Application of Convex Optimization in Neural Networks: Unconstrained Optimization

Ankesh Kumar

August 2, 2024

1 Introduction

Convex optimization plays a crucial role in training neural networks. This document aims to provide a mathematical interpretation of how these techniques are applied, with a specific focus on the XOR problem and various optimization methods.

2 Neural Networks as Convex Optimization Problems

Neural network training can be formulated as an optimization problem. While the overall problem is non-convex due to the nonlinear activation functions, it can be approximated as a convex problem in local regions.

2.1 Mathematical Formulation

For a neural network with parameters $\mathbf{w}$, input $\mathbf{x}$, and target output $\mathbf{y}$, we aim to minimize a loss function $L(\mathbf{w})$:

$$\min_{\mathbf{w}} L(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} l(\mathbf{y}_i, f(\mathbf{x}_i; \mathbf{w})) \tag{1}$$

where $f(\mathbf{x}; \mathbf{w})$ is the neural network function and $l(\cdot, \cdot)$ is a loss metric (e.g., mean squared error).

In a small neighborhood around a point $\mathbf{w}_0$, we can approximate the loss function using Taylor expansion:

$$L(\mathbf{w}) \approx L(\mathbf{w}_0) + \nabla L(\mathbf{w}_0)^T (\mathbf{w} - \mathbf{w}_0) + \frac{1}{2} (\mathbf{w} - \mathbf{w}_0)^T H(\mathbf{w}_0) (\mathbf{w} - \mathbf{w}_0) \tag{2}$$

where $H(\mathbf{w}_0)$ is the Hessian matrix at $\mathbf{w}_0$. If $H(\mathbf{w}_0)$ is positive definite, this local region is convex.

3 XOR Problem Implementation

3.1 Problem Definition

The XOR (exclusive OR) problem is a classic example used to demonstrate the capabilities of neural networks, particularly their ability to solve non-linearly separable problems. The XOR function is defined as follows:

x_1	x_2	$\text{XOR}(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	0

3.2 Neural Network Architecture

To solve the XOR problem, we use a simple neural network with one hidden layer consisting of:

- Two input neurons (for x_1 and x_2)
- Two hidden neurons
- One output neuron

3.3 Optimization Method

The XOR problem is solved using Stochastic Gradient Descent (SGD) with Backpropagation. This method updates the weights after each training example using the gradient of the error, computed via backpropagation. The learning rate η (eta in the code) is set to 10.0, which determines the step size of each weight update.

The weight update rule follows the form:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \nabla L(\mathbf{w}_k) \tag{3}$$

where $\nabla L(\mathbf{w}_k)$ is computed using backpropagation for each layer of the network.

3.4 Implementation Code

Here's the MATLAB code used to implement and train the neural network for the XOR problem:

```
% Initialization
eta = 10.0; % Step size
epochs = 500; % Number of epochs for training

% Initial weights and shift parameters
w11 = 0.1; w12 = -0.2; w21 = -0.3; w22 = 0.4;
w31 = 0.5; w32 = -0.6;
theta1 = 0.7; theta2 = -0.8; theta3 = 0.9;

% Training data for XOR problem
X = [0 0; 0 1; 1 0; 1 1];
y = [0; 1; 1; 0];

% Sigmoid activation function and its derivative
sigmoid = @(v) 1 / (1 + exp(-v));
sigmoid_derivative = @(v) sigmoid(v) * (1 - sigmoid(v));

% Track error over epochs
errors = zeros(epochs, 1);

% Training process
for epoch = 1:epochs
    total_error = 0;
    for i = 1:4
        % Forward propagation
        x_d = X(i, :)';
        y_d = y(i);
        v1 = w11 * x_d(1) + w12 * x_d(2) - theta1;
        v2 = w21 * x_d(1) + w22 * x_d(2) - theta2;
        h1 = sigmoid(v1);
        h2 = sigmoid(v2);
        v3 = w31 * h1 + w32 * h2 - theta3;
        y_hat = sigmoid(v3);

        % Compute error terms
        delta3 = (y_d - y_hat) * sigmoid_derivative(v3);
        delta1 = delta3 * w31 * sigmoid_derivative(v1);
        delta2 = delta3 * w32 * sigmoid_derivative(v2);

        % Update weights
        w31 = w31 + eta * delta3 * h1;
        w32 = w32 + eta * delta3 * h2;
        theta3 = theta3 - eta * delta3;
        w11 = w11 + eta * delta1 * x_d(1);
```

```
        w12 = w12 + eta * delta1 * x_d(2);
        w21 = w21 + eta * delta2 * x_d(1);
        w22 = w22 + eta * delta2 * x_d(2);
        theta1 = theta1 - eta * delta1;
        theta2 = theta2 - eta * delta2;

        % Accumulate total error
        total_error = total_error + 0.5 * (y_d - y_hat)^2;
    end
    errors(epoch) = total_error;
end

% Output results
fprintf('Final weights and shift parameters:\n');
fprintf('w11 = %.4f, w12 = %.4f\n', w11, w12);
fprintf('w21 = %.4f, w22 = %.4f\n', w21, w22);
fprintf('w31 = %.4f, w32 = %.4f\n', w31, w32);
fprintf('theta1 = %.4f, theta2 = %.4f, theta3 = %.4f\n', theta1, theta2, theta3);

% Test the trained network
for i = 1:4
    x_d = X(i, :)';
    v1 = w11 * x_d(1) + w12 * x_d(2) - theta1;
    v2 = w21 * x_d(1) + w22 * x_d(2) - theta2;
    h1 = sigmoid(v1);
    h2 = sigmoid(v2);
    v3 = w31 * h1 + w32 * h2 - theta3;
    y_hat = sigmoid(v3);
    fprintf('Input: [%d, %d], Output: %.4f, Target: %d\n', x_d(1), x_d(2), y_hat, y(i));
end

% Plot error convergence
figure;
plot(1:epochs, errors);
xlabel('Epoch');
ylabel('Total Error');
title('Error Convergence Over Epochs');
```

