Perceptron algorithm

In this notebook, we will implement the perceptron algorithm for a simple 2d discrimination problem, have a look at the loss function and see the evolution of the solution within the iterations.

In [1]:
```
%matplotlib notebook

import numpy as np
from matplotlib import cm
import matplotlib.pyplot as plt
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from mpl_toolkits.mplot3d import Axes3D
import time
```

In [2]:
```
# Fix seed
np.random.seed(seed=5)
```

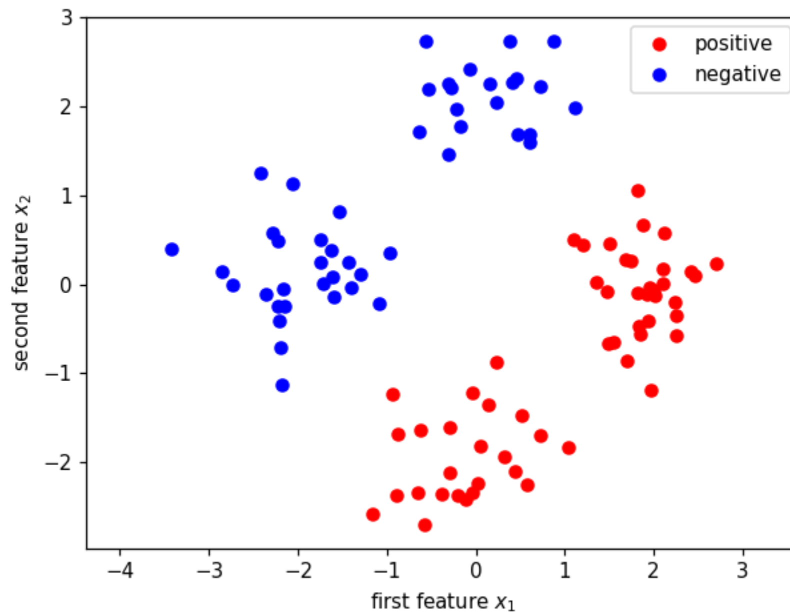
2d data generation of two classes that are likely to be linearly separable

In [3]:
```
# Size of the training set
N = 100

# Generate desired labels
d = np.random.choice([-1, 1], size=N)

# Generate 2d data
idx = d == 1
x = np.zeros((N, 2))
x[~idx, :] = [0, 2]
x[idx, :] = [2, 0]
x[:50, 0] += -2
x[:50, 1] += -2
x = x + np.random.randn(N, 2)/2
```

```
In [4]: # Data visualization
        plt.figure(dpi=72)
        plt.plot(x[idx, 0], x[idx, 1], 'or', label='positive')
        plt.plot(x[~idx, 0], x[~idx, 1], 'ob', label='negative')
        plt.axis('equal')
        plt.xlabel('first feature $x_1$')
        plt.ylabel('second feature $x_2$')
        plt.legend()
        plt.show()
```



Model

In this experiments the data have been generated in a way that a separating hyperplane pass through (0,0), such that we can choose the bias as $b = 0$.

```
In [5]: b = 0
```

The purpose is to find the two synaptic weights w_1 and w_2 that encode an hyplerplane separating the data.

Defining the loss

Given w and a collection of x and d, the perceptron loss can be implemented by making the observation:

$$\sum_{\substack{(x,d) \\ y=<w,x>\neq d}} -d < w, x > = \sum_{(x,d)} \max(-d < w, x >, 0$$

In [6]:
```
loss = lambda w, x, d : np.maximum(-d * x.dot(w), 0).sum()
```

where $w = (w_1, w_2)^T$ is a column vector containing both synaptic weights.

Let us visualize the loss

Since we have only two parameters w_1 and w_2, this is convenient as we can display the evolution of the loss with respect to these two parameters.

To this end, we will consider Z values for w_1 ranging from -10 and 10. We will also consider Z values for w_2 ranging from -10 and 10. To generate all combinations (Z^2) of these two parameter values, we will use the meshgrid function.

In [7]:
```
Z = 30
w1_grid = np.linspace(-10, 10, Z)
w2_grid = np.linspace(-10, 10, Z)
w1_mesh, w2_mesh = np.meshgrid(w1_grid, w2_grid, indexing='ij')
w_mesh = np.stack((w1_mesh, w2_mesh), axis=2)
```

Now we can evaluate the total loss on that grid

In [8]:
```
loss_mesh_total = \
    np.array([[loss(w_mesh[i,j], x, d) \
                   for j in range(Z)] for i in range(Z)])
```

