PW 09 Bütikofer Jaggi

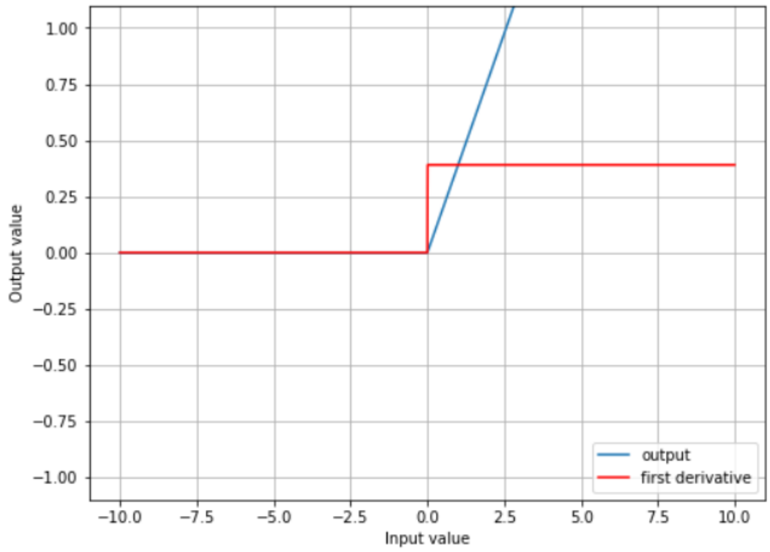
1. The Perceptron and the Delta rule

1_activation_function

```
def relu(neta):
    '''the activation function of a rectified Linear Unit (ReLU)'''
    output = neta * (neta > 0)
    d_output = 1.0 * (neta > 0)
    return (output, d_output)
```

Activation f... Relu

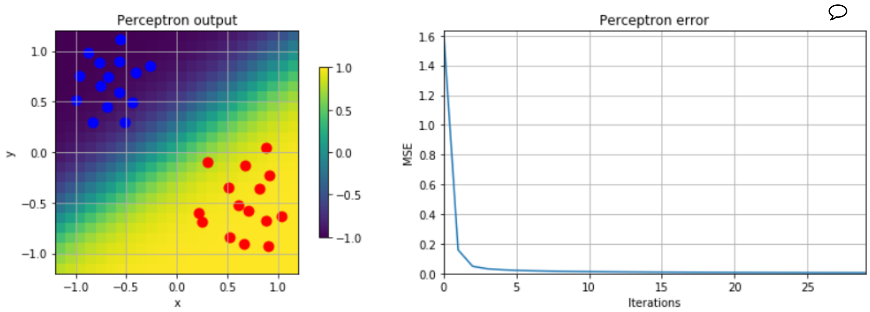
Weight 0.39



The sigmoid has two horizontal asymptote in y=0 and in y = 1, the hyperbolic tangent in y =-1 and y=1 and the linear function doesn't have asymptote. The derivative of the sigmoid and the hyperbolic function look like a Gaussian distribution and the linear function derivative is a straight.

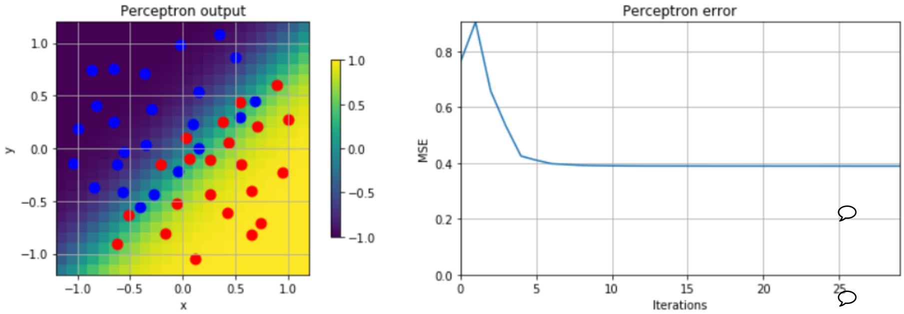
4_1_delta_rule_points

When well defined



The peceptron defines well 2 different classes. The error goes with a few iteration near 0

