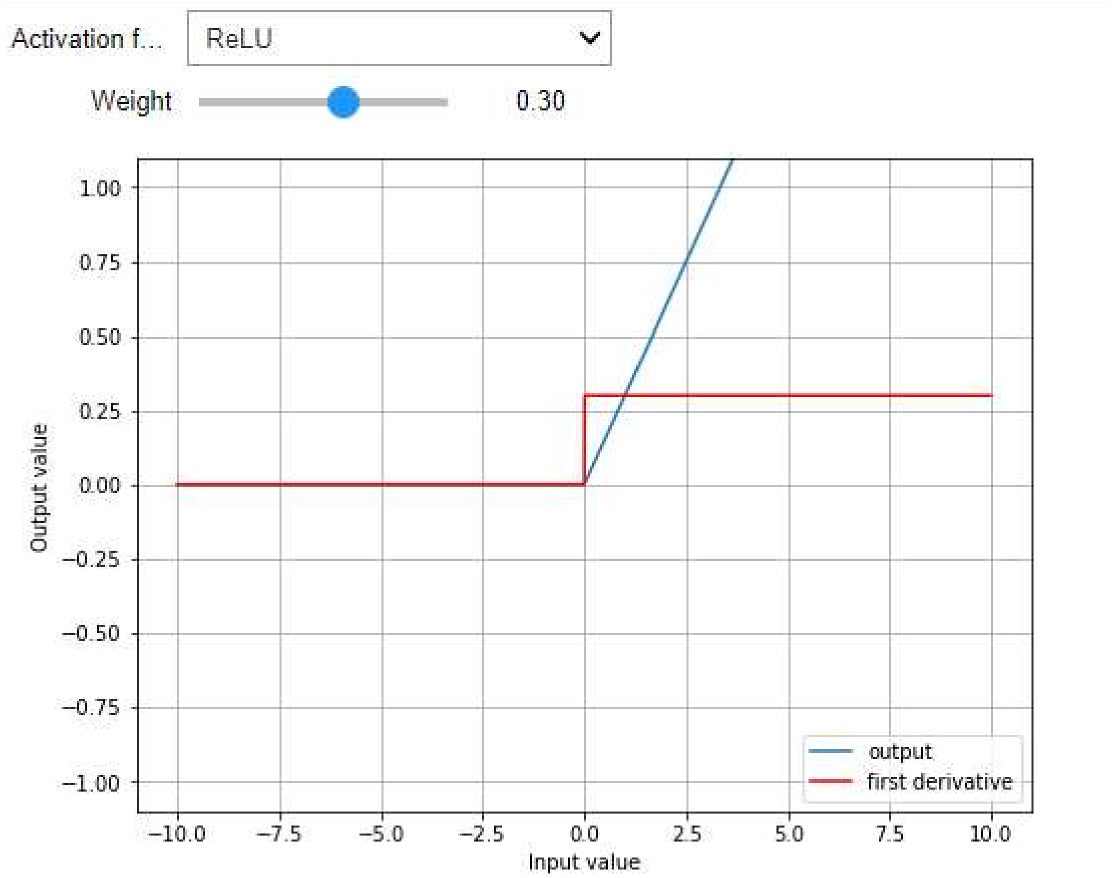
Abdalla Farid & Noah Graells

ReLU

In [ ]:

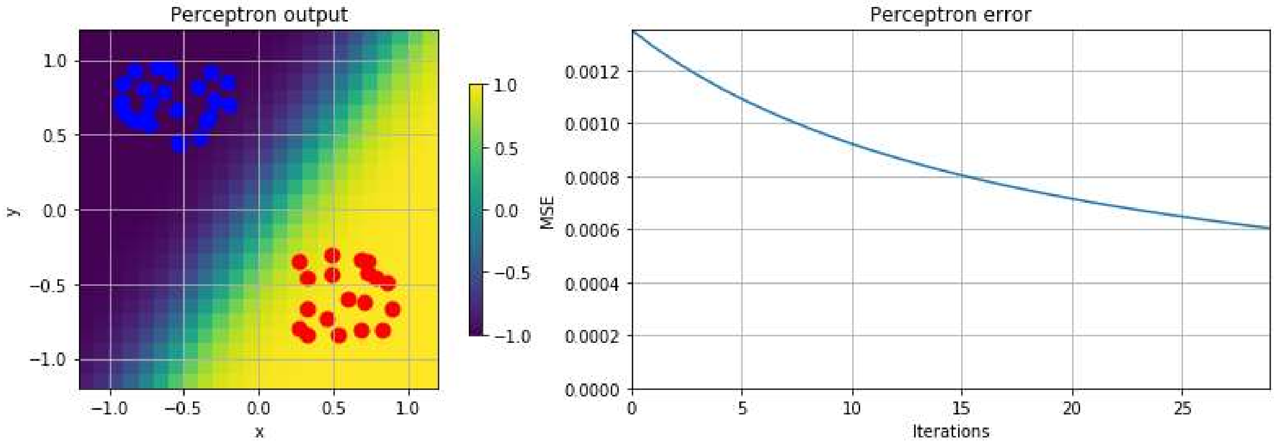
```
def relu(neta):
    output = np.array([0 if x < 0 else x for x in neta])
    d_output = np.array([0 if x < 0 else 1 for x in neta])
    return (output, d_output)
```



