formula for binary cross entropy with l2 regularization

The formula for binary cross entropy with L2 regularization is:

$J(\theta) = - \frac{1}{m} \sum\_{i=1}^m [y^{(i)} \log h\_\theta(x^{(i)}) + (1-y^{(i)})\log(1-h\_\theta(x^{(i)}))] + \frac{\lambda}{2m} \sum\_{j=1}^n \theta\_j^2$

where:

$J(\theta)$ is the cost function that we want to minimize,

$m$ is the number of training examples,

$y^{(i)}$ is the true label of the $i$-th training example,

$h\_\theta(x^{(i)})$ is the predicted output for the $i$-th training example,

$n$ is the number of parameters in the model,

$\theta\_j$ is the $j$-th parameter,

$\lambda$ is the regularization parameter (also called the regularization strength or the regularization coefficient).

The first term in the formula is the binary cross entropy loss function, which measures the difference between the predicted output and the true label. The second term is the L2 regularization term, which penalizes large values of the parameters to prevent overfitting. The regularization strength is controlled by the parameter $\lambda$, which is a hyperparameter that needs to be tuned.

which one is correct

import tensorflow as tf

def binary\_crossentropy\_with\_l2\_regularization(y\_true, y\_pred, l2\_lambda):

loss = tf.keras.losses.binary\_crossentropy(y\_true, y\_pred)

regularization = tf.reduce\_sum([tf.nn.l2\_loss(var) for var in tf.trainable\_variables()])

return loss + l2\_lambda \* regularization

or

def binary\_crossentropy\_with\_l2(y\_true, y\_pred, l2\_reg=0.01):

cross\_entropy = tf.keras.losses.binary\_crossentropy(y\_true, y\_pred)

l2\_loss = l2\_reg \* tf.reduce\_sum([tf.nn.l2\_loss(var) for var in tf.trainable\_variables()])

loss = cross\_entropy + l2\_loss

return tf.keras.backend.mean(loss)

Both implementations are correct and achieve the same purpose of applying L2 regularization to binary cross-entropy loss. The first implementation calculates the L2 regularization term as a separate step and adds it to the binary cross-entropy loss, while the second implementation combines the two terms into a single loss value. The second implementation also includes a default value of 0.01 for the L2 regularization coefficient, whereas the first implementation requires this value to be explicitly passed as an argument.

EXAMPLE

import tensorflow as tf

def binary\_crossentropy\_with\_l2\_regularization(y\_true, y\_pred, l2\_lambda):

loss = tf.keras.losses.binary\_crossentropy(y\_true, y\_pred)

regularization = tf.reduce\_sum([tf.nn.l2\_loss(var) for var in tf.trainable\_variables()])

return loss + l2\_lambda \* regularization

# define the model

model = tf.keras.Sequential([...])

# compile the model

model.compile(optimizer='adam', loss=binary\_crossentropy\_with\_l2\_regularization, metrics=['accuracy'])

# train the model

model.fit(x\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(x\_val, y\_val))

binary cross entropy with l2 regularization and lovasz loss python function

Here's an implementation of a function that combines binary cross-entropy with L2 regularization and Lovasz loss:

scss

Copy code

import tensorflow as tf

import keras.backend as K

def binary\_crossentropy\_with\_l2\_and\_lovasz(y\_true, y\_pred, l2\_lambda=0.01, smooth=1):

bce\_loss = K.mean(K.binary\_crossentropy(y\_true, y\_pred), axis=-1)

l2\_loss = l2\_lambda \* K.sum([K.sum(K.square(w)) for w in model.trainable\_weights])

lovasz\_grad = tf.expand\_dims(y\_true - y\_pred, axis=-1)

lovasz\_loss = lovasz\_hinge(lovasz\_grad, y\_true, smooth=smooth)

return bce\_loss + l2\_loss + lovasz\_loss

def lovasz\_hinge(logits, labels, per\_image=True, ignore=None, smooth=1.0):

