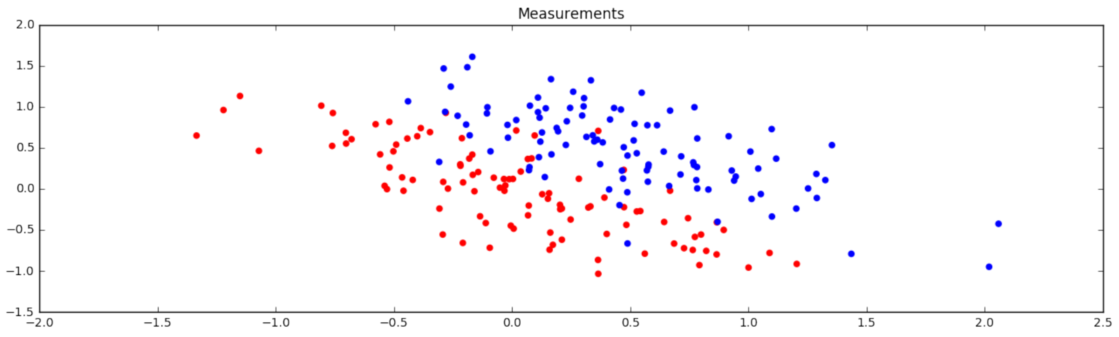
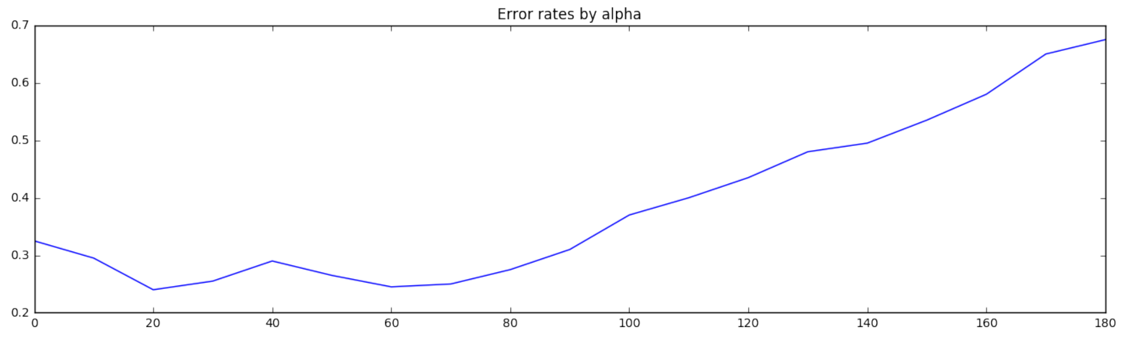
MI1 - H2

November 3, 2016

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.cm as cm
        import itertools
        %matplotlib inline

In [3]: data = np.loadtxt('applesOranges.csv', skiprows=1, dtype=bytes, delimiter='

In [4]: def weight(alpha):
            # Use unit circle definition
            rad = np.radians(alpha)
            return np.array((np.cos(rad), np.sin(rad)))

        def f(x, w, theta):
            return (np.sign(w.dot(x) + theta) + 1) / 2

        def error(yT, x, w, theta):
            # Use linear error measure
            return np.abs(yT - f(x, w, theta))

        def plot(ax, data, **kwargs):
            mapping = np.array(data).T
            ax.plot(mapping[0], mapping[1], **kwargs)
            ax.set_title(kwargs['label'])

        def scatter(ax, data, **kwargs):
            mapping = np.array(data).T
            ax.scatter(mapping[0], mapping[1], **kwargs)
            ax.set_title(kwargs['label'])