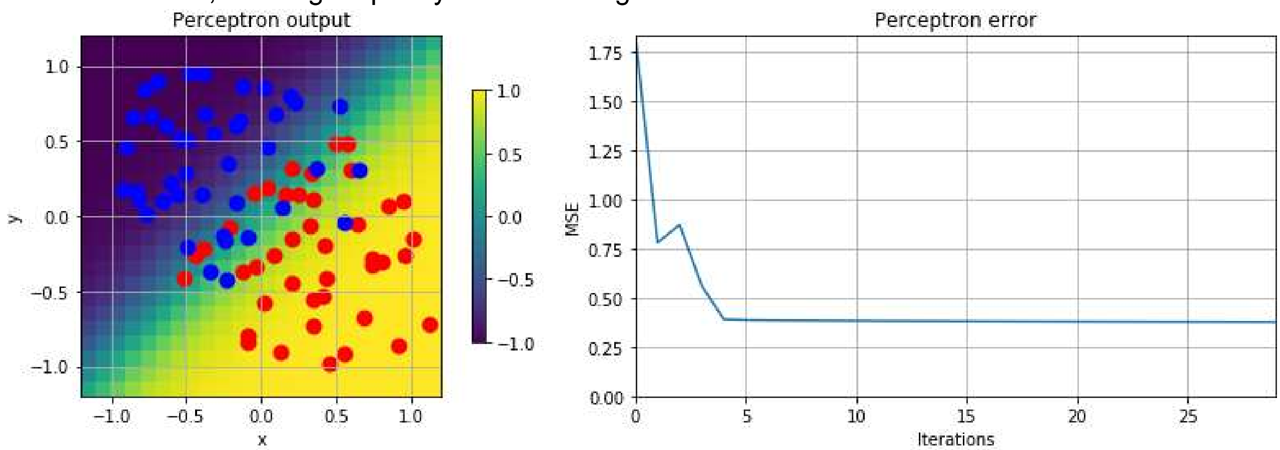
4_delta-rule

Answers

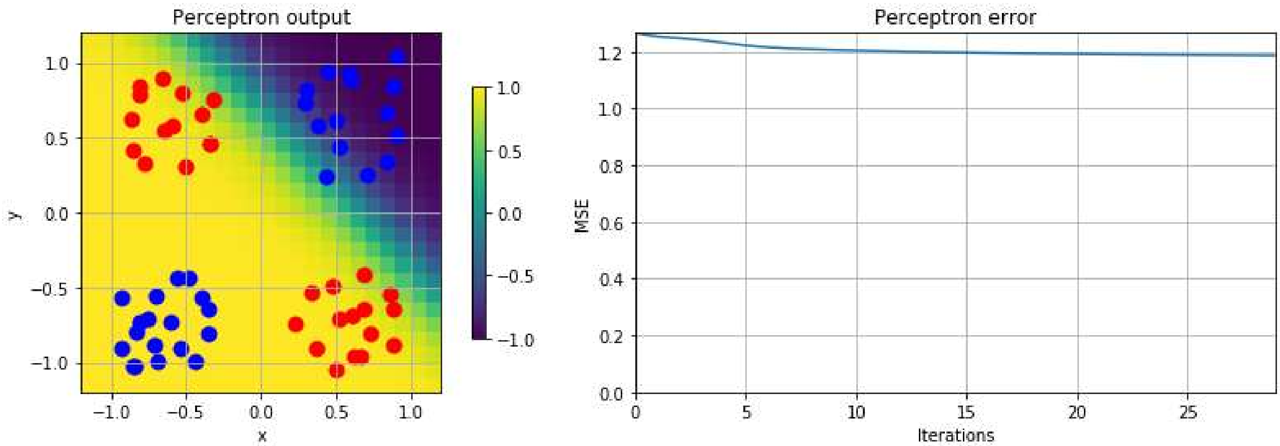
1. No oscillation, converges quickly and finds a good solution (low error)