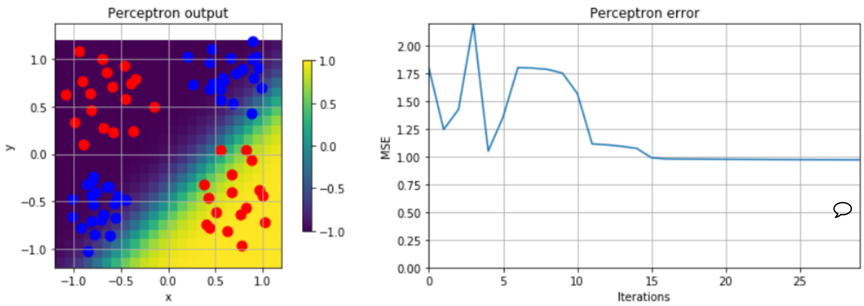
When classes overlap



Because of the overlap the error function doesn t go near 0 but near 0.4 (because of the overlapping points). It doesn t take more iteration to converge to 0.4.

There is only one oscillation, no significant.

Not with single line



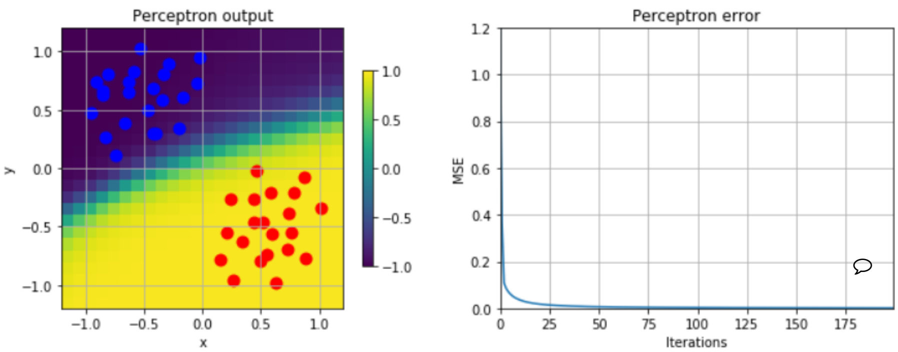
The error converge near 1.0. When there is not a single line separation, the number iteration increase and the error is higher.

Local minima are found after fewer iteration than the global minima, we converge to the global minima.

2. Backpropagation

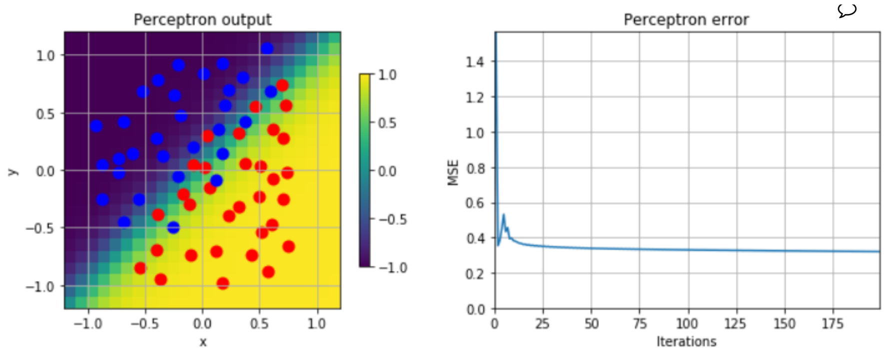
5_backpropagation

When well defined



The error converges to 0. It converge with a few iteration.

