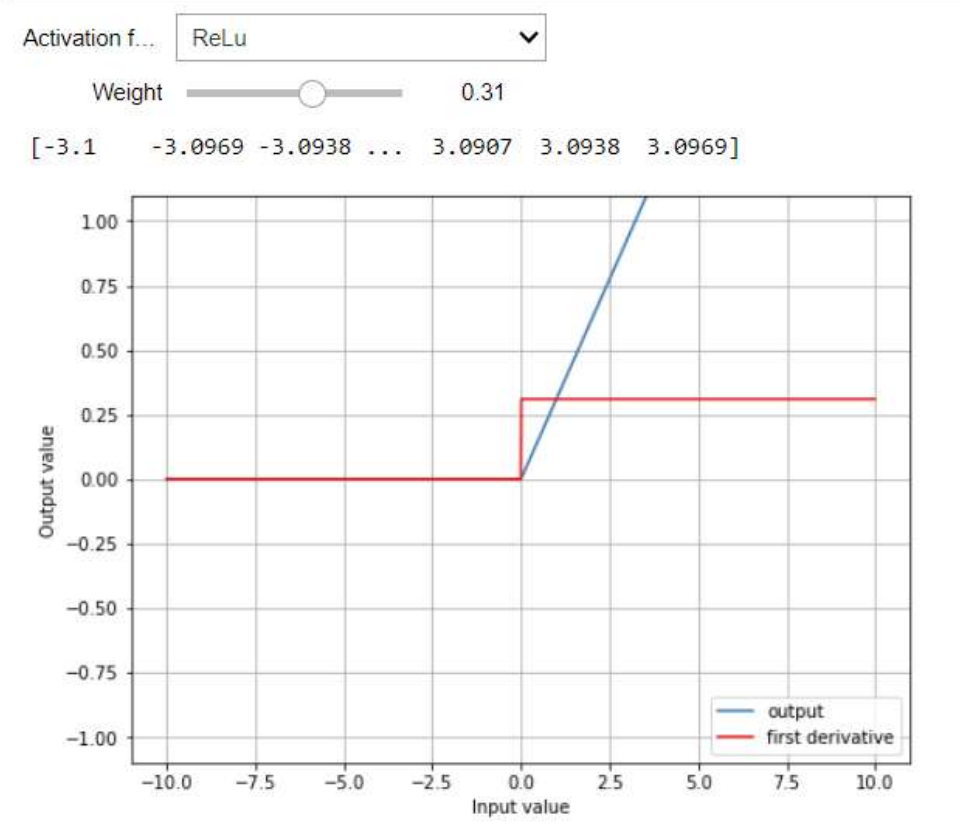
ML-PW-09

by Aurélien Héritier and Jean Nanchen

Task 1

Implementation of the ReLu activation function (source code) + example of visualization

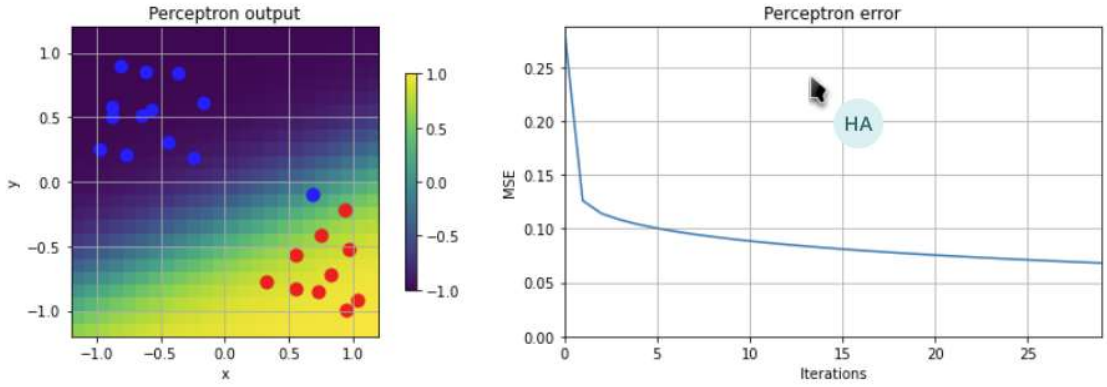
```python
def reLu(neta):
    print(neta)
    output = np.maximum(0,neta)
    d_output = np.array([1 if out >= 0 else 0 for out in neta])
    return (output, d_output)
```