2. Small oscillations, converges quickly. Error not as good as the first one but still ok.

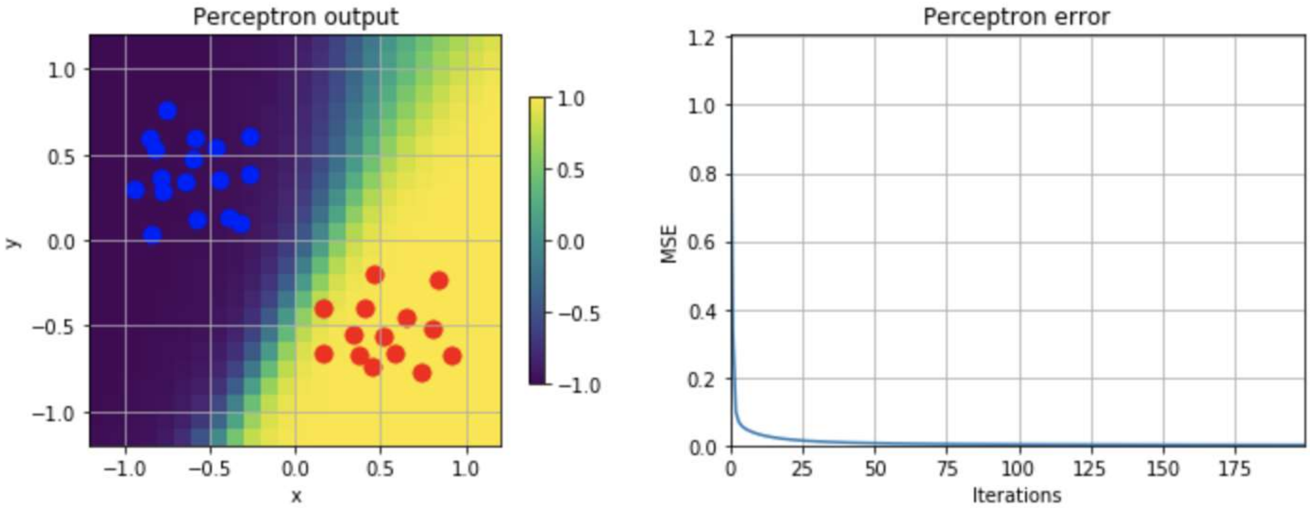


3. Big error, doesn't converge.

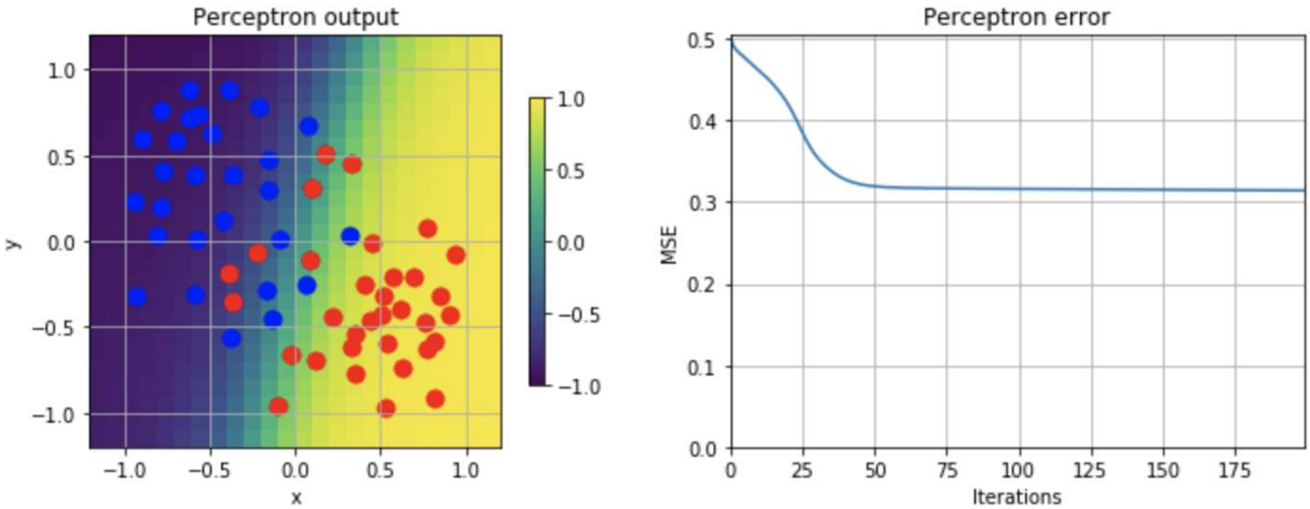


5_backpropagation

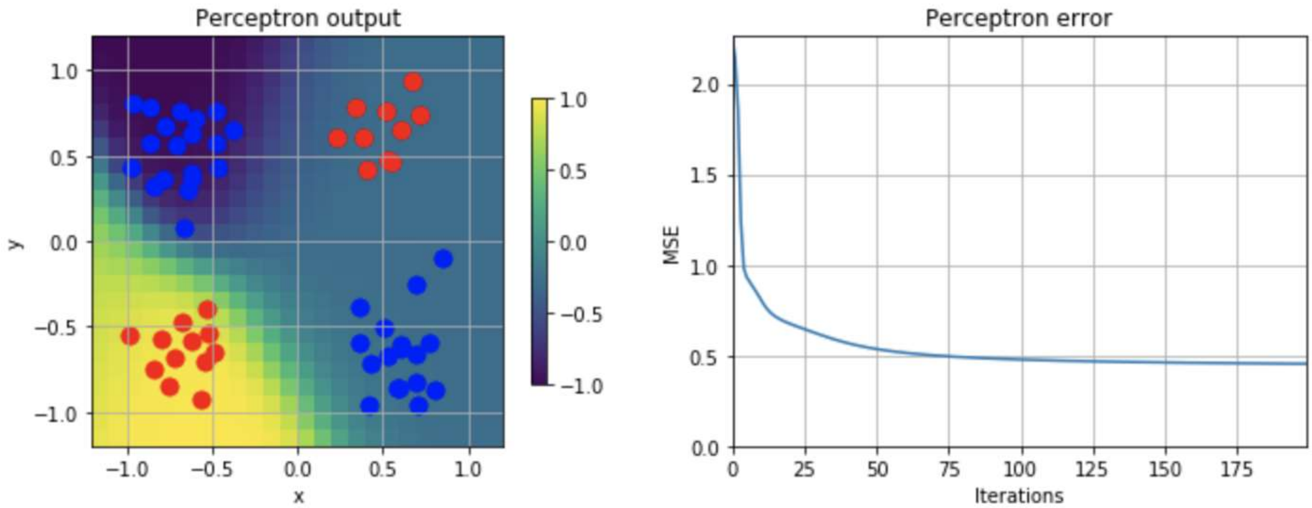
When classes are separable, we reach quickly the optimal solution with a really low error :



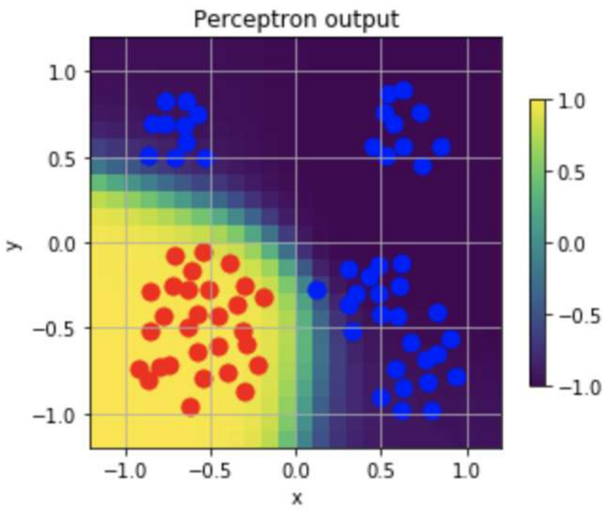
With overlapping classes, it converges quite quickly but the error is quite big since the classes aren't well defined. The convergence is smoother than without the backpropagation :

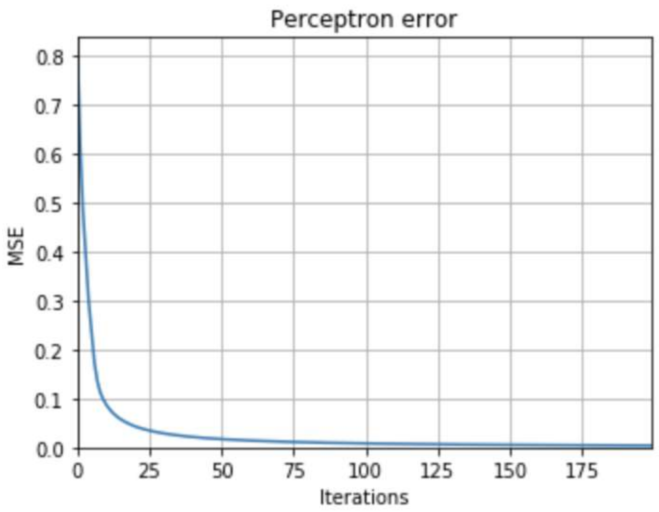


With non-separable classes, the error is big but not as big as the exercise 4. There is not really a local minima since there is an horizontal asymptote :