In [5]: # Exercise 1.a
        fig, ax = plt.subplots(1, 1, figsize=(13, 4))
        colors = ['red' if x[2]==0 else 'blue' for x in data]
        scatter(ax, data[:, :2], label='Measurements', color=colors)
        fig.tight_layout()
```

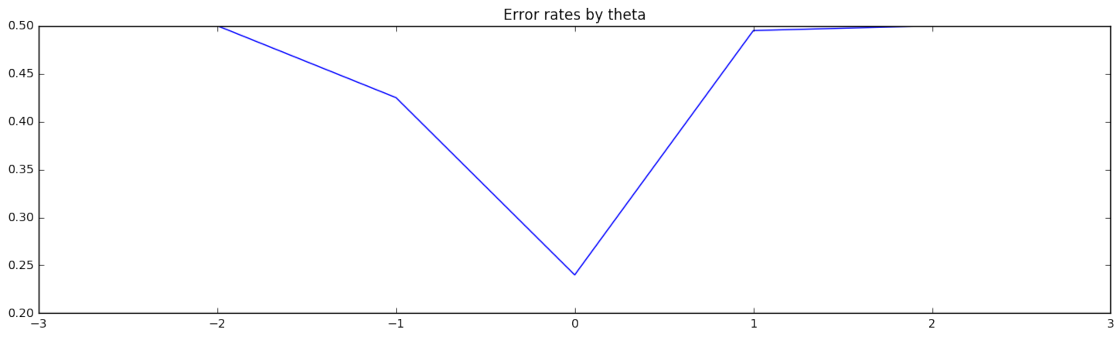


```
In [6]: # Exercise 1.b
        theta = 0
        alphas = range(0, 181, 10)
        errors = []

        for alpha in alphas:
            w = weight(alpha)
            e = [error(yT, (x1, x2), w, theta) for x1, x2, yT in data]
            classification = [f(x[:2], w, 1) for x in data]
            eT = np.average(e)
            errors.append((alpha, eT))

        fig, ax = plt.subplots(1, 1, figsize=(13, 4))
        plot(ax, errors, label='Error rates by alpha')
        fig.tight_layout()
```



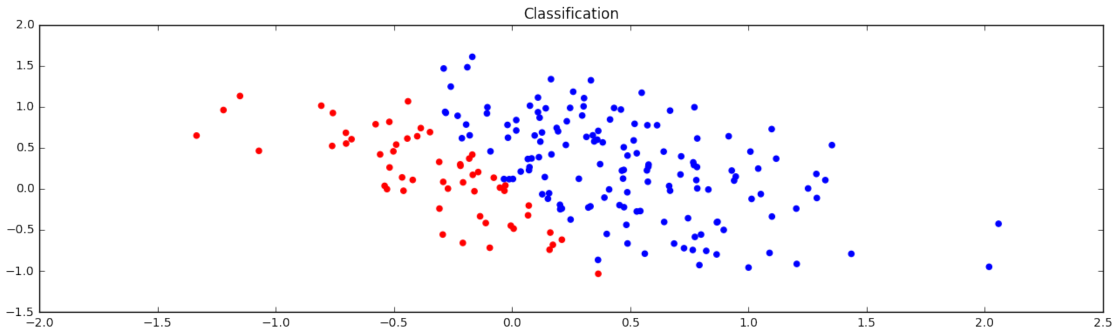
```
In [7]: # Exercise 1.c
        thetas = range(-3, 4)
        alpha = min(errors, key = lambda x: x[1])[0]
        w = weight(alpha)
        errors2 = []
```

```
for theta in thetas:
    e = [error(yT, (x1, x2), w, theta) for x1, x2, yT in data]
    eT = np.average(e)
    errors2.append((theta, eT))

fig, ax = plt.subplots(1, 1, figsize=(13, 4))
plot(ax, errors2, label='Error rates by theta')
fig.tight_layout()
```



```
In [8]: # Exercise 1.d
        theta = min(errors2, key = lambda x: x[1])[0]
        alpha = min(errors, key = lambda x: x[1])[0]
        w = weight(alpha)

        fig, ax = plt.subplots(1, 1, figsize=(13, 4))
        classification = [f(x[:2], w, theta) for x in data]
        eT = np.average([error(yT, (x1, x2), w, theta) for x1, x2, yT in data])
        colors = ['red' if x==0 else 'blue' for x in classification]
        scatter(ax, data[:, :2], label='Classification', color=colors)
        fig.tight_layout()

