Roc-star : An objective function for ROC-AUC that actually works.

For binary classification. evferybody loves the Area Under the Curve (AUC) metric, but nobody directly targets it in their loss function. Instead folks use a proxy function like Binary Cross Entropy (BCE).

This works fairly well, most of the time. But we're left with a nagging question : could we get a higher score with a loss function closer in nature to AUC?

It's a fair guess that we could. There have been lots of attempts at such a better loss for AUC. (One very common tactic is some form of rank-loss function, e.g. Hinge Rank-Loss.) In practice, however, no clear winner has ever emerged. No serious challenge to the default choice of BCE.

There are even considerations beyond performance. Since BCE is a fundamentally different creature than AUC, BCE tends to misbehave in the final stretch of training where we are trying to steer it toward the highest AUC score.

A good deal of the AUC optimization actually ends up occurring in the tuning of hyper-parameters. Early Stopping becomes an uncomfortable necessity as the model may diverge sharply at any time from its high score.

We'd like a loss function that gives us higher scores and less trouble.

We present such a function here.

The Problem : AUC is bumpy

My favorite working definition of AUC is this: Let's call the binary class labels "Black" (0) and "White" (1). Pick one black element at random and let x be its predicted value. Now pick a random white element with value y. Then,

AUC = the probability that the elements are in the right order. That is, $x<y$.

That's it. For a given set of points like the Training Set, we can get this probability by doing a brute-force calculation. Scan the set of all possible black/white pairs , and count the portion that are right-ordered.

We can easily see that the AUC score is not differentiable (a smooth curve with respect to either *x* or *y*.) Take any element (any color) and move it slightly enough that it doesn't hit a neighboring point. The AUC stays the same. Once the point does cross a neighbor, we have a chance at flipping one of the x<y comparisons and the AUC can instantaneously change. So the AUC makes no smooth transitions.

That's a problem for Neural Nets, where we need a differentiable loss function.

The Search : Ancients and Artifacts.

So we set out to find a *differentiable* function which is close as possible to AUC.

I dug back through the existing literature and found nothing that worked in practice. Finally I came across a curious piece of code that somebody had checked into the TFLearn codebase.

Without fanfare, it promised differentiable deliverance from BCE in the form of a new loss function.

(***Don't try it, it blows up***.) :
http://tflearn.org/objectives/#roc-auc-score

```
#Bad code, do not use.
def roc_auc_score(y_pred, y_true):
  """ ROC AUC Score.
  Approximates the Area Under Curve score, using approximation based on
  the Wilcoxon-Mann-Whitney U statistic.
  Yan, L., Dodier, R., Mozer, M. C., & Wolniewicz, R. (2003).
  Optimizing Classifier Performance via an Approximation to the Wilcoxon-Mann-
  Measures overall performance for a full range of threshold levels.
  Arguments:
```

```
        y_pred: `Tensor`. Predicted values.
        y_true: `Tensor` . Targets (labels), a probability distribution.
    """
    with tf.name_scope("RocAucScore"):
        pos = tf.boolean_mask(y_pred, tf.cast(y_true, tf.bool))
        neg = tf.boolean_mask(y_pred, ~tf.cast(y_true, tf.bool))
.
.
(more bad code)
```

It doesn't work at all. (Blowing up is actually the least of its problems). But it mentions the paper it was based on.

Even though the paper is *ancient*, dating back to 2003, I found that with a little work - some extension of the math and careful coding - it actually works. It's uber-fast, with speed comparable to BCE (and just as vectorizable for a GPU/MPP) *. In my tests, it gives higher AUC scores than BCE, is less sensitive to Learning Rate (avoiding the need for a Scheduler in my tests), and eliminates entirely the need for Early Stopping.

OK, let's turn to the original paper : **Optimizing Classifier Performance via an Approximation to the Wilcoxon-Mann-Whitney Statistic.**

The paper

The authors, *Yan et. al* , motivate the discussion by writing the AUC score in a particular form. Recall our example where we calculate AUC by doing a brute-force count over the set of possible black/white pairs to find the portion that are right-ordered. Let **B** be the set of black values and **W** the set of white values. All possible pairs are the Cartesian Product **B** X **W**. To count the right-ordered pairs we write :

$$\sum_{(x,y) \in \mathbf{B} \times \mathbf{W}} \begin{cases} 1 : x < y \\ 0 : otherwise \end{cases}$$

This is really just straining mathematical notation to say 'count the right-ordered pairs.' If we divide this sum by the total number of pairs , |**B**| * |**W**|, we get exactly the AUC metric.

Historically, this is called the normalized normalized Wilcoxon-Mann-Whitney (WMW) statistic.

To make a loss function from this, we could just flip the x < y comparison to x > y in order to penalize wrong-ordered pairs. The problem, of course, is that discontinuous jump when *x* crosses y .