The result is similar to the first question since classes are separable. The convergences is not as quick though :





6_backpropagation_momentum

In [4]:

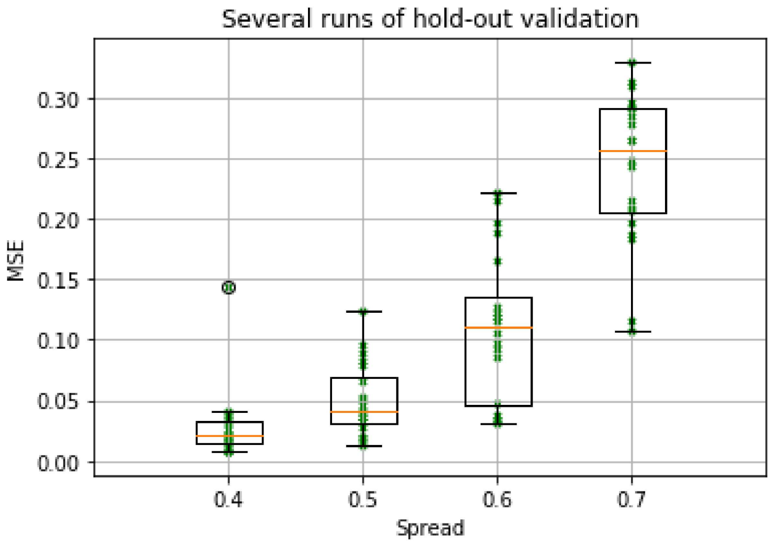
```
class MLP:
    '''
    This code was adapted from:
    https://rolisz.ro/2013/04/18/neural-networks-in-python/
    '''
    def __tanh(self, x):
        '''Hyperbolic tangent function'''
        return np.tanh(x)

    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 inputs (fir
st layer)
        self.n_outputs = layers[-1]                            # Number of ouputs (las
t layer)
        self.layers = layers
                                                               # Activation function
        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 weight
s of the MLP

    def init_weights(self):
        '''
        This function creates the matrix of weights and initialiazes their values to sm
all values
        '''
        self.weights = []                                      # Start with an empty l
ist
        for i in range(1, len(self.layers) - 1):               # Iterates through the
 layers
                                                               # np.random.random((M,
 N)) returns a MxN matrix
                                                               # of random floats in
```

```
 [0.0, 1.0).
                                                                  # (self.layers[i] + 1)
 is number of neurons in layer i plus the bias unit
            self.weights.append((2 * np.random.random((self.layers[i - 1] + 1, self.lay
ers[i] + 1)) - 1) * 0.25)
                                                                  # delta_weights are ini
tialized to zero
                                                                  # Append a last set of
 weigths connecting the output of the network
        self.weights.append((2 * np.random.random((self.layers[i] + 1, self.layers[i +
1])) - 1) * 0.25)

    def fit(self, data_train, data_test=None, learning_rate=0.1, momentum=0.5, epochs=1
00):
        '''
        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 lear
ning
        '''
        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 in
puts 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 p
rovided
            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 arra
ys of activations
        for l in self.layers:
            a.append(np.zeros(l))                                 # One array of zeros pe
r layer

        for k in range(epochs):                                   # Iterate through the e
pochs
            error_it = np.zeros(X.shape[0])                       # Initialize an array t
o store the errors during training (n examples)

            for it in range(X.shape[0]):                          # Iterate through the e
xamples in the training set
                i = np.random.randint(X.shape[0])                 # Select one random exa
mple
                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 t
he activation of each layer
```