3.5 Results and Convergence

After running the code, we obtained the following results:

```
Final weights and shift parameters:
w11 = -5.2114, w12 = -5.2418
w21 = -6.7971, w22 = -7.1834
w31 = 9.1886, w32 = -9.4581
theta1 = -7.7544, theta2 = -2.7793, theta3 = 4.3581

Input: [0, 0], Output: 0.0166, Target: 0
Input: [0, 1], Output: 0.9825, Target: 1
Input: [1, 0], Output: 0.9819, Target: 1
Input: [1, 1], Output: 0.0223, Target: 0
```

The results show that the trained network successfully approximates the XOR function. The error convergence plot as in Figure 1. demonstrates how the total error decreases over the training epochs, indicating that the network is learning the XOR function. Figure 2. shows a plot of the function implemented after training this network.

4 Comparison of Optimization Methods

4.1 Gradient Descent

Gradient descent updates the weights in the direction of steepest descent of the loss function.

Mathematical Formulation: For a loss function $L(\mathbf{w})$, the weight update rule is:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \nabla L(\mathbf{w}_k) \tag{4}$$

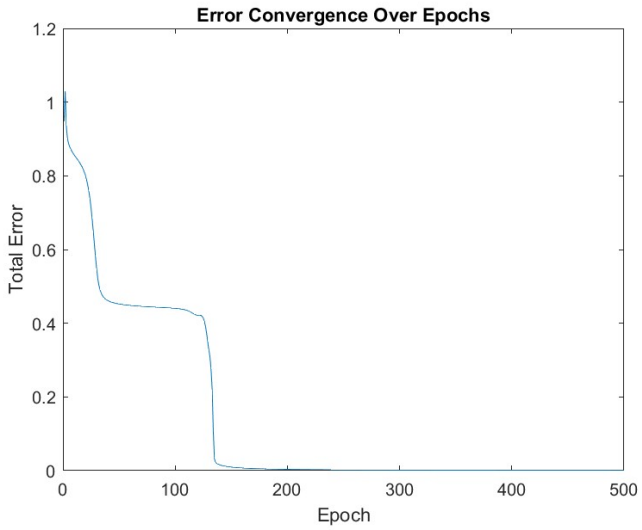

Figure 1: Error convergence plot showing the total error decreasing over the training epochs.