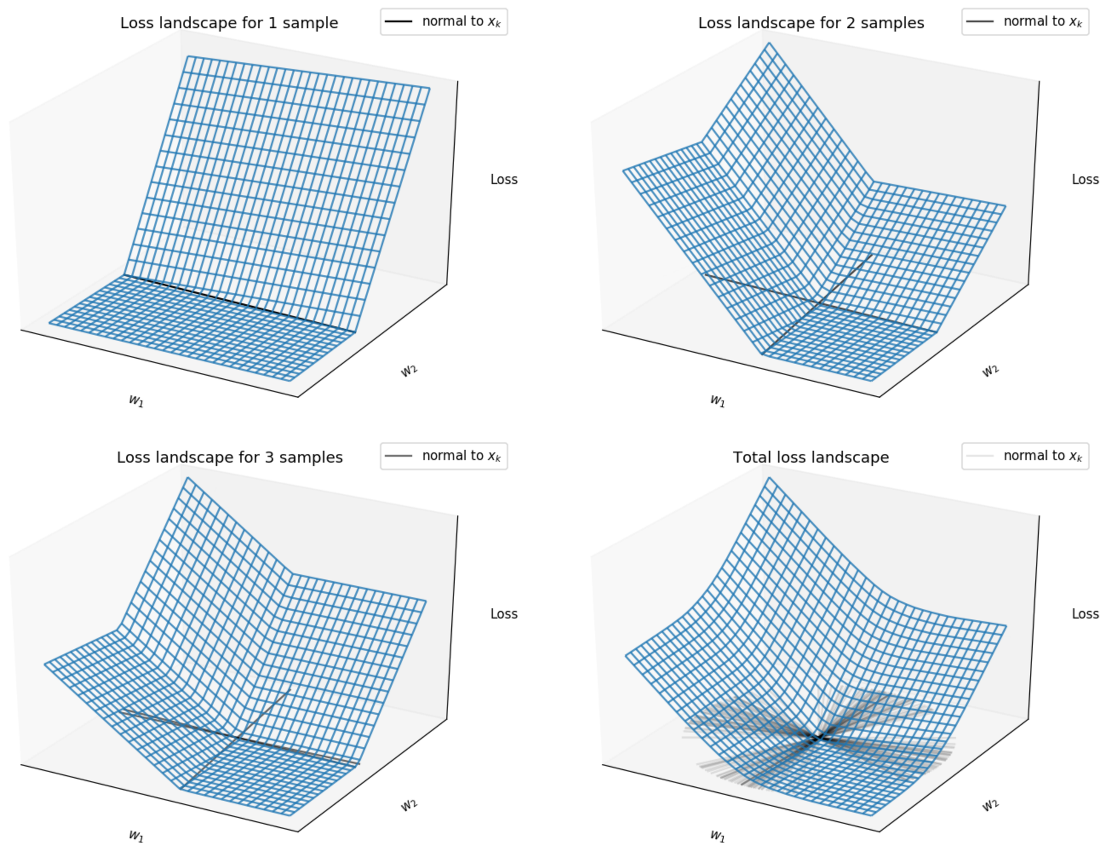
We can also evaluate the loss if we consider $1, 2$ or 3 samples.

In [9]:
```
loss_mesh_subset = \
    [np.array([[loss(w_mesh[i,j], x[:k], d[:k]) \
                   for j in range(Z)] for i in range(Z)]) \
     for k in range(1, 4)]
```

Now we display all of that

In [10]:

```
fig = plt.figure(figsize=np.array([2*6.4, 2*4.8]), dpi=72)
ax = [fig.add_subplot(2, 2, k+1, projection='3d') \
      for k in range(4)]
for k in range(4):
    ax[k].plot_wireframe(w1_mesh, w2_mesh, \
                         loss_mesh_subset[k] if k < 3 else \
                         loss_mesh_total, \
                         alpha=0.8)
    n = np.array([1, 2, 3, N])[k]
    for l in range(n):
        xb = 10 * x[l] / np.sqrt(sum(x[l]**2))
        ax[k].plot([-xb[1], xb[1]], [+xb[0], -xb[0]], \
                   'k-', alpha=np.sqrt(1/n), \
                   label='normal to $x_k$' if l == 0 else None)
ax[0].set_title('Loss landscape for 1 sample')
ax[1].set_title('Loss landscape for 2 samples')
ax[2].set_title('Loss landscape for 3 samples')
ax[3].set_title('Total loss landscape')
for k in range(4):
    ax[k].set_xlabel('$w_1$')
    ax[k].set_ylabel('$w_2$')
    ax[k].set_zlabel('Loss')
    ax[k].set_xticks([], [])
    ax[k].set_yticks([], [])
    ax[k].set_zticks([], [])
    ax[k].legend()
fig.tight_layout()
```



Let's run the algorithm

First of all, we will prepare three subfigures in which we will display the result: loss landscape, loss evolution with the iterations, and the data with the hyperplane.