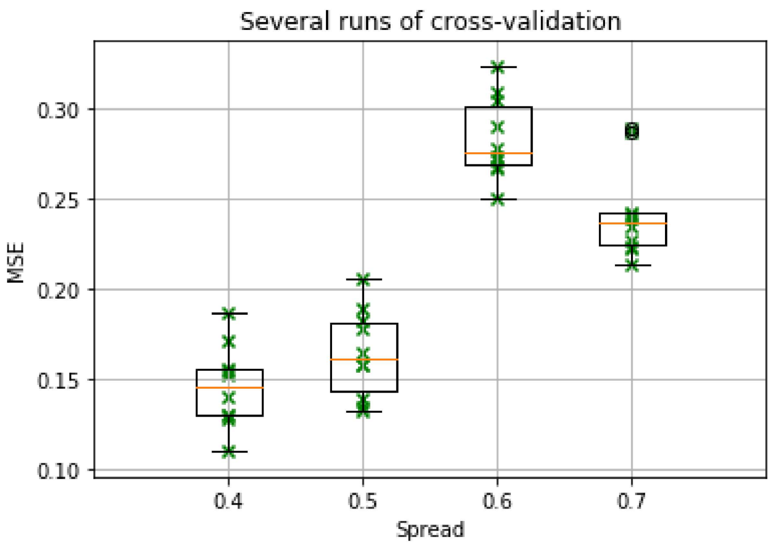
```
                    a[l+1] = self.activation(np.dot(a[l], self.weights[l])) # Apply the
activation function to the product input.weights

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

                                                                # Back-propagation
                                                                # We need to begin at t
he layer previous to the last one (out->in)
                for l in range(len(a) - 2, 0, -1):              # Append a delta for ea
ch layer
                    deltas.append(deltas[-1].dot(self.weights[l].T) * self.activation_d
eriv(a[l]))
                deltas.reverse()                                # Reverse the list (in-
>out)

                                                                # Update
                for i in range(len(self.weights)):              # Iterate through the l
ayers
                    previous_delta_weights = 0
                    layer = np.atleast_2d(a[i])                 # Activation
