Supervised Learning

Michael Untereiner

System of Partial Differential Equations for Error Backpropagation:

Let E be the error function, w be the weight vector, x be the input vector, y be the output.

$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial w}$

$\frac{\partial E}{\partial y} = t - y(x)$

$\frac{\partial y}{\partial z} = y(1 - y)$

$\frac{\partial z}{\partial w} = x$

$y = f(z) = \frac{1}{1+e^{-z}}$

$z(x) = xz + b$

By following the slope of the error function along its steepest descent, we optimize the hidden edge weights to minimize the error function across all inputs. Hopefully, we fall into a local minimum which approximates the global minimum error.

Implementation of Gradient Descent Algorithm:

```
import numpy as np

# inputs
X = np.array(input)

# outputs
# x.T is the transpose of x, making this a column vector
y = np.array([output]).T

# define the sigmoid function
def sigmoid(x, derivative=False):
    if (derivative == True):
        return x * (1 - x)
    else:
        return 1 / (1 + np.exp(-x))

# choose a random seed for reproducible results
np.random.seed(1)

# learning rate
alpha = 0.1

# number of nodes in the hidden layer
num_hidden = pow(n, 2)

# initialize weights randomly with mean 0 and range [-1, 1]
# the +1 in the 1st dimension of the weight matrices is for the bias weight
hidden_weights = 2*np.random.random((X.shape[1] + 1, num_hidden)) - 1
output_weights = 2*np.random.random((num_hidden + 1, y.shape[1])) - 1

# number of iterations of gradient descent
num_iterations = 10000

# for each iteration of gradient descent
for i in range(num_iterations):

    # forward phase
    # np.hstack((np.ones(...), X) adds a fixed input of 1 for the bias weight
    input_layer_outputs = np.hstack((np.ones((X.shape[0], 1)), X))
    hidden_layer_outputs = np.hstack((np.ones((X.shape[0], 1)),
                                      sigmoid(np.dot(input_layer_outputs, hidden_weights))))
    output_layer_outputs = np.dot(hidden_layer_outputs, output_weights)

    # backward phase
    # output layer error term
    output_error = output_layer_outputs - y
    # hidden layer error term
    # [:, 1:] removes the bias term from the backpropagation
    hidden_error = hidden_layer_outputs[:, 1:] * (1 - hidden_layer_outputs[:, 1:]) *
                   np.dot(output_error, output_weights.T[:, 1:])

    # partial derivatives
    hidden_pd = input_layer_outputs[:, :, np.newaxis] * hidden_error[: , np.newaxis, :]
    output_pd = hidden_layer_outputs[:, :, np.newaxis] * output_error[:, np.newaxis, :]

    # average for total gradients
    total_hidden_gradient = np.average(hidden_pd, axis=0)
    total_output_gradient = np.average(output_pd, axis=0)

    # update weights
    hidden_weights += - alpha * total_hidden_gradient
    output_weights += - alpha * total_output_gradient
```

The matrix X is the set of row vector inputs $\xrightarrow{x}$ and the coloumn vector $\xrightarrow{y}$ contains the corresponding correct outputs y for each input $\xrightarrow{x}$. The size of the hidden layer can be modified with the variable num_hidden and the learning rate can be adjusted with alpha. The number of iterations along the gradient is determined by num_iterations.

Now let's introduce the problem we want to solve with a neural network, the n-Queens problem.

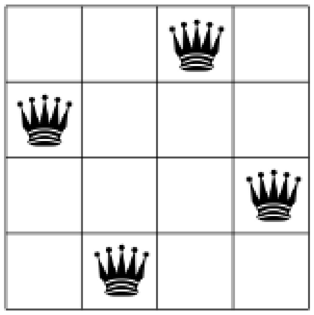
Given an n x n chess board, place n queens on the board such that no two queens threaten each other. That is, place all n queens such that no two queens appear on the same horizontal, vertical, or diagonal on the board.

n-Queens recursive solution:

```
#n queens solution finder
x = {}
n = 4

def place(k, i):
    if (i in x.values()):
        return False
    j = 1
    while(j < k):
        if abs(x[j]-i) == abs(j-k):
            return False
        j+=1
    return True

def clear_future_blocks(k):
    for i in range(k,n+1):
       x[i]=None

def NQueens(y, k):
    for i in range(1, n + 1):
        clear_future_blocks(k)
        if place(k, i):
            x[k] = i
            if (k==n):
                X = []
                for j in x:
                    X += [x[j]]
                y += [X]
            else:
                NQueens(y, k+1)
```

n-Queens is a computationally intensive constraint satisfaction problem. The only known solution requires an exhaustive state space search and becomes intractable for n > 8. The n-Queens constraint satisfaction problem can be thought of as a high-dimensional linear transformation, i.e. a logical function which should be decidable by a neural network with a succifiently large number of hidden nodes.