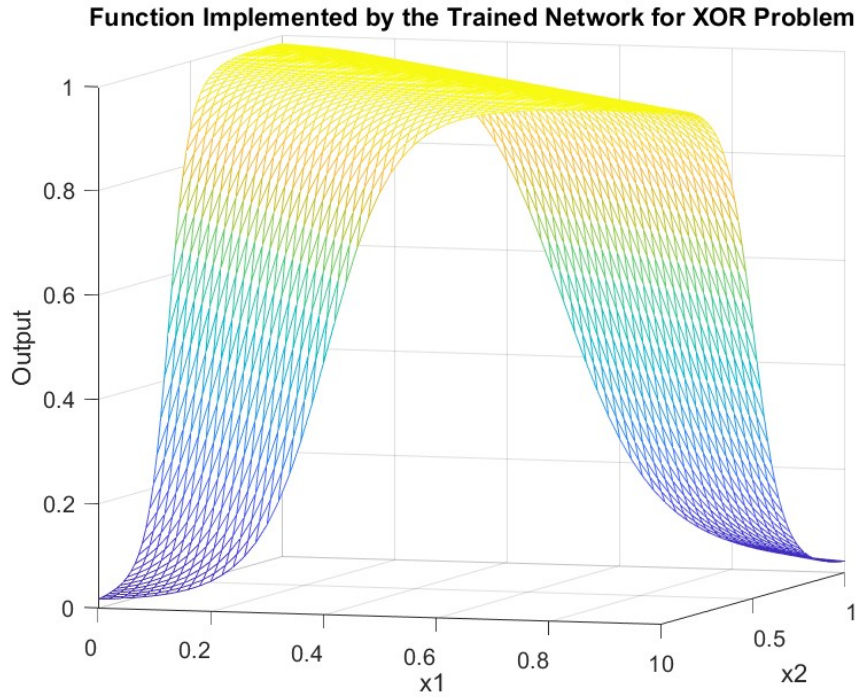

Figure 2: Plot of the function implemented by the trained network.

where η is the learning rate and $\nabla L(\mathbf{w}_k)$ is the gradient of the loss function with respect to the weights.

Code Implementation: To implement gradient descent, we would modify our weight update rules in the XOR problem code as follows:

```
% Compute gradients
dL_dw11 = delta1 * x_d(1);
dL_dw12 = delta1 * x_d(2);
dL_dw21 = delta2 * x_d(1);
dL_dw22 = delta2 * x_d(2);
dL_dw31 = delta3 * h1;
dL_dw32 = delta3 * h2;

% Update weights
```

```
w11 = w11 - eta * dL_dw11;
w12 = w12 - eta * dL_dw12;
w21 = w21 - eta * dL_dw21;
w22 = w22 - eta * dL_dw22;
w31 = w31 - eta * dL_dw31;
w32 = w32 - eta * dL_dw32;
```

4.2 Quasi-Newton Methods

Quasi-Newton methods, such as the BFGS algorithm, approximate the Hessian matrix to take more informed steps towards the minimum.

Mathematical Formulation: The BFGS update rule for the approximate inverse Hessian H_k is:

$$H_{k+1} = (I - \rho_k \mathbf{s}_k \mathbf{y}_k^T) H_k (I - \rho_k \mathbf{y}_k \mathbf{s}_k^T) + \rho_k \mathbf{s}_k \mathbf{s}_k^T \tag{5}$$

where:

- $\mathbf{s}_k = \mathbf{w}_{k+1} - \mathbf{w}_k$
- $\mathbf{y}_k = \nabla L(\mathbf{w}_{k+1}) - \nabla L(\mathbf{w}_k)$
- $\rho_k = \frac{1}{\mathbf{y}_k^T \mathbf{s}_k}$

The weight update rule becomes:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - H_k \nabla L(\mathbf{w}_k) \tag{6}$$

Code Implementation: To implement BFGS, we would need to modify our main training loop:

```
% Initialize H as identity matrix
n = numel([w11; w12; w21; w22; w31; w32]);
H = eye(n);

for epoch = 1:epochs
    w_old = [w11; w12; w21; w22; w31; w32];
    grad_old = compute_gradient(w_old, X, y);

    % Compute search direction
    p = -H * grad_old;

    % Line search to find step size alpha
    alpha = line_search(w_old, p, X, y);

    % Update weights
    w_new = w_old + alpha * p;
    [w11, w12, w21, w22, w31, w32] = unpack_weights(w_new);

    % Compute new gradient
    grad_new = compute_gradient(w_new, X, y);

    % Compute s and y
    s = w_new - w_old;
    y = grad_new - grad_old;

    % BFGS update
    rho = 1 / (y' * s);
    H = (eye(n) - rho * s * y') * H * (eye(n) - rho * y * s') + rho * s * s';

    % Compute error
    total_error = compute_error(w_new, X, y);
    errors(epoch) = total_error;
end
```

Note that this implementation requires additional helper functions like ‘compute_gradient‘, ‘line_search‘, ‘unpack_weights‘, and ‘compute_error‘.