Task 2

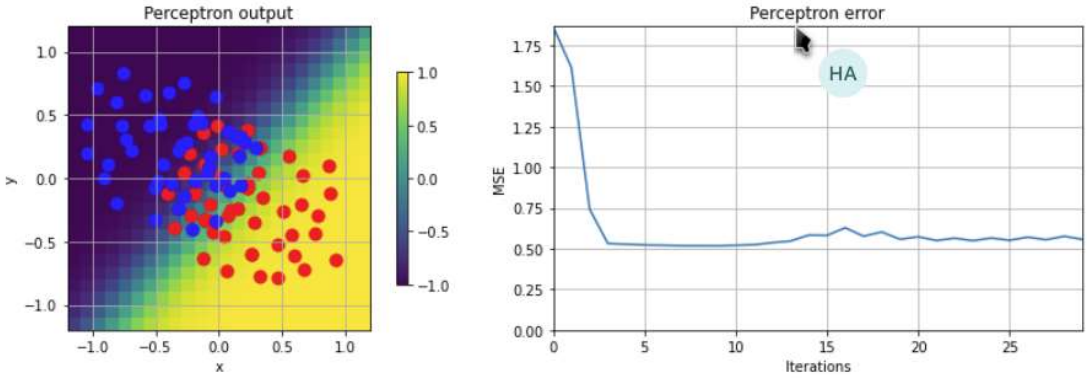
Answer questions 1-3 from the 4_delta-rule notebook and present the resulting plot when the option SHOW_VIDEO is set to False

1. What happens if the boundaries between both classes are well defined?



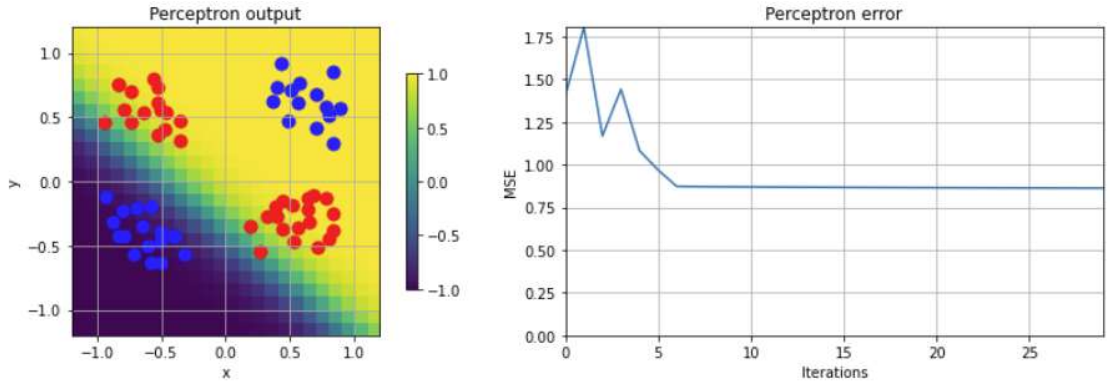
The line between the 2 classes separate well the 2 separate cluster. There is a little error dot, and the system calculate the good separation.

2. What happens if the classes overlap? What could you say about oscillations in the error signal?



The oscillation is because data (blue and red) are in the same place at the middle.