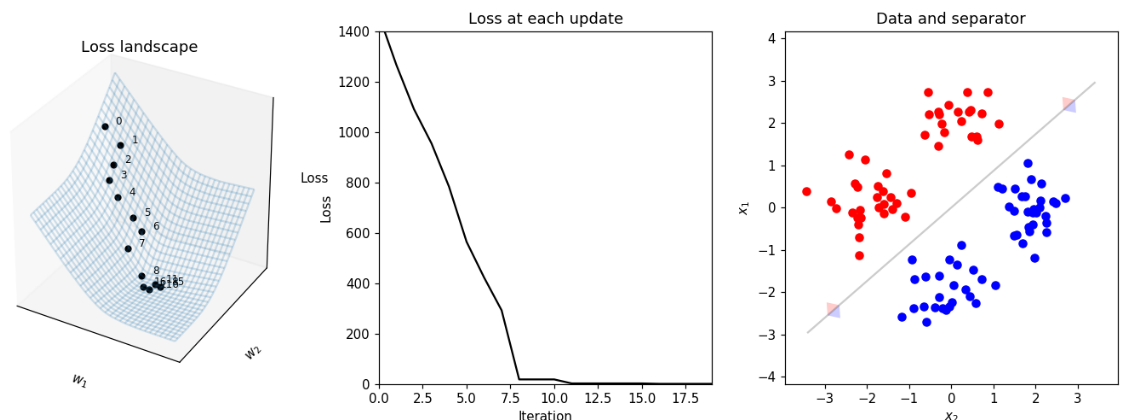
In [11]:
```
fig = plt.figure(figsize=np.array([2*6.4, 4.8]), dpi=72)
ax1 = fig.add_subplot(1, 3, 1, projection='3d')
ax2 = fig.add_subplot(1, 3, 2)
ax3 = fig.add_subplot(1, 3, 3)

ax1.set_title('Loss landscape')
ax1.plot_wireframe(w1_mesh, w2_mesh, \
                   loss_mesh_total, alpha=0.2)
ax1.set_xlabel('$w_1$')
ax1.set_ylabel('$w_2$')
ax1.set_zlabel('Loss')
ax1.set_xticks([], [])
ax1.set_yticks([], [])
ax1.set_zticks([], [])

ax2.set_title('Loss at each update')
ax2.set_xlim(0, 19)
ax2.set_ylim(0, 1400)
ax2.set_xlabel('Iteration')
ax2.set_ylabel('Loss')

ax3.set_title('Data and separator')
ax3.plot(x[idx, 0], x[idx, 1], 'ob')
ax3.plot(x[~idx, 0], x[~idx, 1], 'or')
ax3.axis('equal')
ax3.set_xlim(-5, 5)
ax3.set_ylim(-5, 5)
ax3.set_xlabel('$x_2$')
ax3.set_ylabel('$x_1$')

fig.tight_layout()
```



Now we can implement the algorithm that will update the three above subfigures.

In [12]:

```
# Initialization
w = np.array([-9, 7])

# Number of epochs
T = 2

loss_t = np.zeros(N * T)
for t in range(T):
    for i in range(N):
        # iteration index
        k = t * N + i

        # Compute prediction
        y = np.sign(x[i].dot(w))
        if y != d[i]:
            # Update synaptic weights
            w = w + d[i] * x[i]

        # Store current loss
        loss_t[k] = loss(w, x, d)

        # Update display only the first time or when changed occur
        if y != d[i] or not ax3.patches:

            # Delete previous drawing if any
            if ax3.patches:
                del ax2.lines[-1]
                del ax3.lines[-1]
                del ax3.patches[:]

            # Draw the current solution on the landscape
            ax1.scatter(w[0], w[1], loss_t[k], \
                        color='k', marker='o')
            ax1.text(w[0]+1, w[1]+1, loss_t[k], \
                     '%d' % (t*N+i), \
                     size=8, zorder=1, color='k')

            # Draw the separator
            wb = 4.5 / np.sqrt(np.sum(w**2)) * w
            h = ax3.plot(np.array([-1, 1]) * -wb[1], \
                         np.array([-1, 1]) * wb[0], \
                         'k-', alpha=.2)

            # Draw triangles to know which side is which
            wb = 3.7 / np.sqrt(np.sum(w**2)) * w;
            ax3.fill(-wb[1] + np.array([-wb[1] / 20, +wb[1] / 20, \
                                        +wb[0] / 15]),
                     +wb[0] + np.array([+wb[0] / 20, -wb[0] / 20, \
                                        +wb[1] / 15]),
                     'b', alpha=.2)
            ax3.fill(+wb[1] + np.array([+wb[1] / 20, -wb[1] / 20, \
                                        -wb[0] / 15]),
                     -wb[0] + np.array([-wb[0] / 20, +wb[0] / 20, \
                                        -wb[1] / 15]),
                     'r', alpha=.2)
            ax3.fill(-wb[1] + np.array([-wb[1] / 20, +wb[1] / 20, \
```

```
                                  -wb[0] / 15]),
               +wb[0] + np.array([+wb[0] / 20, -wb[0] / 20, \
                                  -wb[1] / 15]),
               'r', alpha=.2)
    ax3.fill(+wb[1] + np.array([+wb[1] / 20, -wb[1] / 20, \
                                  +wb[0] / 15]),
               -wb[0] + np.array([-wb[0] / 20, +wb[0] / 20, \
                                  +wb[1] / 15]),
               'b', alpha=.2)

    # Draw loss
    ax2.plot(loss_t[:(k+1)], 'k')

    # Force display and wait
    fig.canvas.draw()
    fig.canvas.flush_events()
    time.sleep(1)
```