INF721 - Deep Learning
L6: Backpropagation

Prof. Lucas N. Ferreira
Universidade Federal de Viçosa

2024/2

1 Introduction

This lecture covers the backpropagation algorithm, which is fundamental for training neural networks efficiently. We'll explore how backpropagation works, its relationship to computational graphs, and see examples of applying it to logistic regression and multilayer perceptrons.

2 Review of Previous Lecture

Before diving into backpropagation, let's briefly review key concepts from the previous lecture:

- Non-linearly separable problems: We discussed how multilayer perceptrons can solve problems that linear models cannot.
- Multilayer Perceptron (MLP): We introduced the architecture of MLPs, which consist of multiple layers of neurons.
- Forward pass: We learned how to compute the output of an MLP given an input.
- Hypothesis space: We explored how MLPs can represent composite functions, expanding their representational power.
- Categorical Cross-Entropy Loss: We introduced this loss function for multiclass classification problems.

3 Gradient Descent for Neural Networks

Gradient descent is the core optimization algorithm used to train neural networks. The general process remains the same as for simpler models, but computing the gradients becomes more complex. Here's a high-level overview of gradient descent for a 2-layer neural network:

```
def optimize(x, y, lr, n_iter):
    # Initialize weights randomly close to 0
    W_1, b_1, W_2, b_2 = init_weights_rand()

    for t in range(n_iter):
        # Forward pass: Predict labels
        y_hat = forward(W_1, b_1, W_2, b_2)

        # Compute gradients
        dw_1, db_1, dw_2, db_2 = backward()

        # Update weights
        W_1 = W_1 - lr * dw_1
        b_1 = b_1 - lr * db_1
        W_2 = W_2 - lr * dw_2
        b_2 = b_2 - lr * db_2

    return W_1, b_1, W_2, b_2
```

The key challenge lies in computing the gradients efficiently, which is where backpropagation comes in.

4 The Need for Backpropagation

For simple models like linear or logistic regression, we can compute gradients by hand:

- Linear Regression: $\frac{\partial L}{\partial w} = (\hat{y} - y)x$, $\frac{\partial L}{\partial b} = (\hat{y} - y)$
- Logistic Regression: $\frac{\partial L}{\partial w} = (\hat{y} - y)x$, $\frac{\partial L}{\partial b} = (\hat{y} - y)$

However, as neural networks grow in size and complexity, manual computation of gradients becomes:

1. Error-prone: It's easy to make mistakes in long derivations.
2. Inflexible: Changing the model or loss function requires recomputing all gradients.
3. Time-consuming: Deriving gradients for large networks is tedious and impractical.

Backpropagation solves these issues by providing an efficient, automated way to compute gradients for any neural network architecture.

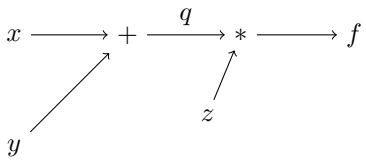
5 Computational Graphs

Before diving into backpropagation, we need to understand computational graphs. A computational graph is a directed graph that represents mathematical operations:

- Nodes represent functions of their inputs
- Edges represent function arguments

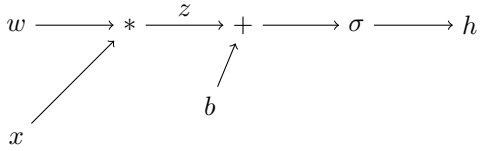
5.1 Example 1: Simple Function

Consider the function $f(x, y, z) = (x + y)z$. We can represent this as a computational graph:



5.2 Example 2: Logistic Regression

We can also represent more complex functions, like logistic regression, as a computational graph:



Here, $h(w, x, b) = \sigma(wx + b)$, where σ is the sigmoid function.

Computational graphs provide a structured way to represent and compute complex functions, which is crucial for implementing backpropagation efficiently.

6 Backpropagation Algorithm

Backpropagation is an algorithm that uses computational graphs and the chain rule of calculus to compute gradients of a function efficiently. It consists of two main steps:

1. Forward Pass: Compute the outputs of the function, storing partial results in each node.
2. Backward Pass: Compute the derivative of the output with respect to each input, using the chain rule and the stored partial results.

6.1 Chain Rule Reminder

The chain rule states that for composite functions:

$$\frac{d}{dx}f(g(x)) = \frac{df}{dg} \cdot \frac{dg}{dx}$$

This is crucial for backpropagation, as neural networks are essentially large composite functions.