3. What happens if it is not possible to separate the classes with a single line? What could you say about local minima?

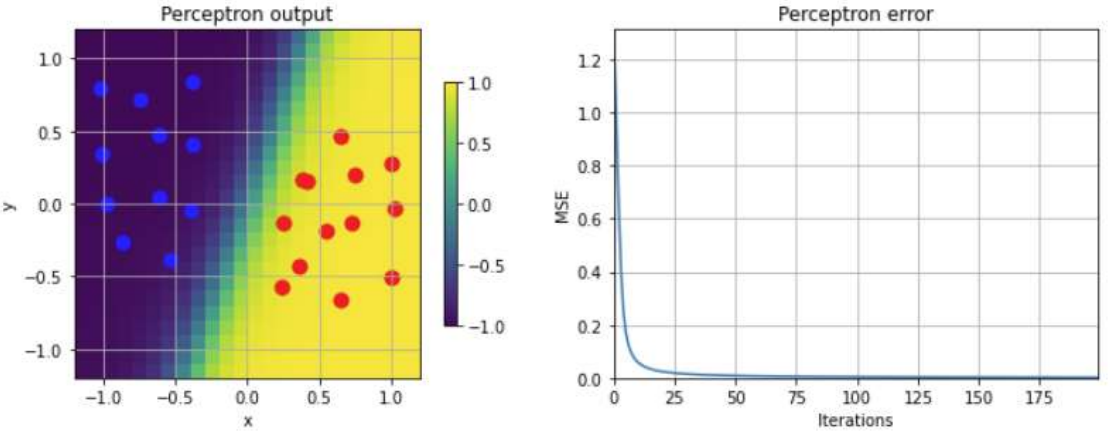


There is a separation between one cluster of blue and the rest of the red dots. But because there is only one line, the dataset cannot be well separated.

Task 3

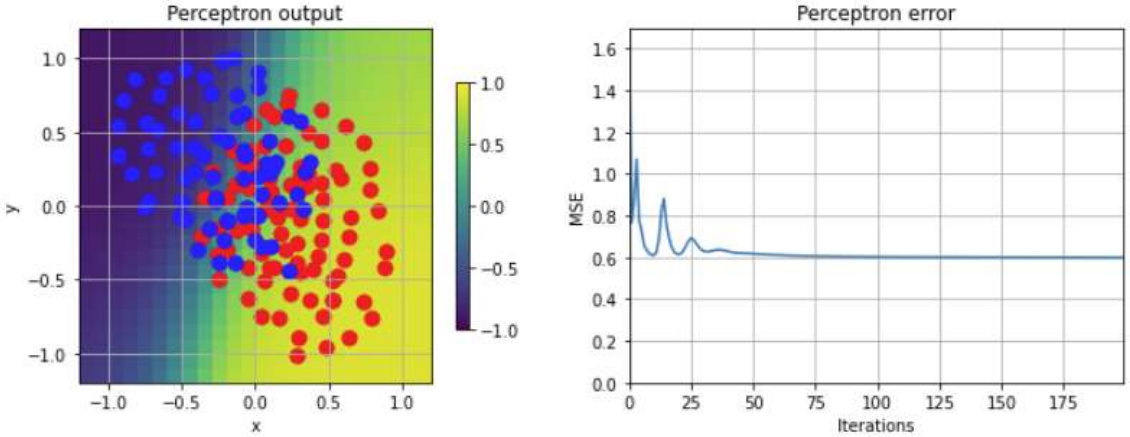
Answer questions 1-4 from the 5_backpropagation notebook and present the resulting plot when the option SHOW_VIDEO is set to False

