Classification and Loss Evaluation - Softmax and Cross Entropy Loss

Lets dig a little deep into how we convert the output of our CNN into probability - Softmax; and the loss measure to guide our optimization - Cross Entropy.

PARAS DAHAL

Note: Complete source code can be found here https://github.com/parasdahal/deepnet

The Softmax Function

Softmax function takes an N-dimensional vector of real numbers and transforms it into a vector of real number in range (0,1) which add upto 1. $p_i = \frac{e^{a_i}}{\sum_{k=1}^{N} e_k^a}$

As the name suggests, softmax function is a “soft” version of max function. Instead of selecting one maximum value, it breaks the whole (1) with maximal element getting the largest portion of the distribution, but other smaller elements getting some of it as well.

This property of softmax function that it outputs a probability distribution makes it suitable for probabilistic interpretation in classification tasks.

In python, we the code for softmax function as follows:

```python
def softmax(X):
    exps = np.exp(X)
    return exps / np.sum(exps)
```

We have to note that the numerical range of floating point numbers in numpy is limited. For `float64` the upper bound is 10^{308}. For exponential, its not difficult to overshoot that limit, in which case python returns `nan` .

To make our softmax function numerically stable, we simply normalize the values in the vector, by multiplying the numerator and denominator with a constant C.

$$\begin{aligned} p_i &= \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}} \\ &= \frac{Ce^{a_i}}{C\sum_{k=1}^{N} e^{a_k}} \\ &= \frac{e^{a_i + \log(C)}}{\sum_{k=1}^{N} e^{a_k + \log(C)}} \end{aligned}$$

We can choose an arbitrary value for $log(C)$ term, but generally $log(C) = -max(a)$ is chosen, as it shifts all of elements in the vector to negative to zero, and negatives with large exponents saturate to zero rather than the infinity, avoiding overflowing and resulting in `nan` .

The code for our stable softmax is as follows:

```python
def stable_softmax(X):
    exps = np.exp(X - np.max(X))
    return exps / np.sum(exps)
```

Derivative of Softmax

Due to the desirable property of softmax function outputting a probability distribution, we use it as the final layer in neural networks. For this we need to calculate the derivative or gradient and pass it back to the previous layer during backpropagation.

$$\frac{\partial p_i}{\partial a_j} = \frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j}$$

From quotient rule we know that for $f(x) = \frac{g(x)}{h(x)}$, we have $f'(x) = \frac{g\prime(x)h(x) - h\prime(x)g(x)}{h(x)^2}$

.

In our case $g(x) = e^{a_i}$ and $h(x) = \sum_{k=1}^{N} e^{a_k}$. In $h(x)$, $\frac{\partial}{\partial e^{a_j}}$ will always be e^{a_j} has it will always have e^{a_j}. But we have to note that in $g(x)$, $\frac{\partial}{\partial e^{a_j}}$ will be e^{a_j} only if $i = j$, otherwise its 0.

If $i = j$,

$$\begin{aligned}\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} &= \frac{e^{a_i} \sum_{k=1}^{N} e^{a_k} - e^{a_j} e^{a_i}}{\left(\sum_{k=1}^{N} e^{a_k}\right)^2} \\ &= \frac{e^{a_i} \left(\sum_{k=1}^{N} e^{a_k} - e^{a_j}\right)}{\left(\sum_{k=1}^{N} e^{a_k}\right)^2} \\ &= \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} \times \frac{\left(\sum_{k=1}^{N} e^{a_k} - e^{a_j}\right)}{\sum_{k=1}^{N} e^{a_k}} \\ &= p_i(1 - p_j)\end{aligned}$$

For $i \neq j$,

$$\begin{aligned}\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} &= \frac{0 - e^{a_j} e^{a_i}}{\left(\sum_{k=1}^{N} e^{a_k}\right)^2} \\ &= \frac{-e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} \times \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}} \\ &= -p_j . p_i\end{aligned}$$

So the derivative of the softmax function is given as,

$$\frac{\partial p_i}{\partial a_j} = \begin{cases} p_i(1 - p_j) & if \quad i = j \\ -p_j . p_i & if \quad i \neq j \end{cases}$$