        print('Parameters: Alpha = {}, Theta = {}, Error rate = {}'.format(alpha, t
```

```
Parameters: Alpha = 20, Theta = 0, Error rate = 0.24
```



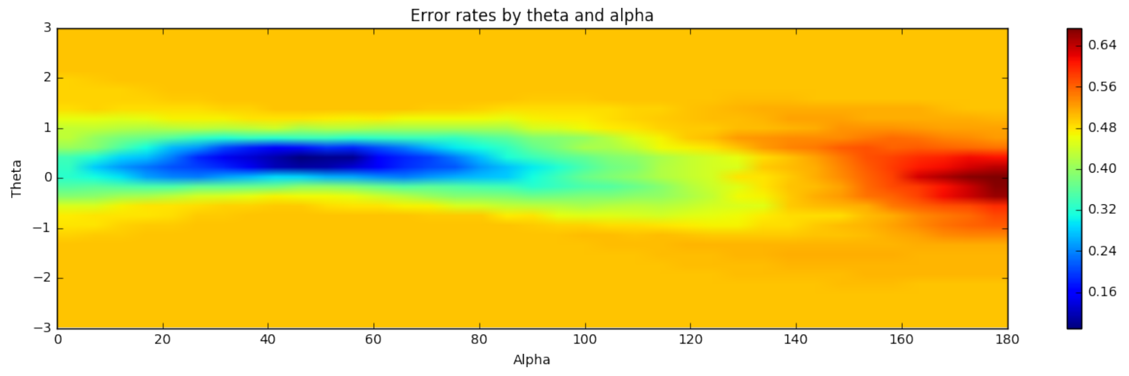
0.1 Interpretation

The weighting vector defines if a datapoint is on the left or the right side of it. In the end the scalar product of the weighting vector on a data point returns a positive or a negative number. By applying the sign function to the return value you get one of two possible values defining the label of the given datapoint.

The optimized parameters alpha and theta define the weight vector such that the resulting seperating line is close to the optimal solution. Because there is only one applied connectionist neuron there is only a linear seperating line which doesn't fit exactly to the given measurements.

```
In [45]: # Exercise 1.e
         thetas = np.arange(-3, 3.1, 0.2)
         alphas = range(0, 181, 5)
         results = []
         min_value = 1
         best_params = None

         for alpha in alphas:
             w = weight(alpha)
             row = []
             for theta in thetas:
                 e = [error(yT, (x1, x2), w, theta) for x1, x2, yT in data]
                 eT = np.average(e)
                 row.append(eT)
                 if eT < min_value:
                     best_params = (alpha, theta)
                     min_value = eT
             results.append(row)


         fig, ax = plt.subplots(1, 1, figsize= (13, 4))
         cax = ax.imshow(np.array(results).T, extent=[0, 180, -3, 3], aspect='auto'
         fig.colorbar(cax)
         ax.set_title('Error rates by theta and alpha')
         ax.set_ylabel('Theta')
         ax.set_xlabel('Alpha')
         fig.tight_layout()

