Mathematical Derivation: Softmax + CrossEntropy Gradients

1. Function Definitions

Softmax Function

For input vector **z** = $[z_1, z_2, ..., z_{\square}]$, the softmax function produces:

$\sigma(z)_i = \exp(z_i) / \Sigma_{\square} \exp(z_{\square})$

Where:

- **$\sigma(z)_i$** = softmax output for class i
- **z_i** = raw logit for class i
- The denominator ensures all outputs sum to 1

Cross-Entropy Loss Function

For true label **y** (one-hot encoded) and predicted probabilities **p**:

$L = -\Sigma_i y_i \log(p_i)$

For single sample with true class **c**:

$L = -\log(p^c)$

Where **p^c** is the predicted probability for the correct class.

2. Individual Derivatives

2.1 Cross-Entropy Derivative

$\partial L/\partial p_i$ = derivative of loss with respect to softmax output

For the correct class (i = c):

$\partial L/\partial p^c = \partial(-\log(p^c))/\partial p^c = -1/p^c$

For incorrect classes (i ≠ c):

$\partial L/\partial p_i = \partial(-\log(p^c))/\partial p_i = 0$

In vector form:

$\partial L/\partial p = [-1/p^c, 0, 0, ..., 0]$ (only non-zero at correct class position)

2.2 Softmax Derivative (The Complex Part)

$\partial p_i/\partial z_\square$ = derivative of softmax output i with respect to input j

This is where it gets complex because changing one input affects ALL outputs (they must sum to 1).

Case 1: i = j (diagonal elements)

$\partial p_i/\partial z_i = \partial/\partial z_i\ [\exp(z_i) / \Sigma_\square \exp(z_\square)]$

Using quotient rule:
$= [\exp(z_i) \times \Sigma_\square \exp(z_\square) - \exp(z_i) \times \exp(z_i)] / (\Sigma_\square \exp(z_\square))^2$
$= \exp(z_i)/\Sigma_\square \exp(z_\square) \times [1 - \exp(z_i)/\Sigma_\square \exp(z_\square)]$
$= p_i \times (1 - p_i)$

Case 2: i ≠ j (off-diagonal elements)

$\partial p_i/\partial z_\square = \partial/\partial z_\square\ [\exp(z_i) / \Sigma_\square \exp(z_\square)]$

Since numerator doesn't depend on $z_\square$:
$= -\exp(z_i) \times \exp(z_\square) / (\Sigma_\square \exp(z_\square))^2$
$= -p_i \times p_\square$

Jacobian Matrix:

$$J = \begin{bmatrix} p_1(1-p_1) & -p_1p_2 & -p_1p_3 \\ -p_2p_1 & p_2(1-p_2) & -p_2p_3 \\ -p_3p_1 & -p_3p_2 & p_3(1-p_3) \end{bmatrix}$$

This is exactly what your complex softmax backward implements:

jacobian_matrix = np.diagflat(single_output) - np.dot(single_output, single_output.T)

3. Chain Rule Application (Separate Approach)

To get **$\partial L/\partial z$**, we need:

$\partial L/\partial z_j = \Sigma_i\ (\partial L/\partial p_i) \times (\partial p_i/\partial z_j)$

For correct class (j = c):

$$\begin{aligned}\partial L/\partial z^c &= (\partial L/\partial p^c) \times (\partial p^c/\partial z^c) + \Sigma_{i}{\neq}^c\ (\partial L/\partial p_i) \times (\partial p_i/\partial z^c)\\ &= (-1/p^c) \times p^c(1-p^c) + \Sigma_{i}{\neq}^c\ (0) \times (-p_i p^c)\\ &= -(1-p^c) + 0\\ &= p^c - 1\end{aligned}$$

For incorrect class (j ≠ c):

$$\begin{aligned}\partial L/\partial z_j &= (\partial L/\partial p^c) \times (\partial p^c/\partial z_j) + \Sigma_{i}{\neq}^c\ (\partial L/\partial p_i) \times (\partial p_i/\partial z_j)\\ &= (-1/p^c) \times (-p^c p_j) + 0\\ &= p_j\end{aligned}$$

4. The Beautiful Simplification

Final result:

$$\partial L/\partial z_i = \begin{cases} p_i - 1, & \text{if } i = \text{correct class}\\ p_i, & \text{if } i \neq \text{correct class}\end{cases}$$

In vector form:

$\partial L/\partial z = p - y\_one\_hot$

Where:

- **p** = softmax output [p_1, p_2, p_3]
- **y_one_hot** = one-hot true label [0, 1, 0]

5. Code Implementation Explanation

```
def backward(self, dvalues, y_true):
    samples = len(dvalues)

    # Convert sparse labels to indices if needed
    if len(y_true.shape) == 2:
        y_true = np.argmax(y_true, axis=1)

    # Start with softmax output: p = [p₁, p₂, p₃]
    self.dinputs = dvalues.copy()  # dvalues = softmax output
```

```
# Subtract 1 from correct class: p - y_one_hot
self.dinputs[range(samples), y_true] -= 1

# Normalize by batch size
self.dinputs = self.dinputs / samples
```

Step-by-Step Example:

Given:

- Softmax output: `p = [0.2, 0.7, 0.1]`
- True class: `1` (middle class)
- One-hot: `y = [0, 1, 0]`

Calculation:

1. `dinputs = [0.2, 0.7, 0.1]` (copy softmax output)
2. `dinputs[1] -= 1` → `[0.2, 0.7-1, 0.1]` = `[0.2, -0.3, 0.1]`
3. Normalize by samples

Interpretation:

- `[0.2, -0.3, 0.1]` means:
  - Class 0: Reduce probability by 0.2
  - Class 1: Increase probability by 0.3 (negative gradient)
  - Class 2: Reduce probability by 0.1

6. Why the Jacobian Cancels Out

The key insight is that when you multiply the complex softmax Jacobian with the simple cross-entropy gradient, most terms cancel:

Complex Jacobian × Simple Gradient = Simple Result

$$\begin{bmatrix} p_1(1-p_1) & -p_1p_2 & -p_1p_3 \\ -p_2p_1 & p_2(1-p_2) & -p_2p_3 \\ -p_3p_1 & -p_3p_2 & p_3(1-p_3) \end{bmatrix} \times \begin{bmatrix} -1/p^c \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} p^c - 1 \\ p_{\square} \\ p_{\square} \end{bmatrix} \begin{matrix} \\ \text{(for } j \neq c) \\ \text{(for } k \neq c) \end{matrix}$$

The beautiful cancellation happens because:

- Cross-entropy only cares about the correct class
- Softmax relationships are perfectly structured
- The mathematics "conspires" to give us the simple result

7. Computational Comparison

Separate Calculation:

```
# For each sample:
#   - Create 3×3 Jacobian matrix (9 operations)
#   - Matrix multiplication (9 more operations)
#   - Loop over all samples
# Total: $O(n \times \text{classes}^2)$ operations
```

Combined Calculation:

```
# For all samples:
#   - Copy array (1 operation)
#   - Subtract 1 at specific indices (1 operation)
#   - Divide by samples (1 operation)
# Total: O(n) operations
```

8. Numerical Stability Benefits

Separate approach:

- Risk of division by very small probabilities
- Multiple floating-point operations compound errors
- Jacobian matrix calculations can be unstable

Combined approach:

- No division by probabilities
- Fewer operations = less numerical error
- More robust to edge cases