Yan et. al* surveys and rejects past work-arounds using continuous approximations to the step (Heaviside) function, such as a Sigmoid curve. Then they pull this out of a hat :

$$\sum_{(x,y)\ \in \mathbf{B}\times\mathbf{W}} \begin{cases} (y+\gamma-x)^p \ : \ y+\gamma > x \\ \\ 0 : otherwise \end{cases}$$

As a loss function, we notice that it is differentiable and convex. Forget Γ for the moment, set it to zero. This function first flips the x>y comparison, the same trick we'd use to convert the Wilcoxon-Mann-Whitney formula to a loss instead of an AUC score. If p were 1, the loss would simply be ReLu(y-x). But there's a hiccup : ReLu is not differentiable at 0. That's not much of a problem in ReLu's more accustomed role as an activation function, but for our purposes here the singularity at 0 lands directly on the thing we are interested most in : the times wben black and white points collide (y=x).

Fortunately, raising ReLu to a power fixes this. ReLu^p with p>1 is differentiable everywhere. *(Yann et. al* don't provide a justification for the exponent, they just drop it on us *Deus Ex Machina* style.)

OK, so *p* should be > 1. Now for Γ : Γ provides a 'padding' which is enforced between two points. We penalize not only wrong-ordered pairs, but also right-ordered pairs which are *too close*. If a right-ordered pair is too close, its elements are at risk of getting swapped in the future by the random jiggling of a stochastic neural net. The idea is to keep them moving apart until they reach a comfortable distance.

And that's the basic idea as outlined in the paper. We just have to make some refinements regarding Γ and p.

About that Γ and p

Here we break a bit with the paper. *Yan et. al* seem a little squeamish on the topic of choosing Γ and p, offering only that a $p = 2$ or $p = 3$ seems good and that Γ should be somewhere between 0.10 and 0.70. Yan essentially wishes us luck with these parameters and bows out.

First, we permanently fix $p = 2$, because any self-respecting loss function should be a sum-of-squares. (There are reasons for this that are beyond the scope if this paper. OK, it's a guess. Just go with it.)

Second and more importantly, let's take a look at Γ. The heuristic of 'somewhere from 0.10 to 0.70' looks strange on the face of it; even if the predictions were normalized to be 0<x<1, this guidance seems overbroad, indifferent to the underlying distributions, and just weird.

We're going to define Γ differently so that it is calculated from the training set.

Back to the training set and its Black/White pairs, **B** X **W**. There are |**B**||**W**| pairs in this set. Of these, AUC |**B**| |**W**| are right-ordered. So, the number of wrong-ordered pairs is (1-AUC) |**B**| |**W**|

When Γ is zero, only these wrong-ordered pairs are in motion (have positive loss.) A positive Γ would expand the set of moving pairs to include some pairs which are right-ordered, but too close. Instead of worrying about Γ's numeric value, we'll define what the number of too-close pairs will be,

We define a constant delta which fixes the proportion of too-close pairs to wrong-ordered pairs.

|**too_close_pairs**| = delta |**wrong_ordered_pairs**|

We fix this delta throughout training and update Γ to conform to it. For given delta, calculate Γ such that

|**pairs** where y+Γ>x| = delta |**pairs** where y>x|

In our experiments we found that delta can range from 0.5 to 2.0, and 1.0 is a good default choice.

So we set delta to 1, p to 2, and forget about Γ altogether,

Let's make code

There's a number of reasons that TFLearn code doesn't work; let's list them here so we can avoid these traps

1. On the first iteration, the predictions might all be zero. That gives a loss of zero, and the model is stalled out. To counter this, on the first epoch we use Binary Cross Entropy to get things moving in the right direction.
2. The TFLearn function looks at batches, not the whole training set. While typical for other loss functions, this is a Bad Idea for AUC - especially for unbalanced data where one class is relatively rare. So, our loss function will examine the whole data set. This also avoids TFLearn's problem that the batch may not contain a mix of black and white data points.

But what about performance ? Does this mean that for each prediction we have to scan the whole data set ? And we can't even parallel process the loss in batches?

No. Suppose we are calculating the loss function for a given white data point, *x*.

We can take a random sub-sample of the black data points to use in this calculation. If we set the size of the sub-sample to be, say, 1000 - we get a very (very) close approximation to the loss function which is easily parallelized into one SIMD instruction. (It is also possible to do this by being clever with cumulative sums, but that's a minor refinement we won't bother with.)

It is then no problem to process these calculations in the familar batch-parallel way. The GPU occupancy is higher, but the real-time performance isn't noticably slower than BCE.

Here's the batch-loss function in PyTorch,

Note that there are some extra parameters. We are passing in the training set from the *last epoch*. Since the entire training set doesn't change much from one epoch to the next, the loss function can compare each prediction again a slightly out-of-date training set without too much ****. There's no real reason for this except to simplify debugging, and in our experiments it did no harm and strangely - appeared to perform better.

Similarly, Γ is an expensive calculation. We still use the sub-sampling trick, but increase the size of the sub-samples to ~10,000 to ensure an accurate estimate. To keep performance

clipping along, we recompute this value only once per epoch. Here's the function to do that :

*** gamma function ***

Here's the helicopter view :
Epochs and Batches

Complete working code.

Sample runs.