```python
import numpy as np
import matplotlib.pyplot as plt
nbr_iterations = 5000

def plot_function():
    x0 = np.linspace(-2, 2, 400)
    x1 = np.linspace(-1, 3, 400)
    X0, X1 = np.meshgrid(x0, x1)
    Z = np.zeros_like(X0)

    for i in range(X0.shape[0]):
        for j in range(X0.shape[1]):
            Z[i, j] = function(np.array([X0[i, j], X1[i, j]]))

    plt.figure(figsize=(8, 6))
    cp = plt.contourf(X0, X1, Z, levels=50, cmap='viridis')
    plt.colorbar(cp)
    plt.title('Contour plot of the function')
    plt.xlabel('x0')
    plt.ylabel('x1')
    plt.show()

def function(x: np.ndarray) -> float:
    return 200 * ((x[1] - (x[0]**2)) ** 2) + ((1 - x[0])**2)

def gradient(x: np.ndarray) -> np.ndarray:
    return np.array([
        -800 * (x[1] - x[0]**2) * x[0] - 2 * (1 - x[0]),
        400 * (x[1] - x[0]**2)
    ])

def hessian(x: np.ndarray, inverse=False, regularization=1e-8) -> np.ndarray:
    hessian = np.array([
        [1600 * x[0]**2 - 800 * (x[1] - (x[0]**2)) + 2, -800 * x[0]],
        [-800 * x[0], 400]
    ])
    if inverse:
        # hessian += np.eye(2) * regularization  # Add regularization term
        return np.linalg.inv(hessian)
    return hessian

def newtons(x: np.ndarray, stopping_criteria=1e-5, iterations:int=nbr_iterations) -> np.ndarray:
    x_values = [np.linalg.norm([1,1] - x)]
    y_values = [function(x)]
    for i in range(iterations):
        x -= np.dot(hessian(x, inverse=True), gradient(x))
        x_values.append(np.linalg.norm([1,1] - x))
        y_values.append(function(x))
    print(x)
    return x_values, y_values

def armijo_condition(x:np.ndarray, lr:float, sigma) -> bool:
    left = function(x + lr*gradient(x))
    right = function(x) - sigma * lr * np.linalg.norm(gradient(x))**2
    return left <= right

def backtracking(x: np.ndarray, stopping_criteria=1e-10, iterations:int=nbr_iterations, sigma=1e-3, beta=0.9, lr=1) -> np.ndarray:
    x_values = [np.linalg.norm([1,1] - x)]
    y_values = [function(x)]
    for i in range(iterations):
        learning_rate = lr
        while not armijo_condition(x,lr=learning_rate, sigma=sigma):
            learning_rate *= beta
        x -= lr * sigma * gradient(x)
        x_values.append(np.linalg.norm([1,1] - x))
        y_values.append(function(x))
    print(f'x: {x}, f(x): {function(x)}')
    return x_values, y_values

def gradient_descent(x: np.ndarray, stopping_criteria=1e-10, iterations:int=nbr_iterations, lr=1e-3) -> np.ndarray:
    x_values = [np.linalg.norm([1,1] - x)]
    y_values = [function(x)]
    for i in range(iterations):
        x -= lr * gradient(x)
        x_values.append(np.linalg.norm([1,1] - x))
        y_values.append(function(x))
    print(f'x: {x}, f(x): {function(x)}')

    return x_values, y_values

iterations = [i for i in range(1, nbr_iterations+2)]

x_new, y_new = newtons(np.array([0.5, 0.5],dtype=float))
x_gd, y_gd = gradient_descent(np.array([0.5, 0.5],dtype=float))
x_back, y_back = backtracking(np.array([0.5, 0.5],dtype=float))

plt.plot(iterations, x_new, label='Newton\'s', c='r')
plt.plot(iterations, x_gd, label='Gradient Descent', c='b')
plt.plot(iterations, x_back, label='Backtracking', c='g')
plt.title("|| x_i - x_* ||")
plt.xlabel("iterations")
plt.ylabel("dist from optimal x_*")
plt.show()

plt.plot(iterations, y_new, label='Newton\'s', c='r')
plt.plot(iterations, y_gd, label='Gradient Descent', c='b')
plt.plot(iterations, y_back, label='Backtracking', c='g')
plt.title("|| f(x_i) ||")
plt.xlabel("iterations")
plt.ylabel("|| f(x_i) ||")
plt.show()
```