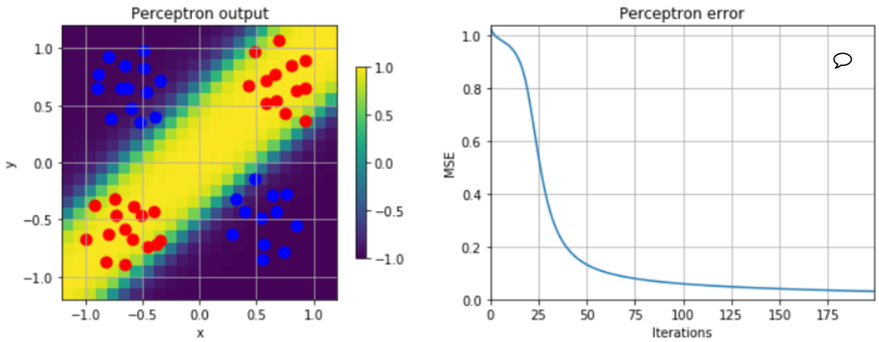
When classes overlap



Because of the overlap the error function doesn t go near 0 but near 0.4 (because of the overlapping points). It doesn t take more iteration to converge to 0.4.

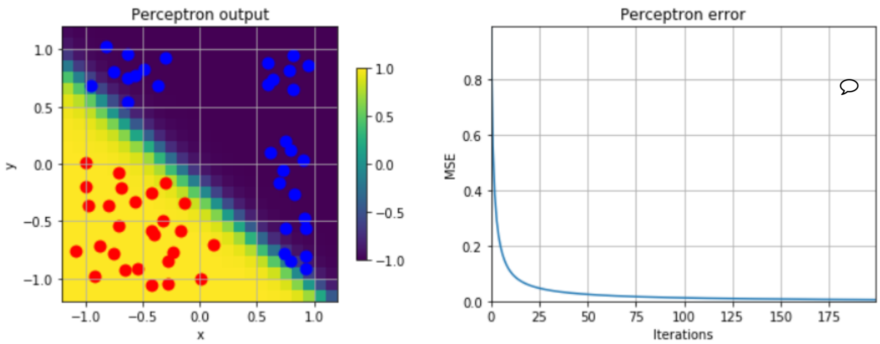
There is only one oscillation, no significant

Not with single line



It’s possible to separate the dataset with 2 lines. The error converge to 0 with more iteration. There is no local minima

Separated in subgroups (blobs)

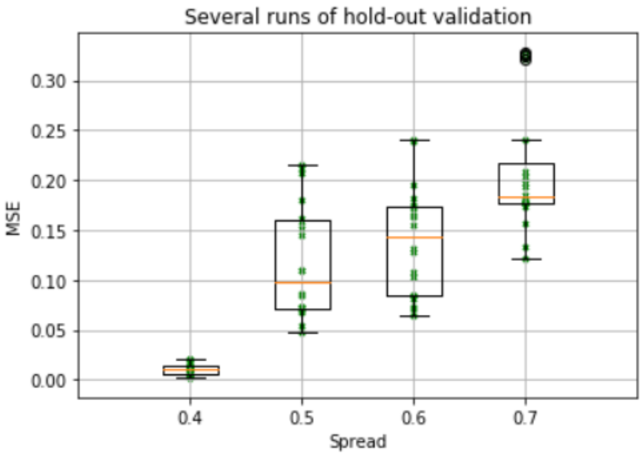


It’s possible to separate the classes in 2 groups. The error converge to 0 and it needs only a few iteration