6.2 Example: Backpropagation on a Simple Function

Let's apply backpropagation to compute gradients for $f(x, y, z) = (x + y)z$, considering $x = -2, y = 5, z = -4$:

1. Forward Pass:

$$\begin{aligned}
x &= -2 \\
y &= 5 \\
z &= -4 \\
q &= x + y = (-2) + 5 = 3 \\
f &= qz = 3 \cdot (-4) = -12
\end{aligned}$$

2. Backward Pass:

$$\begin{aligned}
\frac{\partial f}{\partial f} &= 1 \quad \text{(base case)} \\
\frac{\partial f}{\partial z} &= \frac{\partial f}{\partial z} \cdot \frac{\partial f}{\partial f} = q \cdot 1 = 3 \\
\frac{\partial f}{\partial q} &= \frac{\partial f}{\partial q} \cdot \frac{\partial f}{\partial f} = z \cdot 1 = -4 \\
\frac{\partial f}{\partial y} &= \frac{\partial q}{\partial y} \cdot \frac{\partial f}{\partial q} = 1 \cdot (-4) = -4 \\
\frac{\partial f}{\partial x} &= \frac{\partial q}{\partial x} \cdot \frac{\partial f}{\partial q} = 1 \cdot (-4) = -4
\end{aligned}$$

This example demonstrates how backpropagation efficiently computes gradients by reusing partial results and applying the chain rule. Notice how the gradients form a chain of multiplications to compute the derivatives of the function f with respect to its inputs x, y, z:

$$\begin{aligned}
\frac{\partial f}{\partial x} &= \frac{\partial q}{\partial x} \cdot \frac{\partial f}{\partial q} \cdot \frac{\partial f}{\partial f} \\
\frac{\partial f}{\partial y} &= \frac{\partial q}{\partial y} \cdot \frac{\partial f}{\partial q} \cdot \frac{\partial f}{\partial f} \\
\frac{\partial f}{\partial z} &= \frac{\partial f}{\partial z} \cdot \frac{\partial f}{\partial f}
\end{aligned}$$

7 Backpropagation for Logistic Regression

Now let's apply backpropagation to a more practical example: logistic regression. We'll compute gradients of the binary cross-entropy loss with respect to the model parameters.

Model:

$$\hat{y} = h(x) = \frac{1}{1 + e^{-(wx+b)}}$$

Loss:

$$L(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

Given input $(x, y) = (50, 1)$ and initial parameters $w = 0, b = 0$:

1. Forward Pass:

$$\begin{aligned} z &= wx + b = 0 \cdot 50 + 0 = 0 \\ \hat{y} &= \sigma(z) = \frac{1}{1 + e^{-0}} = 0.5 \\ L &\approx 0.69 \end{aligned}$$

2. Backward Pass:

$$\begin{aligned} \frac{\partial L}{\partial L} &= 1 \\ \frac{\partial L}{\partial \hat{y}} &= \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial L}{\partial L} = \left(-\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}} \right) \cdot 1 = -\frac{1}{0.5} + \frac{0}{0.5} = -2 \\ \frac{\partial L}{\partial z} &= \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial L}{\partial \hat{y}} = [\hat{y}(1 - \hat{y})] \cdot (-2) = [0.5(1 - 0.5)] \cdot (-2) = -0.5 \\ \frac{\partial L}{\partial w} &= \frac{\partial z}{\partial w} \cdot \frac{\partial L}{\partial z} = x \cdot (-0.5) = 50 \cdot (-0.5) = -25 \end{aligned}$$

This example shows how backpropagation computes gradients for logistic regression, which we can then use in gradient descent to update the parameters w and b.

8 Backpropagation for Multilayer Perceptron

Finally, let's extend backpropagation to a 2-layer multilayer perceptron (MLP). This example will demonstrate how backpropagation scales to deeper networks.

Model:

$$\begin{aligned} z^{[1]} &= W^{[1]} x + b^{[1]} \\ a^{[1]} &= g(z^{[1]}) \\ z^{[2]} &= W^{[2]} a^{[1]} + b^{[2]} \\ \hat{y} &= \sigma(z^{[2]}) \end{aligned}$$

Loss:

$$L(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

Let's consider a simple example with the following parameters:

- Input: $x = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and true label: $y = 1$
- Hidden layer size: 2 neurons
- Activation function for hidden layer: ReLU, $g(z) = \max(0, z)$
- Output layer: Sigmoid activation
- Weights and biases:

$$W^{[1]} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}, \quad b^{[1]} = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$
$$W^{[2]} = \begin{bmatrix} 0.5 & 0.6 \end{bmatrix}, \quad b^{[2]} = 0.3$$

We'll compute gradients with respect to $W^{[1]}, b^{[1]}, W^{[2]}$, and $b^{[2]}$. The process is similar to logistic regression but involves more steps due to the additional layer.

1. Forward Pass:

   (a) Compute $z^{[1]}$:

$$\begin{aligned} z^{[1]} &= W^{[1]} x + b^{[1]} \\ &= \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.1(1) + 0.2(2) \\ 0.3(1) + 0.4(2) \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} \\ &= \begin{bmatrix} 0.5 \\ 1.1 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.6 \\ 1.3 \end{bmatrix} \end{aligned}$$

   (b) Compute $a^{[1]}$ using ReLU activation:

$$a^{[1]} = g(z^{[1]}) = \max(0, z^{[1]}) = \begin{bmatrix} \max(0, 0.6) \\ \max(0, 1.3) \end{bmatrix} = \begin{bmatrix} 0.6 \\ 1.3 \end{bmatrix}$$

