Sigmoid Activation and Binary Crossentropy — A Less Than Perfect Match?

Investigating concerns of numerical imprecision

Harald Hentschke Feb 21, 2019 · 7 min read

Really cross, and full of entropy...

In neuronal networks tasked with **binary classification**, **sigmoid** activation in the last (output) layer and **binary crossentropy (BCE)** as the loss function are standard fare. Yet, occasionally one stumbles across statements that this specific combination of last layer-activation and loss may result in **numerical imprecision or even instability**. I wanted to make sure I get the argument, down to the numbers, especially in the framework which I use so far, **Keras**. Sounds interesting? Realize that this may also be relevant for some image segmentation tasks, or multiclass, multilabel problems, not just cats_vs_dogs kind of problems? Then, please follow along. If you appreciate a refresher on BCE beforehand, I'd like to point to an excellent, in-depth explanation of BCE by Daniel Godoy.

1. Definition of the problem

Let's start by dissecting Keras' implementation of BCE:

```
def binary_crossentropy(target, output, from_logits=False):
    """Binary crossentropy between an output tensor and a target tensor.

    # Arguments
        target: A tensor with the same shape as `output`.
```

```
        output: A tensor.
        from_logits: Whether `output` is expected to be a logits tensor.
            By default, we consider that `output`
            encodes a probability distribution.

    # Returns
        A tensor.
    """
    # Note: tf.nn.sigmoid_cross_entropy_with_logits
    # expects logits, Keras expects probabilities.
    if not from_logits:
        # transform back to logits
        _epsilon = _to_tensor(epsilon(), output.dtype.base_dtype)
        output = tf.clip_by_value(output, _epsilon, 1 - _epsilon)
        output = tf.log(output / (1 - output))

    return tf.nn.sigmoid_cross_entropy_with_logits(labels=target,
                                                   logits=output)
```

So, input argument `output` is clipped first, then converted to logits, and then fed into TensorFlow function `tf.nn.sigmoid_cross_entropy_with_logits`. OK…what was logit(s) again? In mathematics, the logit function is the inverse of the sigmoid function, so in theory logit(sigmoid(x)) = x.

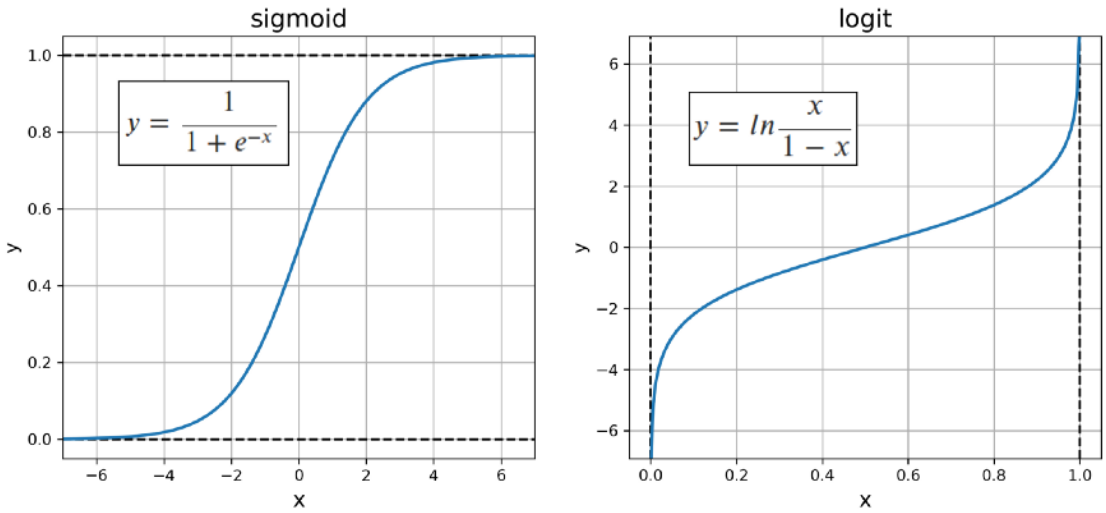


Figure 1: Curves you've likely seen before

In Deep Learning, logits usually and unfortunately means the 'raw' outputs of the last layer of a classification network, that is, the output of the layer **before** it is passed to an activation/normalization function, e.g. the sigmoid. Raw outputs may take on any value. This is what `sigmoid_cross_entropy_with_logits`, the core of Keras's