1. What happens if the boundaries between both classes are well defined?



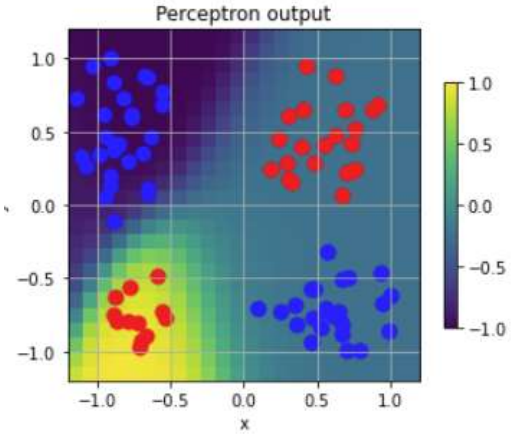
Good separation

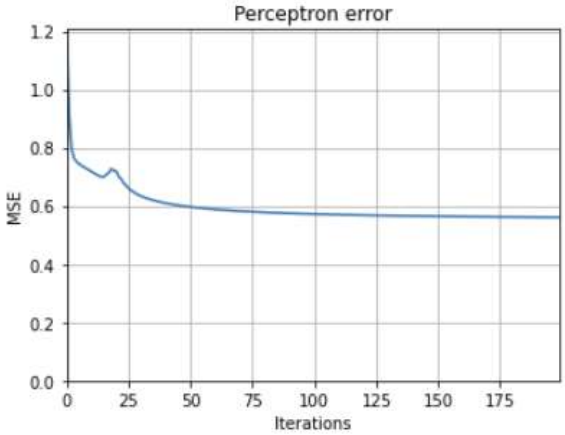
2. What happens if the classes overlap? What could you say about oscillations in the error signal?



MSE is high because the two clusters are overlap

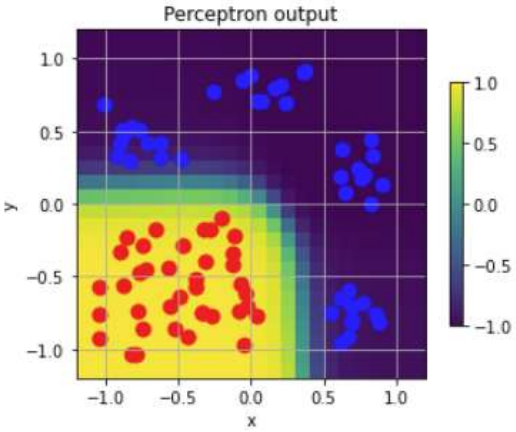
3. What happens if it is not possible to separate the classes with a single line? What could you say about local minima?

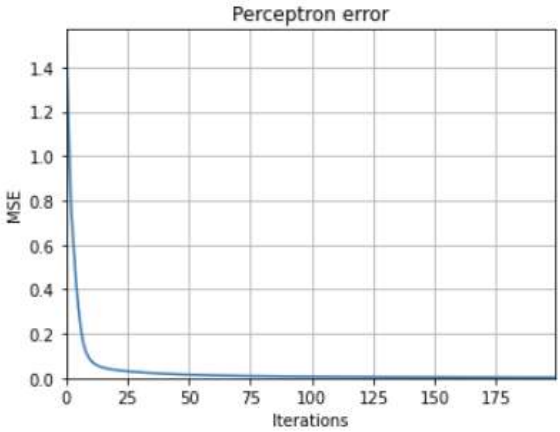




If it's not possible to separater the class with a single line, it will only separate one cluster. In this case it is the smaller cluster (local minima).

4. What happens if the points of one of the classes are separated in subgroups (blobs)?





Great differentiation between blue and red dots

Task 4

Implementation (source code of the modified function) of the Backpropagation with momentum algorithm