                    delta = np.atleast_2d(deltas[i])            # Delta
                                                                # Compute the weight ch
ange using the delta for this layer
                                                                # and the change comput
ed for the previous example for this layer
                    delta_weights = -learning_rate * layer.T.dot(delta) + momentum * pr
evious_delta_weights
                    self.weights[i] += delta_weights            # Update the weights
                    previous_delta_weights = delta_weights


            error_train[k] = np.mean(error_it)                  # Compute the average o
f 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 e
rror after iteration k

        if data_test is None:                                   # If only a training da
ta was provided
            return error_train                                  # Return the error duri
ng training
        else:
            return (error_train, error_test)                    # Otherwise, return bot
h training and testing error

    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 array'

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

7_hold_out_validation



8_cross_validation

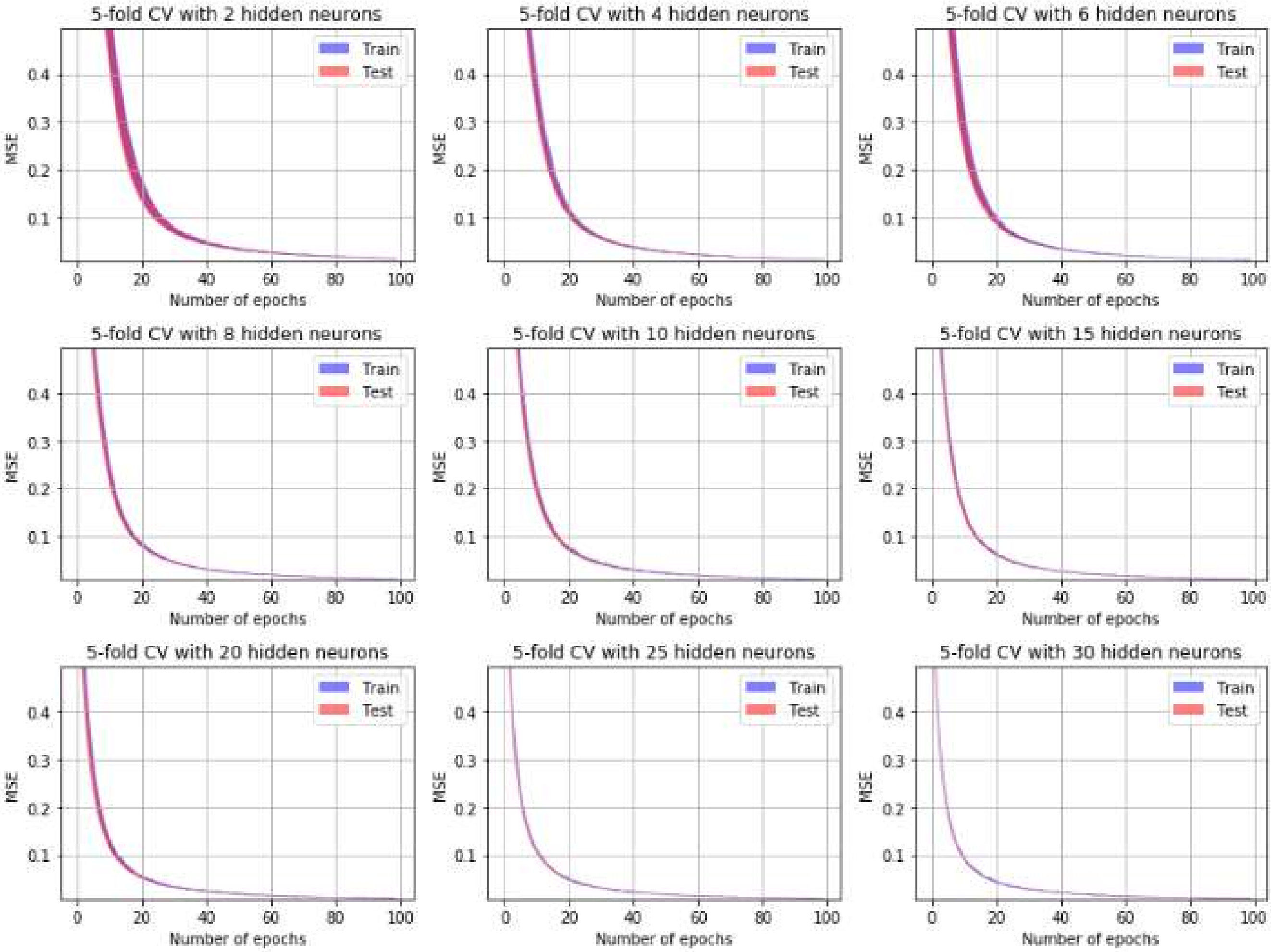


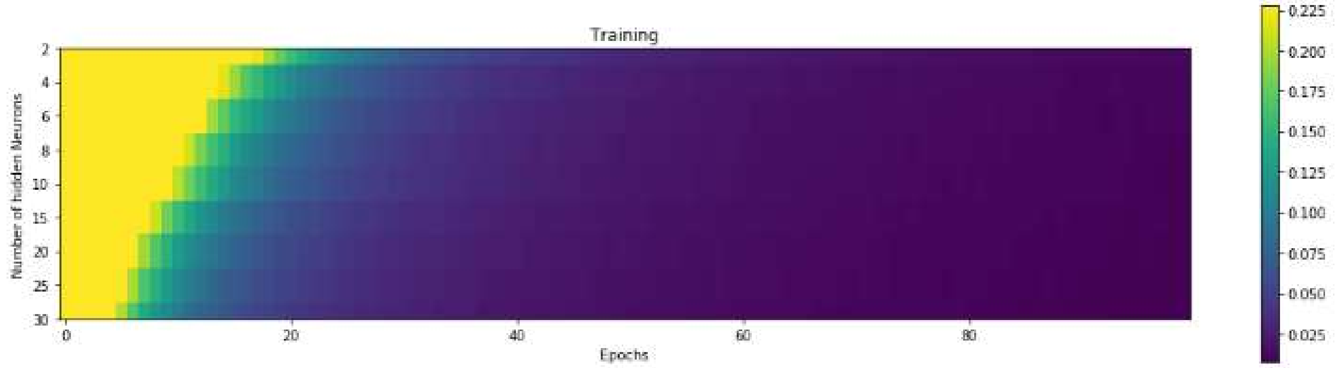
Difference between 7 and 8

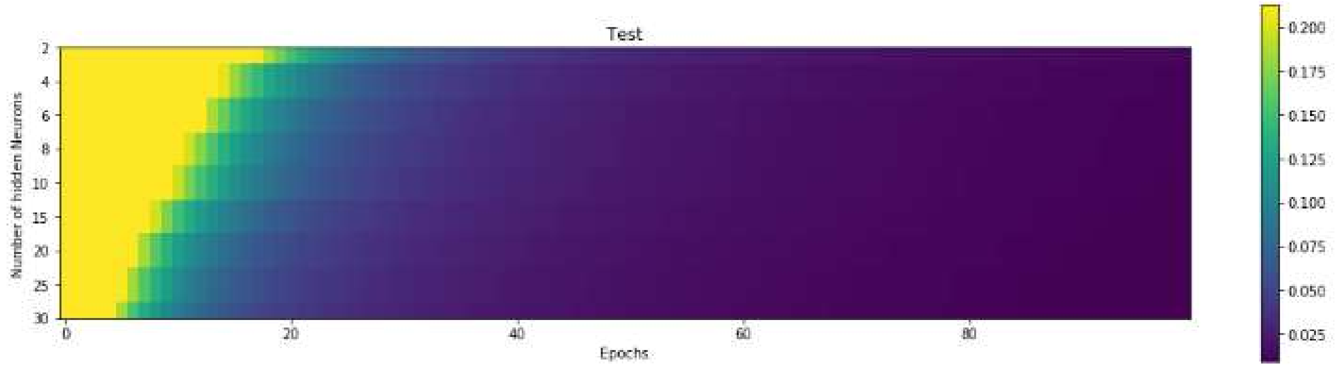
We can see that with the cross validation there are less dispersions than with the hold-out validation. The median error is better for the hold out validation (overfitting).

9_Model_Selection

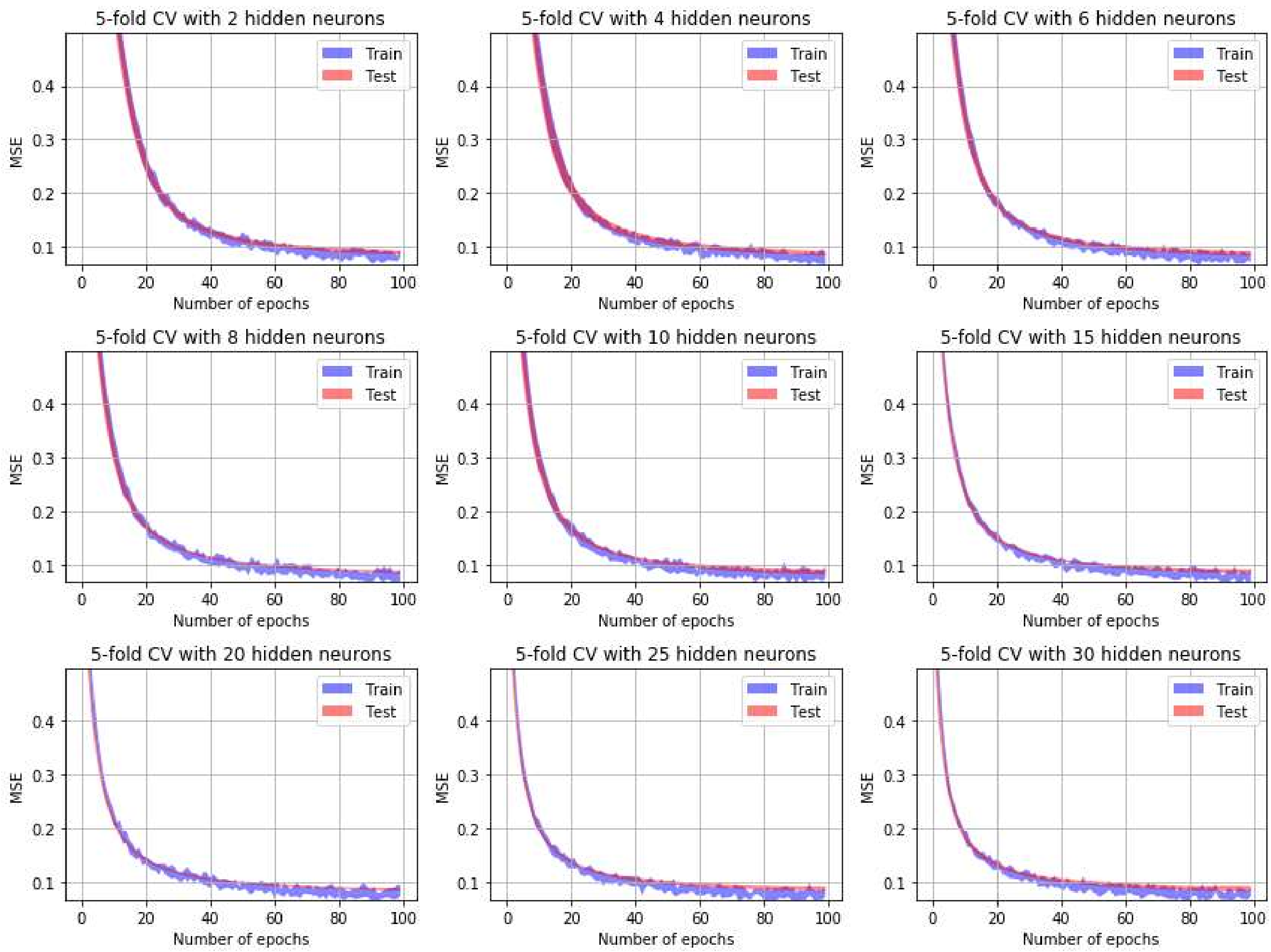
Spread 0.3

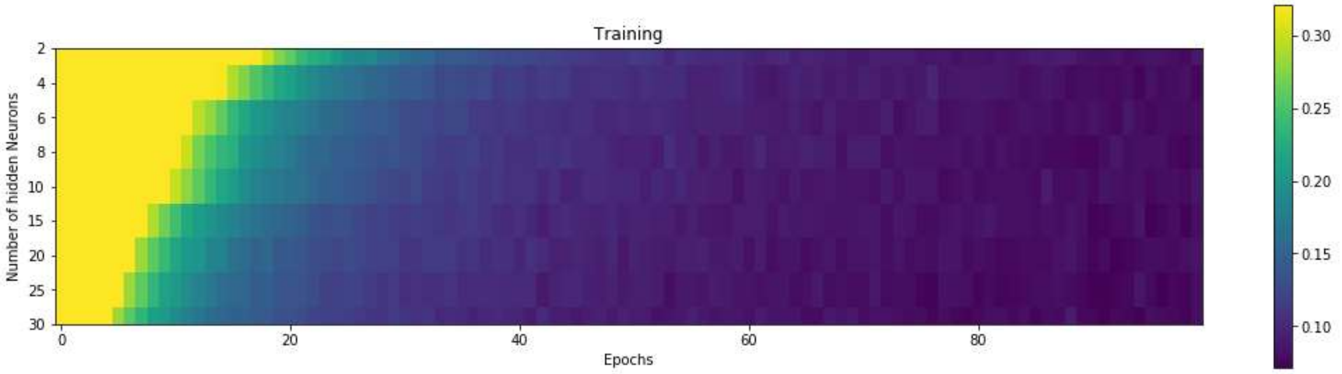


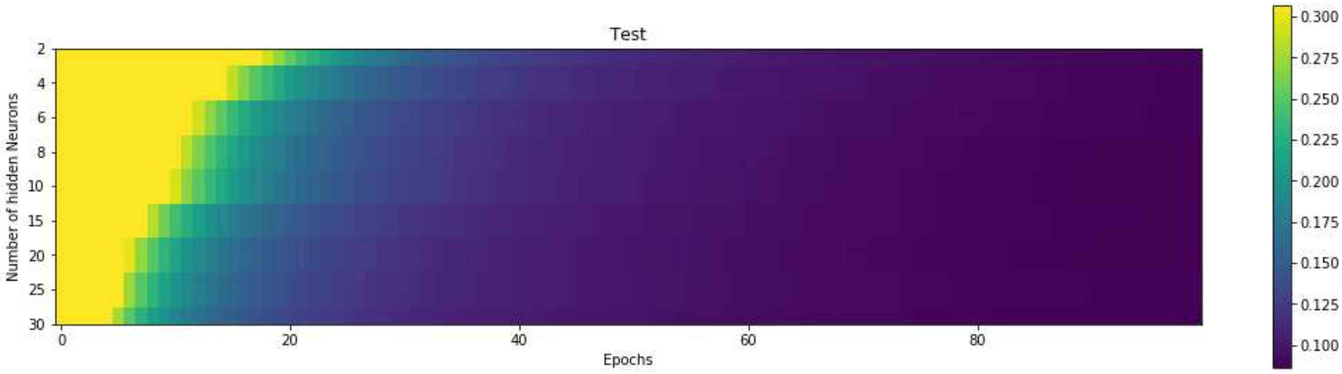

Training
Number of hidden Neurons
Epochs
0 20 40 60 80
2 4 6 8 10 15 20 25 30
0.225 0.200 0.175 0.150 0.125 0.100 0.075 0.050 0.025

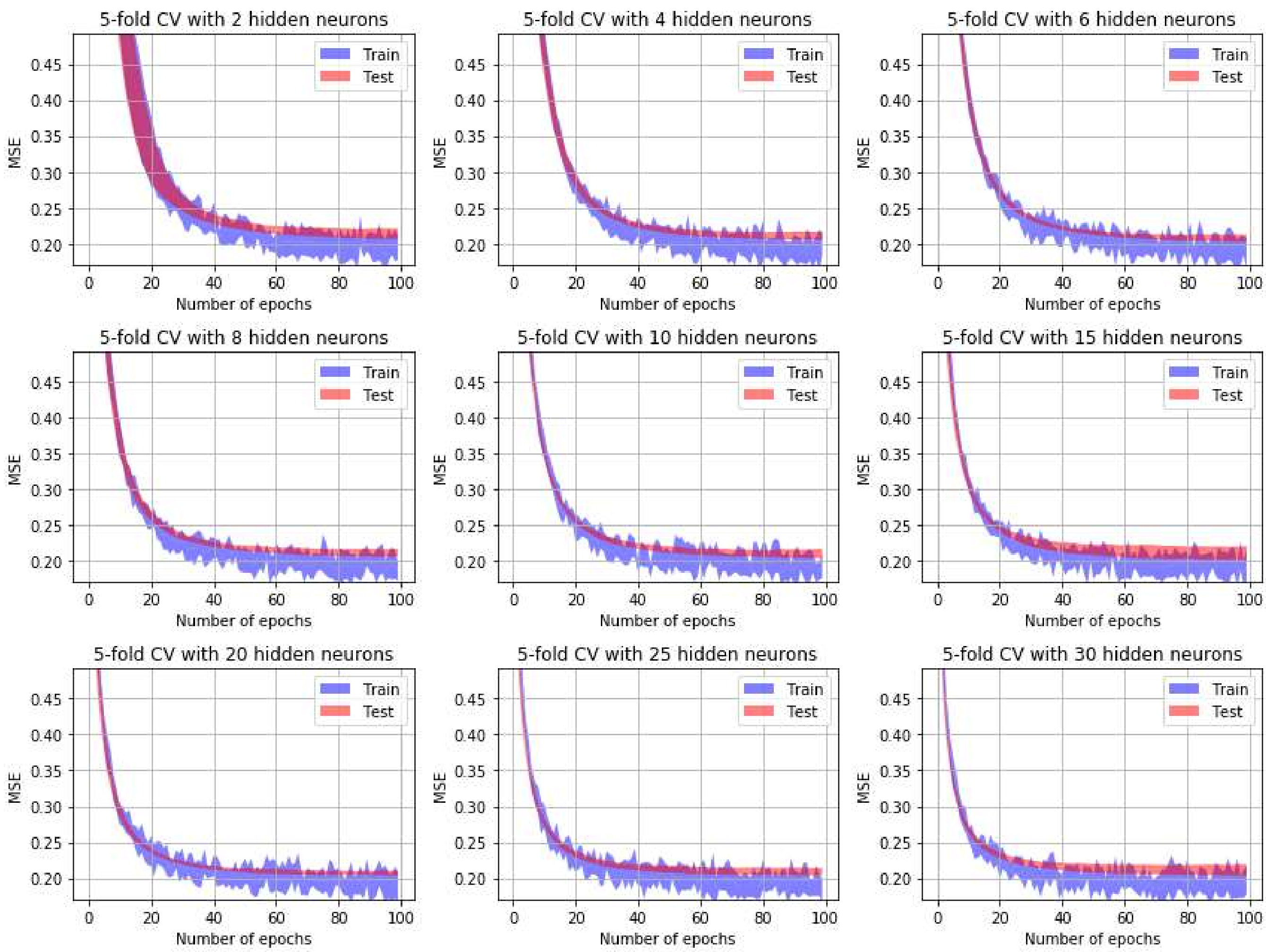

Test
Number of hidden Neurons
Epochs
0 20 40 60 80
2 4 6 8 10 15 20 25 30
0.200 0.175 0.150 0.125 0.100 0.075 0.050 0.025

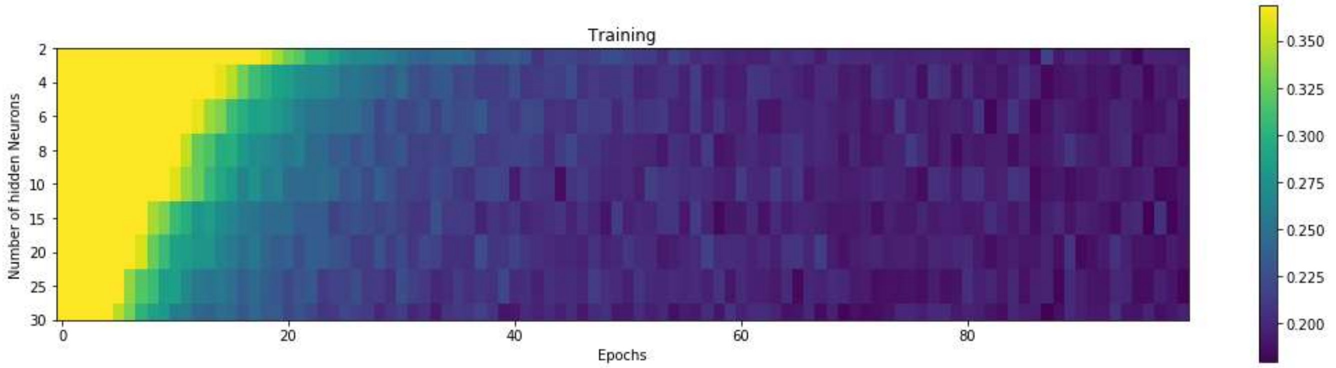
Spread 0.5