         print('Best parameters: Alpha = {}, Theta = {:.2f}, Error rate = {}'.forma
```

```
Best parameters: Alpha = 45, Theta = -0.40, Error rate = 0.09
```



1 Exercise 1.f

> Can the optimization method (e) be applied to any classification problem? Discuss potential problems and give an application example in which the above method must fail.

Since we're only using a single conntectionist neuron, we are only able to execute a linear classification. The seperating line only is a linear function so we cannot find the right classification for some points which are surrounded by a cloud of points with a different label. If we have a dataset which can be classified by a nonlinear function like a quadratic function we would get even worse results than for the given example, because we cannot reproduce the seperating line with a linear classification.

2 Exercise 2.a

> Describe a simple example in which a multilayer perceptron (MLP) can distinguish between two classes, but a single connectionist neuron can not.

input 1	input 2	sum	output
0	1	1	1
1	0	1	1
1	1	2	0

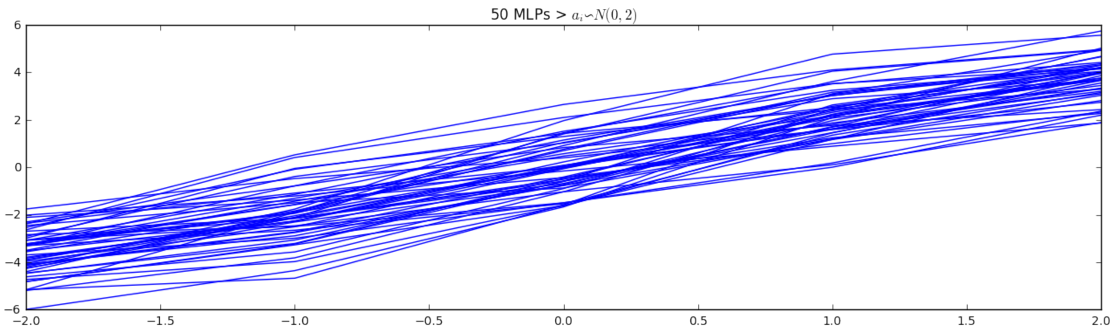
The output values in this example describe a XOR function. There is no definition for the set of weights, theta and the transfer function that can map to the output of this example.

Example of a MLP which mocks the XOR function:

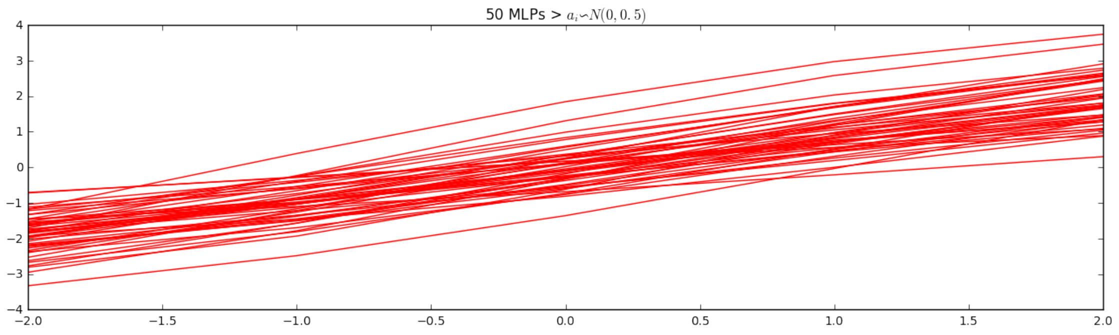
- 2 input units, 1 hidden unit (A), 1 output unit (B)
- f(x)=(sgn(x) + 1)/2
- Theta_A = 1.5
- Theta_B = 0.5

- w_A = (1, 1)
- w_B = (1, -2, 1)

```
In [10]: # Exercise 2.b
         def y(x, w, f, a, b):
             return (w * f(a * (x - b))).sum()

         def generate_mlps(max_a, f=np.tanh, count=50, nodes=10, xs=np.arange(-2, 3
             mlps = [(np.random.random(nodes), np.random.random(nodes) * max_a, np.
             all_ys = [([y(x, w, f, a, b) for x in xs], w, a, b) for w, a, b in mlp
             return [(list(zip(xs, ys)), w, a, b) for ys, w, a, b in all_ys]

         def plot_mlps(values, **kwargs):
             fig, ax = plt.subplots(1, 1, figsize=(13, 4))
             for xs_ys, _, _, _ in values:
                 plot(ax, xs_ys, **kwargs)
             fig.tight_layout()

         functions = generate_mlps(2, np.tanh, 50)
         plot_mlps(functions, label=r'50 MLPs > $a_i \backsim N(0, 2)$', color='blu
```



