Unbalanced data and weighted cross entropy

Asked 7 years, 6 months ago Modified 1 year, 4 months ago Viewed 77k times

71

I'm trying to train a network with an unbalanced data. I have A (198 samples), B (436 samples), C (710 samples), D (272 samples) and I have read about the "weighted_cross_entropy_with_logits" but all the examples I found are for binary classification so I'm not very confident in how to set those weights.

Total samples: 1616

A_weight: 198/1616 = 0.12?

The idea behind, if I understood, is to penalize the errors of the majority class and value more positively the hits in the minority one, right?

My piece of code:

```
weights = tf.constant([0.12, 0.26, 0.43, 0.17])
cost = tf.reduce_mean(tf.nn.weighted_cross_entropy_with_logits(logits=pred,
targets=y, pos_weight=weights))
```

I have read this one and others examples with binary classification but still not very clear.

python machine-learning tensorflow deep-learning

Share
Improve this question
Follow

edited Dec 29, 2022 at 3:34

starball
48.3k ● 28 ● 187 ● 848

asked Jun 15, 2017 at 6:51

Sergiodiaz53
1,278 ● 3 ● 15 ● 23

I want to understand why weighted cross entropy came about, what is it for and what exactly is the idea behind it. What are some materials that you've read? I cannot find many resources. – haneulkim Jul 25, 2022 at 3:07

4 Answers

Sorted by: Highest score (default)

96

Note that `weighted_cross_entropy_with_logits` is the weighted variant of `sigmoid_cross_entropy_with_logits`. Sigmoid cross entropy is typically used for *binary* classification. Yes, it can handle multiple labels, but sigmoid cross entropy

basically makes a (binary) decision on each of them -- for example, for a face recognition net, those (not mutually exclusive) labels could be "*Does the subject wear glasses?*", "*Is the subject female?*", etc.

In binary classification(s), each output channel corresponds to a binary (soft) decision. Therefore, the weighting needs to happen within the computation of the loss. This is what `weighted_cross_entropy_with_logits` does, by weighting one term of the cross-entropy over the other.

In mutually exclusive multilabel classification, we use `softmax_cross_entropy_with_logits`, which behaves differently: each output channel corresponds to the score of a class candidate. The decision comes *after*, by comparing the respective outputs of each channel.

Weighting in before the final decision is therefore a simple matter of modifying the scores before comparing them, typically by multiplication with weights. For example, for a ternary classification task,

```
# your class weights
class_weights = tf.constant([[1.0, 2.0, 3.0]])
# deduce weights for batch samples based on their true label
weights = tf.reduce_sum(class_weights * onehot_labels, axis=1)
# compute your (unweighted) softmax cross entropy loss
unweighted_losses = tf.nn.softmax_cross_entropy_with_logits(onehot_labels,
logits)
# apply the weights, relying on broadcasting of the multiplication
weighted_losses = unweighted_losses * weights
# reduce the result to get your final loss
loss = tf.reduce_mean(weighted_losses)
```

You could also rely on `tf.losses.softmax_cross_entropy` to handle the last three steps.

In your case, where you need to tackle data imbalance, the class weights could indeed be inversely proportional to their frequency in your train data. Normalizing them so that they sum up to one or to the number of classes also makes sense.

Note that in the above, we penalized the loss based on the true label of the samples. We could also have penalized the loss based on the *estimated* labels by simply defining

```
weights = class_weights
```

and the rest of the code need not change thanks to broadcasting magic.

In the general case, you would want weights that depend on the kind of error you make. In other words, for each pair of labels `X` and `Y`, you could choose how to

penalize choosing label `X` when the true label is `Y`. You end up with a whole prior weight matrix, which results in `weights` above being a full `(num_samples, num_classes)` tensor. This goes a bit beyond what you want, but it might be useful to know nonetheless that only your definition of the weight tensor need to change in the code above.