A solution to 4-queens:



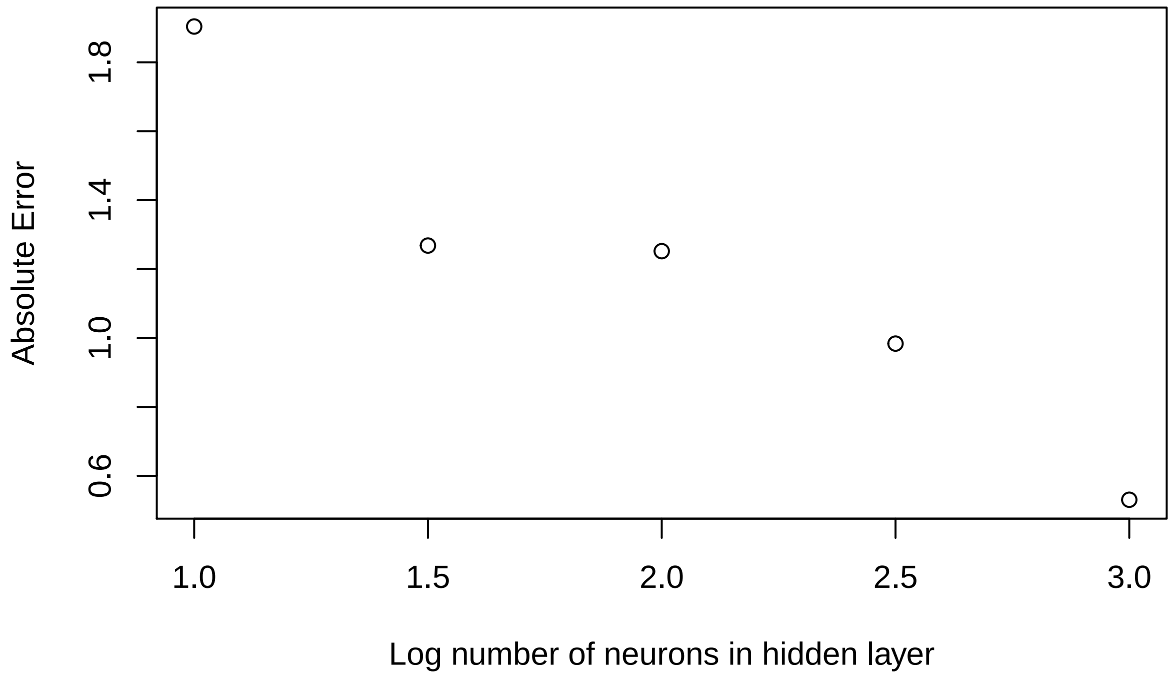
Now let's create our input matrix, consisting of all possible 4x4 board states with 4 queens placed on the board. The input vectors have 16 dimensions, each representng a position on the board. The dimensions take on a value of 1 if a queen is present on that position, and 0 otherwise. We also create the output solution vector, which contains 1 if a particular board state is a solution to n-Queens and 0 otherwise.

Training data:

```
# transition function between board states
def step(l):
    for i in range(0, n):
        if (l[i] < n):
            l[i] += 1
            return l
        else:
            l[i] = 1

# exhaustively generate all board states
def generator():
    start = []
    end = []
    l = []
    for i in range(0, n):
        start += [1]
        end += [n]
    l = start
    result = []
    while (not (l == end)):
        result += [l]
        temp = step(l)
        l = []
        l += temp
        #l = step(l)
    return result

# store input matrix in X and output vector in y
def toMatrix(X, y):
    sol = []
    NQueens(sol, 1)
    g = generator()
    for i in g:
        l = []
        for j in range(0, n):
            queenPos = i[j]
            for k in range(1, n + 1):
                if (queenPos == k):
                    l += [1]
                else:
                    l += [0]
        X += [l]
        validSol = False
        for j in sol:
            if (i == j):
                validSol = True
        if (validSol):
            y += [1]
        else:
            y += [0]

input = [], output = []
```

```
toMatrix(input, output)
```

Now we are ready to run our gradient descent algorithm on the n-Queens problem and compare the error of the neural network as a function of the dimension of the hidden layer. For a given dimension d, the hidden layer in the neural network contained n^d nodes, where n is the chess board side length.



As shown in the graph above, the error decreases linearly with the dimension of the neural network. This suggests that the neural network is able to train much higher-dimensional transformations in the board state space, closely approximating the n-Queens solver at a dimension of $dim = n^3$. Since the dimension of the board state space is n^2, this hidden layer has a higher dimension than the input space, and the large gain in performance is likely due to the ability of this higher dimensionality to represent the super-symmetry of the n-Queens solutions, projecting the higher-order patterns onto the binary output.