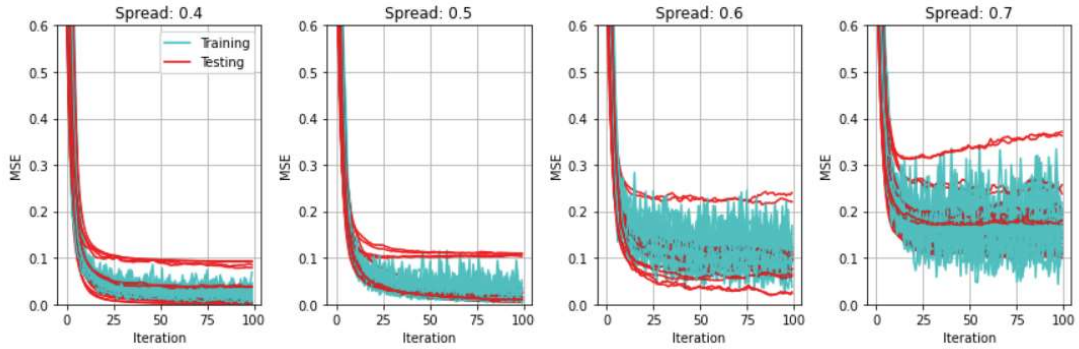
```
def fit(self, data_train, data_test=None, learning_rate=0.1, momentum=0.5, epochs=100):
    '''
    Online learning.
    :param data_train: A tuple (X, y) with input data and targets for training
    :param data_test: A tuple (X, y) with input data and targets for testing
    :param learning_rate: parameters defining the speed of learning
    :param epochs: number of times the dataset is presented to the network for learning
    '''
    X = np.atleast_2d(data_train[0])                      # Inputs for training
    #temp = np.ones([X.shape[0], X.shape[1]+1])            # Append the bias unit to the input layer
    #temp[:, 0:-1] = X
    #X = temp                                             # X contains now the inputs plus a last column of ones (bias unit)
    y = np.array(data_train[1])                           # Targets for training
    error_train = np.zeros(epochs)                        # Initialize the array to store the error during training (epochs)
    if data_test is not None:                             # If the test data is provided
        error_test = np.zeros(epochs)                     # Initialize the array to store the error during testing (epochs)
        out_test = np.zeros(data_test[1].shape)           # Initialize the array to store the output during testing

    a = []                                                # Create a list of arrays of activations
    for l in self.layers:
        a.append(np.zeros(l))                             # One array of zeros per layer

    for k in range(epochs):                               # Iterate through the epochs
        error_it = np.zeros(X.shape[0])                   # Initialize an array to store the errors during training (n examples)
        for it in range(X.shape[0]):                      # Iterate through the examples in the training set
            i = np.random.randint(X.shape[0])             # Select one random example
            a[0] = X[i]                                   # The activation of the first layer is the input values of the example

                                                          # Feed-forward
            for l in range(len(self.weights)):            # Iterate and compute the activation of each layer
                a[l] = np.concatenate((a[l], np.ones(1)))   # Append one for the bias
                a[l+1] = self.activation(np.dot(a[l], self.weights[l])) # Apply the activation function to the product input.weights

            error = a[-1] - y[i]                          # Compute the error: output - target
            error_it[it] = np.mean(error ** 2)            # Store the error of this iteration (average of all the outputs)
            deltas = [error * self.activation_deriv(a[-1])] # Ponderate the error by the derivative = delta

                                                          # Back-propagation
                                                          # We need to begin at the layer previous to the last one (out->in)
            for l in range(len(a) - 2, 0, -1):            # Append a delta for each layer
                deltas.append(deltas[-1].dot(self.weights[l].T) * self.activation_deriv(a[l]))
                deltas[-1] = deltas[-1][:-1]              # delete the delta of the bias since bias units are not connected backwards
            deltas.reverse()                              # Reverse the list (in->out)

                                                          # Update
            for i in range(len(self.weights)):            # Iterate through the layers
                layer = np.atleast_2d(a[i])               # Activation
                delta = np.atleast_2d(deltas[i])          # Delta
                                                          # Compute the weight change using the delta for this layer
                                                          # and the change computed for the previous example for this layer


                delta_weights = (-learning_rate * layer.T.dot(delta)) + (momentum * self.weights[i])
                self.weights[i] += self.weights[i]
        error_train[k] = np.mean(error_it)                # Compute the average of the error of all the examples
        if data_test is not None:                         # If a testing dataset was provided
            error_test[k], _ = self.compute_MSE(data_test)  # Compute the testing error after iteration k

    if data_test is None:                                 # If only a training data was provided
        return error_train                                # Return the error during training
    else:
        return (error_train, error_test)                  # Otherwise, return both training and testing error
```