if per\_image:

loss = tf.stack([lovasz\_hinge\_flat(\*flatten\_binary\_scores(logit, label, ignore), smooth=smooth)

for logit, label in zip(logits, labels)], axis=0)

return tf.reduce\_mean(loss)

else:

return lovasz\_hinge\_flat(\*flatten\_binary\_scores(logits, labels, ignore), smooth=smooth)

def lovasz\_hinge\_flat(logits, labels, smooth=1.0):

if len(labels) == 0:

return tf.constant(0.0)

signs = 2.0 \* labels - 1.0

errors = 1.0 - signs \* logits

errors\_sorted, perm = tf.nn.top\_k(errors, k=tf.shape(errors)[0], name="descending\_sort")

gt\_sorted = tf.gather(signs, perm)

grad = lovasz\_grad(gt\_sorted, smooth)

loss = tf.tensordot(tf.nn.relu(errors\_sorted), tf.stop\_gradient(grad), 1, name="loss\_non\_void")

return loss

def lovasz\_grad(gt\_sorted, smooth=1.0):

gts = tf.reduce\_sum(gt\_sorted, axis=0)

intersection = gts[:-1] \* tf.cumsum(gt\_sorted[:-1], axis=0)

union = gts[:-1] + tf.cumsum(1.0 - gt\_sorted[:-1], axis=0)

jaccard = (intersection + smooth) / (union + smooth)

jaccard = tf.concat([[0], jaccard], axis=0)

return jaccard - tf.reduce\_mean(jaccard)

def flatten\_binary\_scores(y\_pred, y\_true, ignore=None):

if ignore is None:

ignore = []

y\_true, y\_pred = np.asarray(y\_true, dtype=np.bool), np.asarray(y\_pred, dtype=np.float32)

assert y\_true.shape == y\_pred.shape, (y\_true.shape, y\_pred.shape)

assert y\_true.ndim == 1, (y\_true.ndim, y\_pred.ndim)

y\_true = y\_true[np.invert(np.isnan(y\_pred))]

y\_pred = y\_pred[np.invert(np.isnan(y\_pred))]

assert y\_true.shape == y\_pred.shape, (y\_true.shape, y\_pred.shape)

pred = y\_pred.ravel()

labels = y\_true.ravel()

if len(ignore) > 0:

mask = np.logical\_not(np.isin(labels, ignore))

labels = labels[mask]

pred = pred[mask]

return pred, labels

You can call this function in your training code as follows:

python

Copy code

model.compile(optimizer=Adam(lr=starting\_learning\_rate), loss=binary\_crossentropy\_with\_l2\_and\_lovasz, metrics=[jacc\_coef

is Adam is helped in back propagation

Yes, Adam optimization algorithm helps in backpropagation. In fact, Adam is one of the most popular optimization algorithms used in deep learning for backpropagation. Adam combines the benefits of two other optimization algorithms, Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp), to improve the efficiency and speed of the backpropagation process.

During backpropagation, Adam adjusts the learning rate for each weight parameter based on the previous gradients and the moving average of the squared gradients. This adaptive learning rate helps Adam to converge faster and

mananshethia24@gmail.com

is Adam is helped in back propagation

Adam is an optimization algorithm commonly used in deep learning, and it is a variant of stochastic gradient descent (SGD). The purpose of optimization algorithms is to update the model parameters during training, based on the calculated gradients, to minimize the loss function.

Therefore, Adam (like any optimization algorithm) plays a crucial role in the backpropagation process because it determines how the gradients are used to update the model parameters. Specifically, Adam adapts the learning rate for each parameter individually based on the estimate of the first and second moments of the gradients, which can lead to more efficient and effective updates compared to traditional SGD.

So, in short, Adam (like any optimization algorithm) is essential for the backpropagation process and helps to update the model parameters based on the calculated gradients.