Softmax

Mithun Prasad, PhD
miprasad@Microsoft.com

Microsoft

Softmax

- The softmax function takes an N-dimensional vector of arbitrary real values and produces another N-dimensional vector with real values in the range (0, 1) that add up to 1.0.

- Maps $S(a): R^N \rightarrow R^N$

$$S(a): \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_N \end{bmatrix} \rightarrow \begin{bmatrix} S_1 \\ S_2 \\ \dots \\ S_N \end{bmatrix}$$

- Per-element formula is:

$$S_j = \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} \quad \forall j \in 1 \dots N$$

Microsoft

Properties Softmax

- S_j is always positive (because of the exponents).
- The numerator appears in the denominator summed up with some other positive numbers, $S_j < \mathbf{1}.$ Therefore, it's in the range (0, 1).
- For example, the 3-element vector [1.0, 2.0, 3.0] gets transformed into [0.09, 0.24, 0.67], which still preserves these properties.

Probabilistic Interpretation

- The properties of softmax make it suitable for a probabilistic interpretation.

- Very useful in machine learning.

- In multiclass classification tasks, we often want to assign probabilities that our input belongs to one of many output classes.

- If we have N output classes, we're looking for an N-vector of probabilities that sum up to N; sounds familiar?

- We can interpret softmax as follows:

$$S_j = P(y = j|a)$$

Microsoft

Probabilistic Interpretation

- We can interpret softmax as follows:

$$S_j = P(y = j|a)$$

- Where y is the output class numbered $1 \ldots N$. a is any N-vector.

- For example, multiclass logistic regression where an input vector x is multiplied by a weight matrix W, and the result of this dot product is fed into a softmax function to produce probabilities.

- From a probabilistic point of view, softmax is optimal for maximum-likelihood estimation of the model's parameters.

Microsoft

Computing Softmax and Numerical Stability

- A simple way of computing the softmax function on a given vector in Python is:

```python
def softmax(x):
    """Compute the softmax of vector x."""
    exps = np.exp(x)
    return exps / np.sum(exps)
```

- Let's try it with the sample 3-element vector we've used as an example earlier:

```
In [146]: softmax([1, 2, 3])
Out[146]: array([ 0.09003057,  0.24472847,  0.66524096])
```

- However, if we run this function with larger numbers (or large negative numbers) we have a problem:

```
In [148]: softmax([1000, 2000, 3000])
Out[148]: array([ nan,  nan,  nan])
```

Numerical Range Limitations

- The numerical range of the floating-point numbers used by *Numpy* is limited.

- For float64, the maximal representable number is on the order of 10^{308}. Exponentiation in the softmax function makes it possible to easily overshoot this number, even for fairly modest-sized inputs.

- A nice way to avoid this problem is by normalizing the inputs to be not too large or too small, by observing that we can use an arbitrary constant C as follows:

$$S_j = \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} = \frac{Ce^{a_j}}{\sum_{k=1}^{N} Ce^{a_k}}$$

- And then pushing the constant into the exponent, we get:

$$S_j = \frac{e^{a_j+\log(C)}}{\sum_{k=1}^{N} e^{a_k+\log(C)}}$$

Microsoft

Numerical Range Limitations

- Since C is just an arbitrary constant, we can instead write:

$$S_j = \frac{e^{a_j+D}}{\sum_{k=1}^{N} e^{a_k+D}}$$

- Where D is also an arbitrary constant. This formula is equivalent to the original S_j for any D, so we're free to choose a D that will make our computation better numerically. A good choice is the maximum between all inputs, negated:

$$D = -\max(a_{1,}a_2, \dots, a_N)$$

- This will shift the inputs to a range close to zero, assuming the inputs themselves are not too far from each other. Crucially, it shifts them all to be negative (except the maximal a_j which turns into a zero).

- Negatives with large exponents "saturate" to zero rather than infinity, so we have a better chance of avoiding NaNs.

Numerical Range Limitations

```
def stablesoftmax(x):
    """Compute the softmax of vector x in a numerically stable way."""
    shiftx = x - np.max(x)
    exps = np.exp(shiftx)
    return exps / np.sum(exps)
```

- And now:

```
In [150]: stablesoftmax([1000, 2000, 3000])
Out[150]: array([ 0.,  0.,  1.])
```

- Note that this is still imperfect, since mathematically softmax would never really produce a zero, but this is much better than NaNs, and since the distance between the inputs is very large it's expected to get a result extremely close to zero anyway.