`binary_crossentropy`, expects. In Keras, by contrast, the expectation is that the values in variable `output` represent probabilities and are therefore bounded by [0 1] — that's why `from_logits` is by default set to False. So, they need to be converted back to raw values before being fed into `sigmoid_cross_entropy_with_logits`. To repeat, we **first run numbers through one function (sigmoid), only to convert them back using the inverse function (logit)**. This appears circuitous. The real potential problem, though, is the numerical instability that this to and fro may cause, resulting in an overflow in the extreme case. Look at the output of y = logit(sigmoid(x)) when x is of type float32, the default in Keras and, as far as I know, in most other frameworks, too:

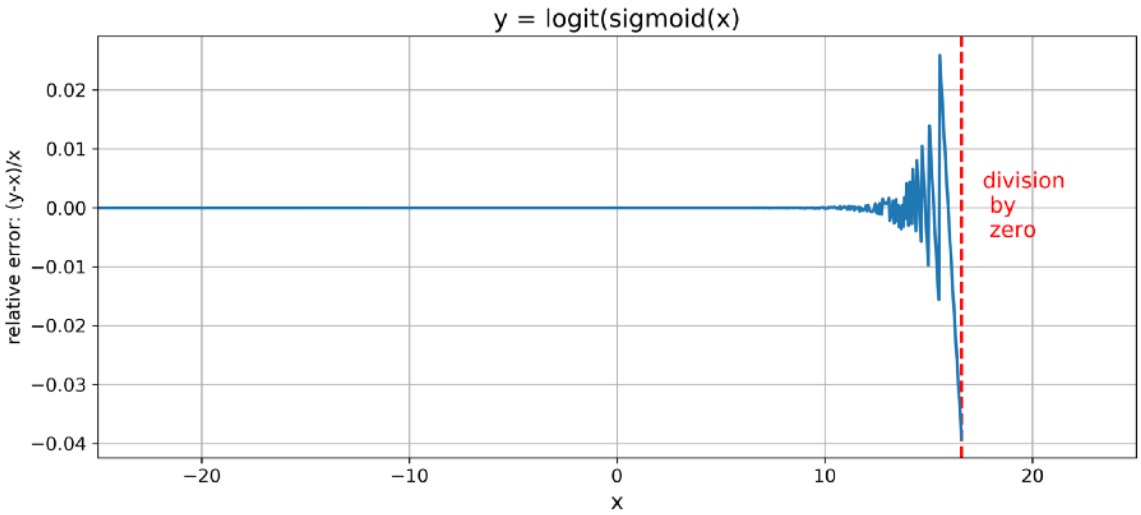


Figure 2: Numerical imprecisions with float32

Starting at about x=14.6 errors hit the 1% range, and above about x=16.6 game's over due to division by zero. Division by zero occurs when the denominator in the sigmoid evaluates to exactly 1 and the denominator in the logit then evaluates to zero. We could have estimated the limit of x via `np.log(np.finfo(np.float32).eps)`, but I found the numerical zigzagging entertaining and worth showing. Anyways, this is why the values of input variable `output` in the Keras function need to be and in fact are clipped. So, to dispel one notion right at the beginning:

Keras's binary_crossentropy, when fed with input resulting from sigmoid activation, will not produce over- or underflow of numbers.

However, the result of the clipping is a flattening of the loss function at the borders. To visualize this, let's

1. copy the neat trick of setting up minimalistic networks with manually set weights, and

2. run single inputs (samples, batch size of 1) through the networks.

Thus, we are able to compare the values of BCE as computed from sigmoid activation in Keras with the values computed from raw outputs in TensorFlow. Here is the first model, using sigmoid activation and Keras's standard BCE:

```
model_p = models.Sequential()
model_p.add(layers.Dense(1, input_dim=1, activation="sigmoid"))
# set layer weights manually
model_p.layers[0].set_weights([np.array([1.0]).reshape((1,1)), np.array([0.0])])
model_p.compile(loss="binary_crossentropy", optimizer="RMSprop")
```

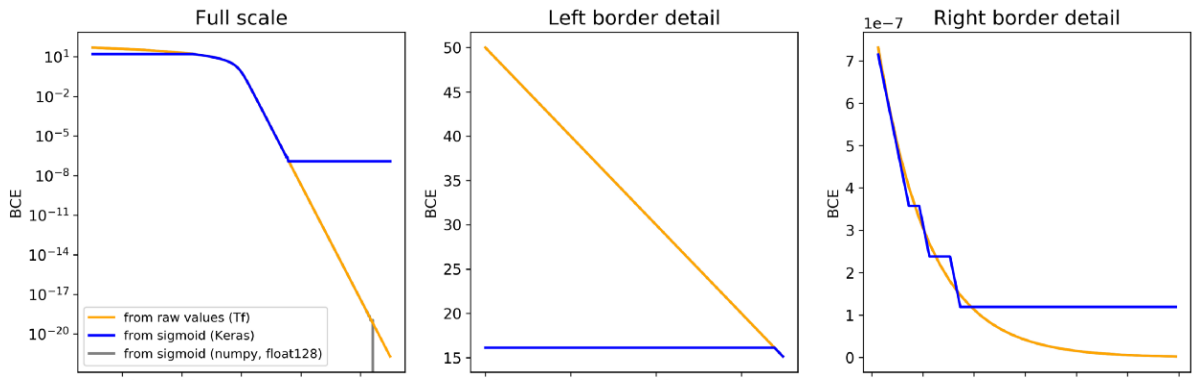
minimal_network_01.py hosted with ♡ by GitHub view raw