```
In [43]: # Exercise 2.c
         functions2 = generate_mlps(0.5, np.tanh, 50)
         plot_mlps(functions2, label=r'50 MLPs > $a_i \backsim N(0, 0.5)$', color='
```



3 Comparison of 2.b and 2.c

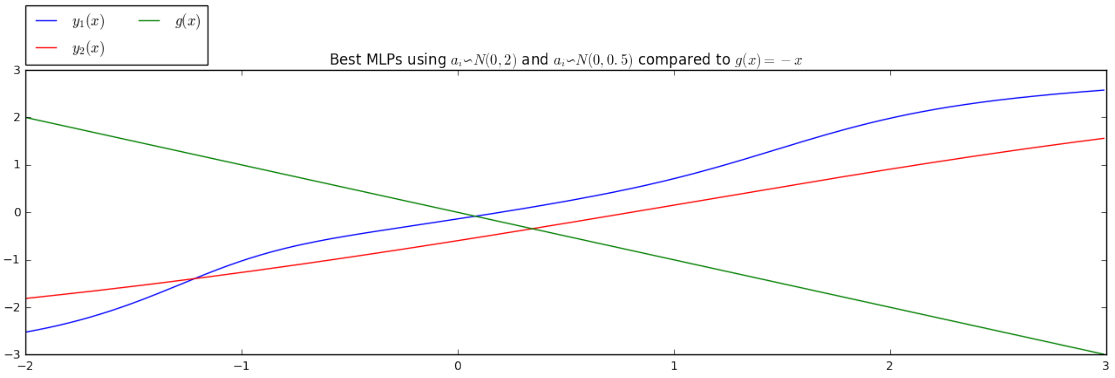
In the first plot the functions are more strongly changing as in the second one. The variance of the plotted values increases with a greater normally distributed range of a_i. In contrast the functions in the second plot are changing more consistently.

Because of a more likely small a_i the hyperbolic tangent gets streched and results in a smoother switching between positive and negative values. That's why the functions in 2.c are looking more like a linear function for the given values of x.

```
In [44]: # Exercise 2.d

         # Input values for evaluation
         xs = np.arange(-2, 3, 0.01)
         g = lambda x: -x
         all_gs = [(x, g(x)) for x in xs]
         all_ys1 = generate_mlps(2, xs=xs)
         squared_errors1 = [(np.average([(xy[1] - xg[1])**2 for xg, xy in zip(all_g
                            for all_ys, w, a, b in all_ys1]
         all_ys2 = generate_mlps(0.5, xs=xs)
         squared_errors2 = [(np.average([(xy[1] - xg[1])**2 for xg, xy in zip(all_g
                            for all_ys, w, a, b in all_ys2]

         best_mlp1 = min(squared_errors1, key=lambda x: x[0])
         best_mlp2 = min(squared_errors2, key=lambda x: x[0])

         fig, ax = plt.subplots(1, 1, figsize=(13, 4))
         plot(ax, best_mlp1[1], label=r'$y_1(x)$', color='blue')
         plot(ax, best_mlp2[1], label=r'$y_2(x)$', color='red')
         plot(ax, all_gs, label=r'$g(x)$', color='green')
         ax.set_title(r'Best MLPs using $a_i \backsim N(0, 2)$ and $a_i \backsim N
         plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3,
                    ncol=2, borderaxespad=0.)
         fig.tight_layout()

         w, a, b = [[float(str(x)[:5]) for x in param] for param in best_mlp1[2:]]
         print('MLP y_1(x):\n\tMSE={:.2f}\n\tw={}\n\ta={}\n\tb={}\n'.format(best_m
         w, a, b = [[float(str(x)[:5]) for x in param] for param in best_mlp2[2:]]
         print('MLP y_2(x):\n\tMSE={:.2f}\n\tw={}\n\ta={}\n\tb={}\n'.format(best_m
```

```
MLP y_1(x):
        MSE=9.22
        w=[0.693, 0.367, 0.087, 0.257, 0.924, 0.447, 0.093, 0.674, 0.39, 0.271]
        a=[1.982, 0.39, 1.078, 1.737, 0.087, 0.149, 1.465, 1.362, 0.007, 0.97]
        b=[-1.24, 1.945, 0.089, -1.76, -0.2, 1.178, 1.877, 1.461, -1.24, 0.142]

MLP y_2(x):
        MSE=6.16
        w=[0.305, 0.265, 0.473, 0.776, 0.057, 0.85, 0.646, 0.18, 0.51, 0.927]
```

```
a=[0.196, 0.258, 0.494, 0.011, 0.104, 0.04, 0.033, 0.079, 0.414, 0.263]
b=[1.317, -0.58, 1.606, -1.24, 0.009, 0.28, -0.11, -1.07, -0.85, 1.959]
```


$y_1(x)$
$y_2(x)$
$g(x)$
Best MLPs using $a_i \backsim N(0,2)$ and $a_i \backsim N(0,0.5)$ compared to $g(x) = -x$