Share

Improve this answer

Follow

edited Jun 15, 2017 at 9:53

answered Jun 15, 2017 at 8:54

P-Gn

24.5k ● 10 ● 94 ● 107

Thanks for the explanation, user1735003! It's much more clear now. But your answer led to me a new question: My data is related in the sense that an A should be similar to a B than a C. So I would like to penalize more a false C than a false B. In that case, and related to your last paragraph, Could I have a tensor of (num_classes, num_classes) where I penalize more this kind of misclassification? in that case: the diagonal of the tensor should be the inversely proportional... again or should be different? Thanks in advance. – Sergiodiaz53 Jun 15, 2017 at 10:09

1 It is hard to give a definitive advice for choosing the weights. Even for unbalanced data, using the inverse frequency is sometimes inappropriate, because (for example) the class that represent 99.9% of your data is also very easy to classify. The situation you describe -- unbalanced multilabel data with yet non-uniform priors over confusions -- is even more complicated. You could start a run without weights or standard counterbalancing weights and decide to modify weights based on the confusion matrix you obtain. Unfortunately weights are in the end yet more hyperparameters to tune. – P-Gn Jun 15, 2017 at 14:56

I understand, @user1735003. I was just wondering if you could tell me some references to check? I mean, I don't know even how to start the tune of the weights, if should I increase or decrease one or the complete row (in case I can use a (num_classes, num_classes) matrix). Thank you so much for your help. – Sergiodiaz53 Jun 20, 2017 at 9:01

Is there any way to weight true positives, true negatives, false positives and false negatives in the context of a binary cross entropy? I think I have a problem that is related to this as described here: stackoverflow.com/questions/48744092/… – Nickpick Feb 12, 2018 at 19:10

fyi the function `tf.nn.softmax_cross_entropy_with_logits` has been deprecated in favor of `tf.nn.softmax_cross_entropy_with_logits_v2` – jkr Oct 1, 2018 at 19:36

4

See this answer for an alternate solution which works with sparse_softmax_cross_entropy:

```
import  tensorflow as tf
import numpy as np

np.random.seed(123)
sess = tf.InteractiveSession()

# let's say we have the logits and labels of a batch of size 6 with 5 classes
logits = tf.constant(np.random.randint(0, 10, 30).reshape(6, 5),
dtype=tf.float32)
```

```
labels = tf.constant(np.random.randint(0, 5, 6), dtype=tf.int32)

# specify some class weightings
class_weights = tf.constant([0.3, 0.1, 0.2, 0.3, 0.1])

# specify the weights for each sample in the batch (without having to compute
the onehot label matrix)
weights = tf.gather(class_weights, labels)

# compute the loss
tf.losses.sparse_softmax_cross_entropy(labels, logits, weights).eval()
```

Share

Improve this answer

Follow

edited Jun 11, 2019 at 15:55

answered Nov 12, 2018 at 23:21

DankMasterDan

2,105 ● 4 ● 27 ● 38

1 Upvoted because this answer had -1 votes. I think this answer deserves at least 0 votes because it lead me to discover tf.gather which made my code very efficient, as I had sparse labels, not dense. – AneesAhmed777 Jun 10, 2019 at 10:54

1 `@DankMasterDan`: Link is good to give credit and context, but do copy paste referenced code into your answer so that it is self-sufficient. – AneesAhmed777 Jun 10, 2019 at 10:56

4

Tensorflow 2.0 Compatible Answer: Migrating the Code specified in P-Gn's Answer to 2.0, for the benefit of the community.

```
# your class weights
class_weights = tf.compat.v2.constant([[1.0, 2.0, 3.0]])
# deduce weights for batch samples based on their true label
weights = tf.compat.v2.reduce_sum(class_weights * onehot_labels, axis=1)
# compute your (unweighted) softmax cross entropy loss
unweighted_losses =
tf.compat.v2.nn.softmax_cross_entropy_with_logits(onehot_labels, logits)
# apply the weights, relying on broadcasting of the multiplication
weighted_losses = unweighted_losses * weights
# reduce the result to get your final loss
loss = tf.reduce_mean(weighted_losses)
```

For more information about migration of code from Tensorflow Version 1.x to 2.x, please refer this Migration Guide.

Share Improve this answer Follow

answered Feb 14, 2020 at 11:37

user11530462

0

You can actually keep the categorical crossentropy loss and train using the `class_weight` parameter. The description says:

> Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class. When class_weight is specified and targets have a rank of 2 or greater, either y must be one-hot encoded, or an explicit final dimension of 1 must be included for sparse class labels.

I used it with `total_samples / (2 * class_occurences)` and worked, which is your `weights` list divided by 2, but your `weights` list should also do the trick, just check which value works best for you.

There is a good TF tutorial for dealing with imbalanceddata here.

Share Improve this answer Follow

answered Jul 26, 2023 at 8:59

J Agustin Barrachina
4,040 ● 3 ● 41 ● 66