   (c) Compute $z^{[2]}$:

$$\begin{aligned} z^{[2]} &= W^{[2]} a^{[1]} + b^{[2]} \\ &= \begin{bmatrix} 0.5 & 0.6 \end{bmatrix} \begin{bmatrix} 0.6 \\ 1.3 \end{bmatrix} + 0.3 = (0.5 \cdot 0.6 + 0.6 \cdot 1.3) + 0.3 \\ &= (0.3 + 0.78) + 0.3 = 1.38 \end{aligned}$$

   (d) Compute $\hat{y}$ using sigmoid activation:

$$\hat{y} = \sigma(z^{[2]}) = \frac{1}{1 + e^{-z^{[2]}}} = \frac{1}{1 + e^{-1.38}} \approx 0.7986$$

(e) Compute the loss:

$$\begin{aligned} L(\hat{y}, y) &= -y \log \hat{y} - (1-y) \log(1-\hat{y}) \\ &= -(1) \log(0.7986) - (1-1) \log(1-0.7986) \\ &= -\log(0.7986) \\ &\approx 0.2247 \end{aligned}$$

2. Backward Pass:

$$\frac{\partial L}{\partial L} = 1$$

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial L}{\partial L} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \cdot 1 = -1.2521$$

$$\frac{\partial L}{\partial z^{[2]}} = \frac{\partial \hat{y}}{\partial z^{[2]}} \cdot \frac{\partial L}{\partial \hat{y}} = \hat{y}(1-\hat{y}) \cdot -1.2521 = -0.2016$$

$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial z^{[2]}}{\partial W^{[2]}} \cdot \frac{\partial L}{\partial z^{[2]}} = (a^{[1]})^T \cdot -0.2016 = \begin{bmatrix} -0.1210 & -0.2621 \end{bmatrix}$$

$$\frac{\partial L}{\partial b^{[2]}} = \frac{\partial z^{[2]}}{\partial b^{[2]}} \cdot \frac{\partial L}{\partial z^{[2]}} = 1 \cdot -0.2016 = -0.2016$$

$$\frac{\partial L}{\partial a^{[1]}} = \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial L}{\partial z^{[2]}} = (W^{[2]})^T \cdot -0.2016 = \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix} \cdot -0.2016 = \begin{bmatrix} -0.1008 \\ -0.1210 \end{bmatrix}$$

$$\frac{\partial L}{\partial z^{[1]}} = \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial L}{\partial a^{[1]}} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \odot \begin{bmatrix} -0.1008 \\ -0.1210 \end{bmatrix} = \begin{bmatrix} -0.1008 \\ -0.1210 \end{bmatrix}$$

$$\frac{\partial L}{\partial W^{[1]}} = \frac{\partial z^{[1]}}{\partial W^{[1]}} \cdot \frac{\partial L}{\partial z^{[1]}} = x^T \cdot \frac{\partial L}{\partial z^{[1]}} = \begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} -0.1008 \\ -0.1210 \end{bmatrix} = \begin{bmatrix} -0.1008 & -0.2016 \\ -0.1210 & -0.2420 \end{bmatrix}$$

$$\frac{\partial L}{\partial b^{[1]}} = \frac{\partial z^{[1]}}{\partial b^{[1]}} \cdot \frac{\partial L}{\partial z^{[1]}} = 1 \cdot \frac{\partial L}{\partial z^{[1]}} = \begin{bmatrix} -0.1008 \\ -0.1210 \end{bmatrix}$$

Note: $\odot$ denotes element-wise multiplication. The derivative of ReLU is 1 for positive inputs and 0 for negative inputs. Since both elements of $z^{[1]}$ are positive, the derivative is 1 for both.

This example demonstrates how backpropagation computes gradients through multiple layers of a neural network. The process involves repeatedly applying the chain rule to propagate gradients backwards from the output layer to the input layer. This allows us to update all parameters of the network based on their contribution to the final loss.

The computed gradients can now be used to update the network parameters using an optimization algorithm like gradient descent:

$$W^{[2]} = W^{[2]} - \alpha \cdot \frac{\partial L}{\partial W^{[2]}}$$
$$b^{[2]} = b^{[2]} - \alpha \cdot \frac{\partial L}{\partial b^{[2]}}$$
$$W^{[1]} = W^{[1]} - \alpha \cdot \frac{\partial L}{\partial W^{[1]}}$$
$$b^{[1]} = b^{[1]} - \alpha \cdot \frac{\partial L}{\partial b^{[1]}}$$

, where α is the learning rate.

9 Conclusion

Backpropagation is a powerful algorithm that enables efficient training of neural networks. By leveraging computational graphs and the chain rule, it provides a systematic way to compute gradients for any network architecture. This allows us to apply gradient descent to optimize complex models with millions of parameters.

Key takeaways:

1. Backpropagation automates gradient computation, reducing errors and increasing flexibility.
2. It uses a forward pass to compute outputs and a backward pass to compute gradients.
3. The algorithm efficiently reuses partial results, making it computationally efficient.
4. Understanding backpropagation is crucial for implementing and optimizing neural networks.

In the next lecture, we'll explore techniques for evaluating deep learning models, including metrics like accuracy, learning curves, cross-validation, and confusion matrices.