The model without sigmoid activation, using a custom-made loss function which plugs the values directly into `sigmoid_cross_entropy_with_logits`:

```
model_logit = models.Sequential()
model_logit.add(layers.Dense(1, input_dim=1, activation=None))
# set layer weights manually
model_logit.layers[0].set_weights([np.array([1.0]).reshape((1,1)), np.array([0.0])])
model_logit.compile(loss=bce_with_logits, optimizer="RMSprop")
```

minimal_network_02.py hosted with ♡ by GitHub view raw

So, if we evaluate the models on a sweeping range of scalar inputs x, setting the label (y) to 1, we can compare the model-generated BCEs with each other and also to the values produced by a naive implementation of BCE computed with a high-precision float. Note again that we're computing BCE from single samples here in order to distil the differences between the methods.



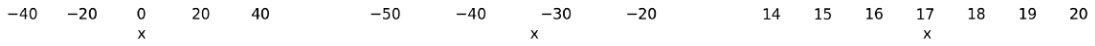


Figure 3: Binary crossentropy computed for single samples in three ways. Note logarithmic y axis in the left plot, and linear y axes in the others.

Interesting! The curve computed from raw values using TensorFlow's `sigmoid_cross_entropy_with_logits` is smooth across the range of x values tested, whereas the curve computed from sigmoid-transformed values with Keras's `binary_crossentropy` flattens in both directions (as predicted). At large positive x values, before hitting the clipping-induced limit, the sigmoid-derived curve shows a step-like appearance. Most notable may be what happens on the left border, which is best appreciated on a linear scale (Fig. 3, center): the more negative the raw value, the more severe the underestimation of its BCE value due to clipping. Finally, even with a float128 we won't get very far if we derive the BCE values from sigmoid-transformed input (gray curve in leftmost plot).

As the loss function is central to learning, this means that a model employing last-layer sigmoid + BCE **cannot discriminate among samples whose predicted class is either in extreme accordance or extreme discordance with their labels**. So, in theory it is true that there is a drawback to using sigmoid + BCE. However, does it matter in practice? After all, we are talking about extreme edge cases. And before we answer that question, a more basic question is in order: do raw last layer outputs reach such extreme values in practice? Let's see.

2. Checking individual samples' raw output values in binary classification networks

I trained three different networks to the task of categorizing dogs vs. cats, using a subset of the 2013 Kaggle competition data set (2000 training images, 1000 for validation; following the example of F. Chollet (*Deep Learning with Python*. Manning Publications Co, Shelter Island, New York. 2017.)). Yes, cats and dogs again, for the sake of ease and focusing on the issue at hand.

The figures and numbers below stem from a simple, hand-crafted convnet: four pairs of 2D convolution/max pooling layers, followed by a single dropout and two dense layers, all with relu activation. The last, single-element, output layer was without activation, and as the loss function I used above-mentioned Keras wrapper for TensorFlow's `sigmoid_cross_entropy_with_logits`.

First, let's find out whether individual images can in fact result in extreme raw values of the output layer. After training, I ran the same images as used for training through the network and obtained their raw output from the last layer. Additionally, I computed the sigmoid-transformed output, as well as the BCE values derived from both outputs. This is what I got after training for **eight epochs**, so with relatively little learning having taken place:

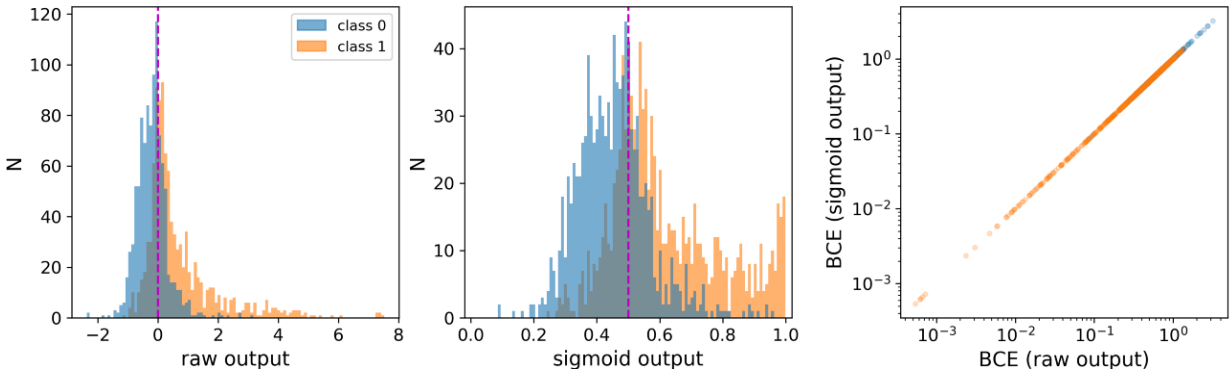


Figure 4: Left and center, distribution of outputs of the last layer after training the network for **eight epochs**. Input for prediction consisted of the same 2000 images as used for training. Left, raw values; center, sigmoid-transformed values. Right, scatter plot of BCE values computed from sigmoid output vs. those computed from raw output. Batch size = 1.

Obviously, in the initial phase of training, we are outside the danger zone; raw last layer output values are bounded by ca [-3 8] in this example, and BCE values computed from raw and sigmoid outputs are identical. Also nice to see is the strong 'squashing' effect of the sigmoid (Fig. 4, center).

What is the picture like when the network is **fully trained** (here defined after not having shown a reduced loss in 15 consecutive epochs)?

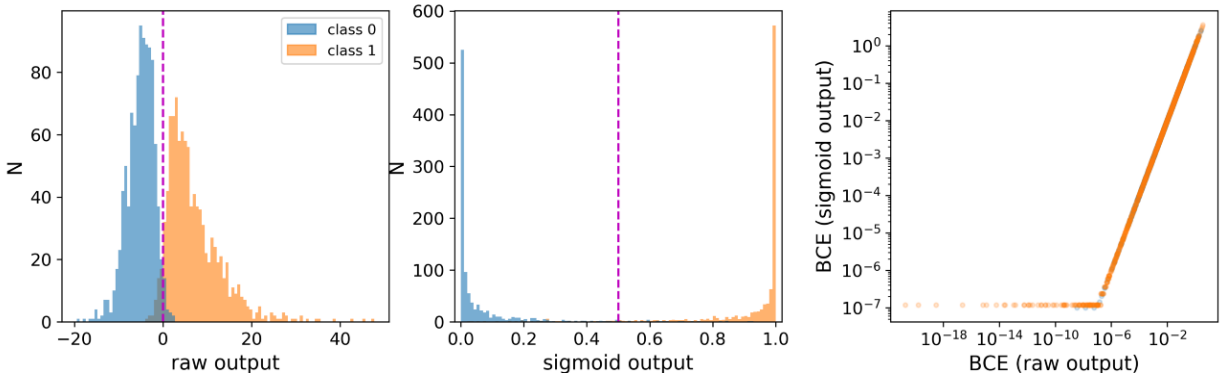


Figure 5: Same depiction as in Fig. 4, but after **full training** of the network (test accuracy of ca. 0.85). Same conventions as in Fig. 4 apply. Note clipping of BCE values computed from sigmoid outputs (right).

Aha —we see a much clearer separation between the classes, as expected. And a small number of images did in fact result in extreme logit values that fall into the clipping range. Let's focus on class 1 (dogs; orange color in the figures); the argument runs similar for the other class. None of the samples result in raw output values more negative than ca. -4, so again that is fine. However, a certain number of samples — the doggiest dogs — reach raw output values larger than about 16. Accordingly, the associated BCE, computed via sigmoid + Keras's binary_crossentropy, is clipped at ca. 10^{-7} (Fig. 5, right; see also Fig. 3). This is a really small value. Would we expect learning to happen in a systematically different fashion if BCE values of doggy dogs and catty cats were smaller (and individually different) when computed without clipping-induced limits? **Particularly if we use reasonable batch sizes, the samples with intermediate or low raw output values will dominate the loss.** Figure 6 illustrates this for the same data as above with a batch size of 4, which is still really on the low side.

