the last fit are preserved). But after each call, use the `plot_classified_regions` method (implemented for you in `Perceptron`). For that visualisation to work as intended, you need to change the way PyCharm displays plots. By default it displays plots in the toolwindow, which (for some odd reason) disables Matplotlib's ability to redraw plots. So you need to go to PyCharm→Preferences→Tools→Python Scientific and disable the "Show plots in toolwindow" option.

Can you see how the Perceptron trains?

Try to load and train on a different dataset:

```python
X, y = helper.load_xor()
```

How does the learning proceed now? Does it work ok? Why yes or why not?

How about a 3-class dataset? Load the following dataset:

```
X, y = helper.load_iris2D()
```

The Iris dataset is a well known benchmark dataset with 150 points in total of 4-dimension and 3 classes (50 per each class). The helper function loads a 2D version of that dataset (2 dimensions are dropped) so that you can have a visualisation of the classification regions. How does a perceptron get on with this dataset?

Exercise 3: Write a new script and test how Perceptron does on a MNIST-like dataset.

To load the dataset, use the provided helper method:

```
X, y = helper.load_digits()
```

This dataset is a set of 8x8 images of downsampled MNIST digits, so the total number of pixels is $D = 64$ and the number of classes is $K = 10$.

Once you train the model, you can use its built-in method `plot_classified_images` to show the input images and corresponding labels (as predicted by the model). The code to plot the first 16 images of the text input would be:

```
model.plot_classified_images(X_test[:16])
```

Also, don't forget to evaluate and check the accuracy of the model on the test set.