Or using Kronecker delta $\delta ij = \begin{cases} 1 & if \quad i = j \\ 0 & if \quad i \neq j \end{cases}$

$$\frac{\partial p_i}{\partial a_j} = p_i(\delta_{ij} - p_j)$$

Cross Entropy Loss

Cross entropy indicates the distance between what the model believes the output distribution should be, and what the original distribution really is. It is defined as, $H(y,p) = -\sum_i y_i log(p_i)$ Cross entropy measure is a widely used alternative of squared error. It is used when node activations can be understood as representing the probability that each hypothesis might be true, i.e. when the output is a probability distribution. Thus it is used as a loss function in neural networks which have softmax activations in the output layer.

```
def cross_entropy(X,y):
    """
    X is the output from fully connected layer (num_examples x num_classes)
    y is labels (num_examples x 1)
        Note that y is not one-hot encoded vector.
        It can be computed as y.argmax(axis=1) from one-hot encoded vectors of labels if required.
    """
    m = y.shape[0]
    p = softmax(X)
    # We use multidimensional array indexing to extract
    # softmax probability of the correct label for each sample.
    # Refer to https://docs.scipy.org/doc/numpy/user/basics.indexing.html#indexing-multi-dimensional-arra
    log_likelihood = -np.log(p[range(m),y])
    loss = np.sum(log_likelihood) / m
    return loss
```

Derivative of Cross Entropy Loss with Softmax

Cross Entropy Loss with Softmax function are used as the output layer extensively. Now we use the derivative of softmax [1] that we derived earlier to derive the derivative of the cross entropy loss function.

$$
\begin{aligned}
L &= -\sum_i y_i log(p_i) \\
\frac{\partial L}{\partial o_i} &= -\sum_k y_k \frac{\partial log(p_k)}{\partial o_i} \\
&= -\sum_k y_k \frac{\partial log(p_k)}{\partial p_k} \times \frac{\partial p_k}{\partial o_i} \\
&= -\sum y_k \frac{1}{p_k} \times \frac{\partial p_k}{\partial o_i}
\end{aligned}
$$

From derivative of softmax we derived earlier,

$$
\begin{aligned}
\frac{\partial L}{\partial o_i} &= -y_i(1-p_i) - \sum_{k \neq i} y_k \frac{1}{p_k}(-p_k . p_i) \\
&= -y_i(1-p_i) + \sum_{k \neq 1} y_k . p_i \\
&= -y_i + y_i p_i + \sum_{k \neq 1} y_k . p_i \\
&= p_i \left(y_i + \sum_{k \neq 1} y_k \right) - y_i \\
&= p_i \left(y_i + \sum_{k \neq 1} y_k \right) - y_i
\end{aligned}
$$

y is a one hot encoded vector for the labels, so$\sum_k y_k = 1$, and $y_i + \sum_{k \neq 1} y_k = 1$. So we have,

$$\frac{\partial L}{\partial o_i} = p_i - y_i$$

which is a very simple and elegant expression. Translating it into code [2]

```
    def delta_cross_entropy(X,y):
        """
        X is the output from fully connected layer (num_examples x num_classes)
        y is labels (num_examples x 1)
            Note that y is not one-hot encoded vector.
            It can be computed as y.argmax(axis=1) from one-hot encoded vectors of labels if required.
        """
        m = y.shape[0]
        grad = softmax(X)
        grad[range(m),y] -= 1
        grad = grad/m
        return grad
```

References

1. **The Softmax function and its derivative** [link]
   Bendersky, E., 2016.

2. **CS231n Convolutional Neural Networks for Visual Recognition** [link]
   Andrej Karpathy, A.K., 2016.

Comments

张强 • 14 days ago

The code for the delta_cross_entropy seems to have something wrong.
x = np.array([11., 42., 3.])
y = np.array([1])
delta_cross_entropy(x,y)

```
---------------------------------------------------------------------------
IndexError Traceback (most recent call last)
<ipython-input-21-bbab36a3762b> in <module>
----> 1 delta_cross_entropy(x,y)

<ipython-input-5-f8bbefba54bf> in delta_cross_entropy(X, y)
8 m = y.shape[0]
9 grad = softmax(X)
---> 10 grad[range(m),y] -= 1
```