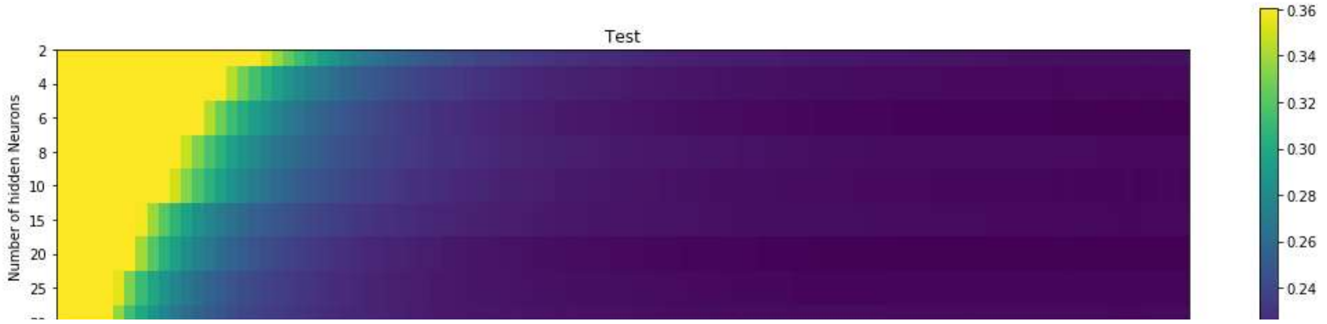

5-fold CV with 2 hidden neurons
5-fold CV with 4 hidden neurons
5-fold CV with 6 hidden neurons
5-fold CV with 8 hidden neurons
5-fold CV with 10 hidden neurons
5-fold CV with 15 hidden neurons
5-fold CV with 20 hidden neurons
5-fold CV with 25 hidden neurons
5-fold CV with 30 hidden neurons
Train
Test
MSE
Number of epochs


Training
Number of hidden Neurons
Epochs


Test
Number of hidden Neurons
Epochs


5-fold CV with 2 hidden neurons
5-fold CV with 4 hidden neurons
5-fold CV with 6 hidden neurons
5-fold CV with 8 hidden neurons
5-fold CV with 10 hidden neurons
5-fold CV with 15 hidden neurons
5-fold CV with 20 hidden neurons
5-fold CV with 25 hidden neurons
5-fold CV with 30 hidden neurons
Train
Test
MSE
Number of epochs





4 hidden neurons is a good tradeoff for convergence time (around 60 epochs) and computation time (since more neurons = more computation time). If we look at the thickness of the red line, 4 neurons seems better (2 neurons results in a thick line for a spread 0.7)

```
In [ ]:
```