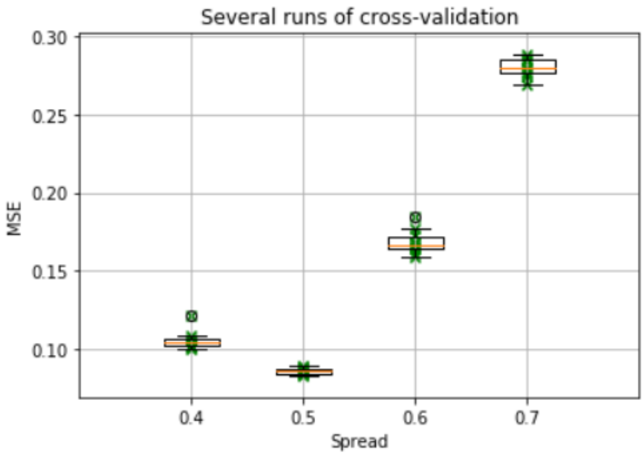
```
class MLP:
    '''
    This code was adapted from:
    https://rolisz.ro/2013/04/18/neural-networks-in-python/
    '''
    def __tanh(self, x):
        '''Hyperbolic tangent function'''
        return np.tanh(x)

    def __tanh_deriv(self, a):
        '''Hyperbolic tangent derivative'''
        return 1.0 - a**2

    def __logistic(self, x):
        '''Sigmoidal function'''
        return 1.0 / (1.0 + np.exp(-x))

    def __logistic_derivative(self, a):
        '''sigmoidal derivative'''
        return a * ( 1 - a )

    def __init__(self, layers, activation='tanh'):
        '''
        :param layers: A list containing the number of units in each layer.
        Should be at least two values
        :param activation: The activation function to be used. Can be
        "logistic" or "tanh"
        '''
        self.n_inputs = layers[0]                               # Number of input
        self.n_outputs = layers[-1]                             # Number of ouput
        self.layers = layers
                                                                # Activation func
        if activation == 'logistic':
            self.activation = self.__logistic
            self.activation_deriv = self.__logistic_derivative
        elif activation == 'tanh':
            self.activation = self.__tanh
            self.activation_deriv = self.__tanh_deriv

        self.init_weights()                                     # Initialize the

    def init_weights(self):
        '''
        This function creates the matrix of weights and initialiazes their values
        '''
        self.weights = []                                       # Start with an e
        self.delta_weights = []
        for i in range(1, len(self.layers) - 1):                # Iterates throug
                                                                # np.random.rando
                                                                # of random float
                                                                # (self.layers[i]
            self.weights.append((2 * np.random.random((self.layers[i - 1] + 1, se
            self.delta_weights.append(np.zeros((self.layers[i-1] +1, self.layers[
                                                                # delta_weights a
                                                                # Append a last s
        self.weights.append((2 * np.random.random((self.layers[i] + 1, self.layer
        self.delta_weights.append(np.zeros((self.layers[i] + 1, self.layers[i + 1

    def fit(self, data_train, data_test=None, learning_rate=0.1, momentum=0.0 ,ep
        '''
        Online learning.
        :param data_train: A tuple (X, y) with input data and targets for trainin
        :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 fo
        '''
        X = np.atleast_2d(data_train[0])                        # Inputs for trai
        temp = np.ones([X.shape[0], X.shape[1]+1])              # Append the bias
        temp[:, 0:-1] = X
        X = temp                                                # X contains now
        y = np.array(data_train[1])                             # Targets for tra
        error_train = np.zeros(epochs)                          # Initialize the
        if data_test is not None:                               # If the test dat
            error_test = np.zeros(epochs)                       # Initialize the
            out_test = np.zeros(data_test[1].shape)             # Initialize the

        a = []                                                  # Create a list o
        for l in self.layers:
            a.append(np.zeros(l))                               # One array of ze

        for k in range(epochs):                                 # Iterate through
            error_it = np.zeros(X.shape[0])                     # Initialize an a
```

```
            for it in range(X.shape[0]):                        # Iterate through
                i = np.random.randint(X.shape[0])               # Select one rand
                a[0] = X[i]                                     # The activation

                                                                # Feed-forward
                for l in range(len(self.weights)):              # Iterate and com
                    a[l+1] = self.activation(np.dot(a[l], self.weights[l])) # App

                error = a[-1] - y[i]                            # Compute the err
                error_it[it] = np.mean(error ** 2)              # Store the error
                deltas = [error * self.activation_deriv(a[-1])] # Ponderate the e

                                                                # Back-propagatio
                                                                # We need to begi
                for l in range(len(a) - 2, 0, -1):              # Append a delta
                    deltas.append(deltas[-1].dot(self.weights[l].T) * self.activa
                deltas.reverse()                                # Reverse the lis

                                                                # Update
                for i in range(len(self.weights)):              # Iterate through
                    layer = np.atleast_2d(a[i])                 # Activation
                    delta = np.atleast_2d(deltas[i])            # Delta
                                                                # Compute the wei
                                                                # and the change
                    self.delta_weights[i] = (-learning_rate * layer.T.dot(delta))
                    self.weights[i] += self.delta_weights[i]            # Update

            error_train[k] = np.mean(error_it)                  # Compute the ave
            if data_test is not None:                           # If a testing da
                error_test[k], _ = self.compute_MSE(data_test)  # Compute the tes

        if data_test is None:                                   # If only a train
            return error_train                                  # Return the erro
        else:
            return (error_train, error_test)                    # Otherwise, retu

    def predict(self, x):
        '''
        Evaluates the network for a single observation
        '''
        x = np.array(x)
        temp = np.ones(x.shape[0]+1)
        temp[0:-1] = x
        a = temp
        for l in range(0, len(self.weights)):
            a = self.activation(np.dot(a, self.weights[l]))
        return a

    def compute_output(self, data):
        '''
        Evaluates the network for a dataset with multiple observations
        '''
        assert len(data.shape) == 2, 'data must be a 2-dimensional array'

        out = np.zeros((data.shape[0], self.n_outputs))
        for r in np.arange(data.shape[0]):
            out[r,:] = self.predict(data[r,:])
        return out

    def compute_MSE(self, data_test):
        '''
        Evaluates the network for a given dataset and
        computes the error between the target data provided
        and the output of the network
        '''
        assert len(data_test[0].shape) == 2, 'data[0] must be a 2-dimensional arr

        out = self.compute_output(data_test[0])
        return (np.mean((data_test[1] - out) ** 2), out)
```