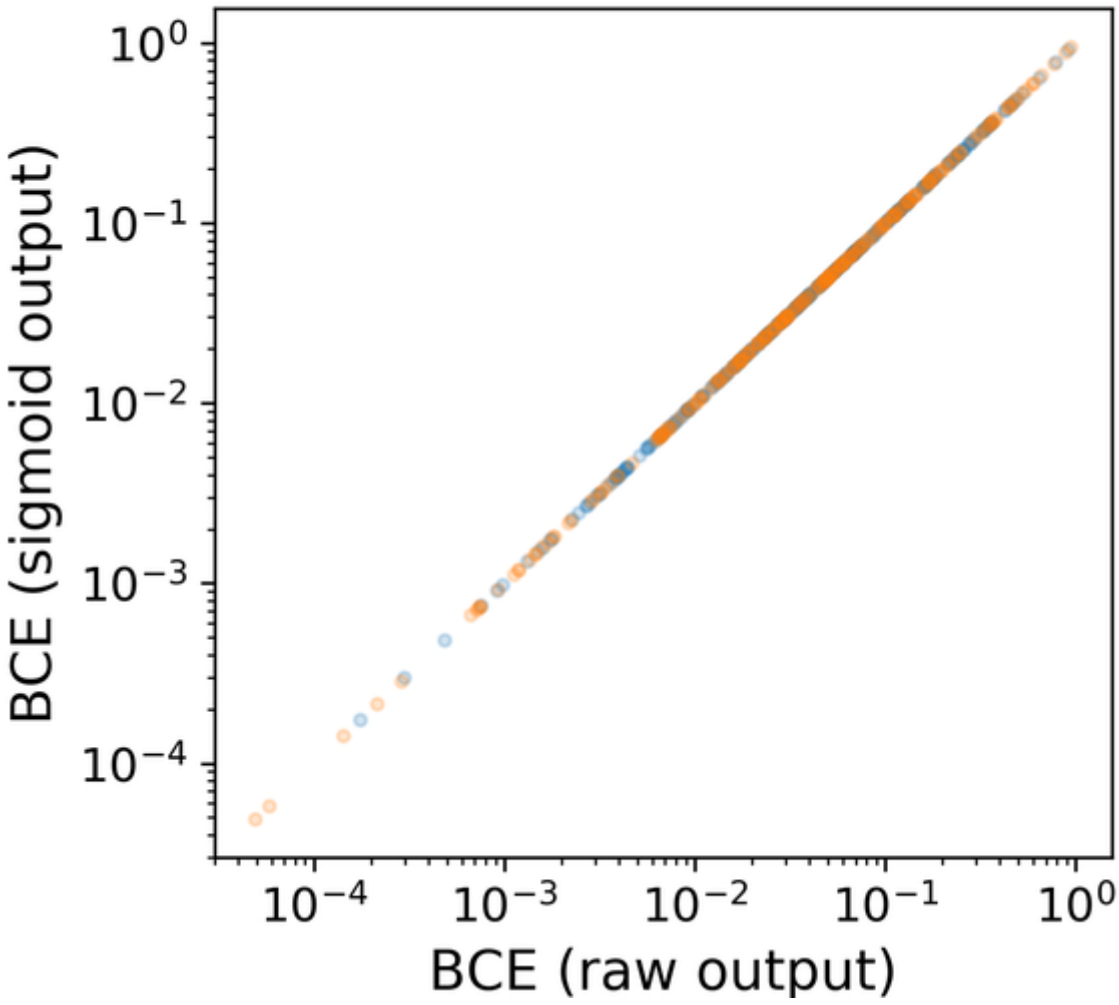


Figure 6: scatter plot of BCE values computed from sigmoid output vs. those computed from raw output of the fully trained network with batch size = 4.

Results were **qualitatively similar with VGG16- and Inception V3-based networks** pretrained on the imagenet data set (trained without fine-tuning of the convolutional parts).

3. Conclusions

First of all, let's reiterate that fears of number under- or overflow due to the combination of sigmoid activation in the last layer and BCE as the loss function are unjustified — in Keras using the TensorFlow backend. I am not familiar with other frameworks (yet), but I would be very surprised if they did not feature similar precautions.

Based on the 'experiments' above with the venerable cats_vs_dogs data set it appears that sigmoid +BCE is fine also in terms of precision. Particularly if you use a reasonable batch size and properly scaled data, it should not matter how BCE is computed. However, this is just one data set and very few models tested on them.

So, my **tentative summary**: If

1. you know or suspect that raw outputs of many of your samples at the last layer attain extreme values, and
2. your batch size is very small, and/or
3. you want to exclude numercial imprecision as a possible (if unlikely) cause of trouble,

it can't harm to compute BCE from raw outputs. Otherwise, just stick with sigmoid+BCE.

Comments, suggestions, experience with other frameworks? Happy to hear about it.

Machine Learning Neural Networks Loss Function Activation Functions Keras