4.3 Kaczmarz's Algorithm

Kaczmarz's algorithm is an iterative method for solving systems of linear equations, which can be applied to neural network training.

Mathematical Formulation: For a system of linear equations $A\mathbf{x} = \mathbf{b}$, the Kaczmarz update rule is:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{b_i - \mathbf{a}_i^T \mathbf{x}_k}{\|\mathbf{a}_i\|^2} \mathbf{a}_i \tag{7}$$

where $\mathbf{a}_i$ is the i-th row of A and b_i is the i-th element of $\mathbf{b}$.

Code Implementation: To apply Kaczmarz's algorithm to neural network training, we need to linearize the problem. Here's a simplified implementation:

```
% Linearize the problem
A = [X, ones(size(X, 1), 1)];  % Add bias term
b = y;

% Initialize weights
w = zeros(size(A, 2), 1);

for epoch = 1:epochs
    for i = 1:size(A, 1)
        a_i = A(i, :)';
        b_i = b(i);

        % Kaczmarz update
        w = w + (b_i - a_i' * w) / (a_i' * a_i) * a_i;
    end

    % Compute error
    y_pred = A * w;
    total_error = sum((y - y_pred).^2) / 2;
    errors(epoch) = total_error;
end

% Extract weights and bias
w11 = w(1); w12 = w(2); w21 = w(3); w22 = w(4);
bias = w(end);
```

Note that this implementation is for a single-layer network and would need to be adapted for multi-layer networks.

5 Why Gradient Descent Works

Gradient descent is effective for neural network training due to several factors:

1. **Convexity in local regions:** While the overall loss landscape of neural networks is non-convex, it often exhibits local convexity, allowing gradient descent to make progress. The loss function in a small neighborhood around a point $\mathbf{w}_0$ can be approximated as:

$$L(\mathbf{w}) \approx L(\mathbf{w}_0) + \nabla L(\mathbf{w}_0)^T (\mathbf{w} - \mathbf{w}_0) + \frac{1}{2} (\mathbf{w} - \mathbf{w}_0)^T H(\mathbf{w}_0) (\mathbf{w} - \mathbf{w}_0) \tag{8}$$

where $H(\mathbf{w}_0)$ is the Hessian matrix at $\mathbf{w}_0$. If $H(\mathbf{w}_0)$ is positive definite, this local region is convex.

2. **Stochasticity:** Stochastic gradient descent (SGD) introduces noise, which can help escape shallow local minima and saddle points. The update rule for SGD is:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \nabla L_i(\mathbf{w}_k) \tag{9}$$

where L_i is the loss for a single example or mini-batch.

3. **Overparameterization:** Modern neural networks are often overparameterized, creating a loss landscape with many good local minima. This can be represented mathematically as having more parameters than necessary:

$$\dim(\mathbf{w}) \gg \dim(\text{input}) \times \dim(\text{output}) \tag{10}$$

4. **Adaptive variants:** Techniques like Adam and RMSprop adapt the learning rate, improving convergence. For example, the Adam update rule is:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\eta}{\sqrt{\hat{v}_k} + \epsilon} \hat{m}_k \tag{11}$$

   where $\hat{m}_k$ and $\hat{v}_k$ are bias-corrected estimates of the first and second moments of the gradients.

However, gradient descent can struggle with:

- Vanishing/exploding gradients in deep networks
- Saddle points in high-dimensional spaces
- Very flat or sharp regions in the loss landscape

Each optimization method has its strengths and weaknesses:

- **Gradient Descent:** Simple to implement and works well for most problems, but can be slow to converge for ill-conditioned problems.
- **Quasi-Newton Methods:** Faster convergence and better handling of ill-conditioned problems, but higher computational complexity per iteration.
- **Kaczmarz's Algorithm:** Efficient for large-scale linear systems, but may struggle with highly nonlinear problems.

Gradient descent is often preferred in deep learning due to its simplicity, scalability to large datasets (when used in its stochastic form), and effectiveness when combined with techniques like momentum and adaptive learning rates.

6 Conclusion

While gradient descent remains a popular choice for neural network optimization due to its simplicity and effectiveness, more advanced methods like quasi-Newton and Kaczmarz's algorithm offer advantages in certain scenarios. The choice of optimization method depends on the specific problem, computational resources, and desired trade-offs between speed, memory usage, and accuracy. Understanding the mathematical foundations and implementation details of these methods allows for more informed decisions in algorithm selection and potential hybridization of techniques for optimal performance.