4. Crossvalidation



The results vary a lot. We can see that as the spread increase the error rate logically increase too.



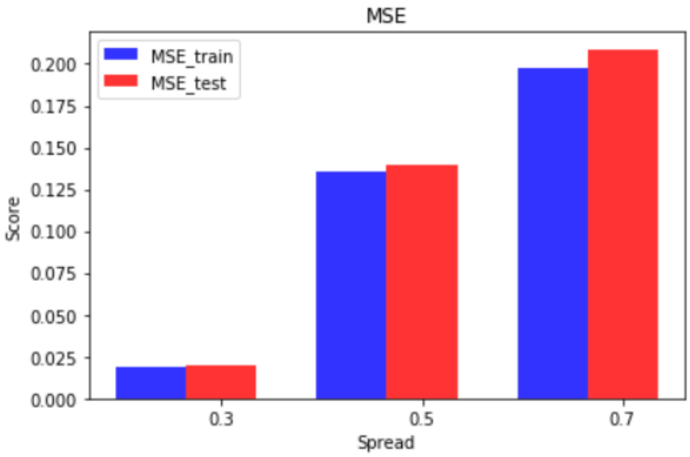
We can see that similar as the hold ou validation, when the spread increase the error rate increase too.

If we compare the results of the two methods, we see that the hold out can give the best result in some case but the worst too. the k-fold give a bit worse result on average but the results doesn't vary as much.

5. Model building

Spread (0.3, 0.5, 0.7)

```
Spread = 0.3
MSE training:  0.018772047265949793
MSE test:  0.020179179118967955
Confusion matrix:
[[100.   0.]
 [  0. 100.]]

 Spread = 0.5
MSE training:  0.1354363862451034
MSE test:  0.1392736811000205
Confusion matrix:
[[97.  3.]
 [ 4. 96.]]

 Spread = 0.7
MSE training:  0.19707509868666367
MSE test:  0.2086012143215099
Confusion matrix:
[[93.  7.]
 [ 5. 95.]]
```



The final result is 4 hidden neurons with 60 epochs. We can see that even if the spread is 0.7 we have a pretty good result. With 4 neurons the results are good and we have less computation time.