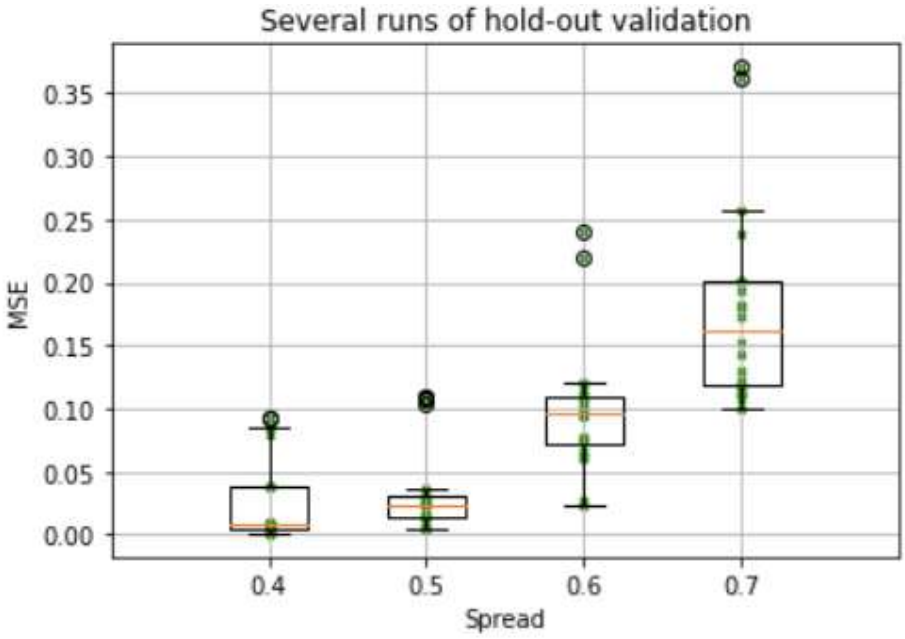
Task 5

Run notebooks 7 and 8, provide the final plots MSE vs spread and comment the difference between results

Notebook 7

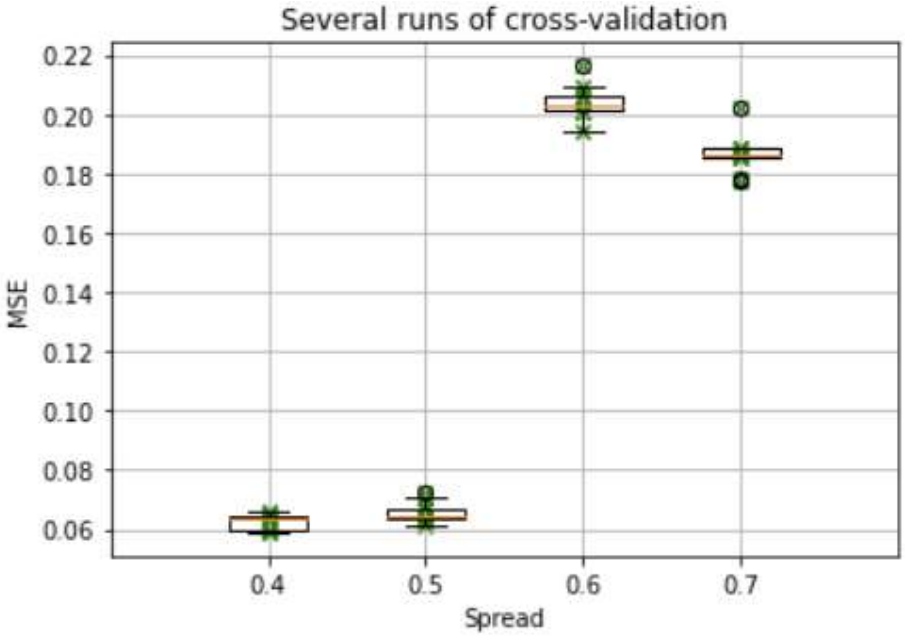
In this plot, we can observe that reds curves don't finish at the same MSE (at the last iteration). Somes data partitions are memorized by the neural network and the training error will be low and the testing error will be high.





Notebook 8

The MSE value is more homogenic (closer to each other) with cross-validation than the hold-out validation



Task 6

Run notebook 9 for three different spread values (e.g., 0.3, 0.5 and 0.7), describe the topology of the final model chosen (e.g., layers, hidden neurones, activation functions), the final learning parameters (learning rate, number of iterations, momentum term, etc) and justify your selection (e.g., based on the plots of MSE vs parameters)

To select the learning parameters we need to follow this rules :

- the error curve oscillates -> reduce the learning rate
- the error curve is very smooth and does not change -> increase the learning rate
- the model does not converge -> try different values of momentum

The compute time is too big, we don't run this notebook with multiple values.

Best case :
65 epochs
4 neurones