Keras loss functions custom

I'm looking for a way to create a loss function that looks like this: the function should then be increased for consideration. Is it possible to reach Keras? All the suggestions as this can be achieved are greatly appreciated. def special_loss_function (y_true, y_pred, reward_if_correct, def special_loss_function (y_pred, reward_if_correct, punishment_if_false): Loss = if binary classification is correct to apply the reward for this training item under weight, if binary classification is incorrect, apply a penalty for this training item under weight) return K.mean (loss, ass = -1) The approach I've been looking for in my example is to put the weight together with y_true and then reduce the tensor into two, separate weight and y_true as shown below. Is an approach like this possible at all, or does it interfere with the normalization process etc? def decompose_y_true (y_true_and_weights): y_true = y_true_and_weights [:,1] weights = y_true_and_weights [:, 1:] return y_true, weights def custom_loss (y_true_and_weights, y_pred): y_true, y_weights = decompose_y_true (y_true_and_weights) loss = # some loss action return K.mean (loss, ass = -1) Much more elegant would be if I could go in my weight over the sample_weights parameter fit function, but there seems to be some limits to what shape these scales can have, and there's no way to get them a function loss as far as I can tell. Is there any way to put this into a loss function somehow so I can run on them from there? Photo Dhruv Deshmukh on UnsplashWe use loss functions to calculate how well a given algorithm fits the data it is trained. The loss account is based on the difference between forecasted and actual values. If the predicted values are far from the actual values, the loss function will produce a very large number. Keras is a library for the creation of neural networks. It is open source and written in Python. Keras does not support low-level calculation, but it works above libraries like Theano and TensorFlow.In this tutorial, we will use TensorFlow as keras backend. Back-end is a Keras library used to perform calculations such as tensors products, convolutions, and other similar activities. Photo Karim MANJRA on UnsplashCommonly uses the loss features kerasAs above, we can create a custom loss function of our own; but before that, it's good to talk about the existing, ready-made loss features available to Keras. Below are the two most commonly used ones. LOGCOSHLogcosh is a prediction error in the hyperbolic cosinous logarithm. logcosh in general is similar to the mean square error described below, but it is not strongly influenced by incorrect predictions. The MSEMSE or mean squared error is similar to the logcosh as described above, however, MSE measures the average of error squares. It is calculated as the mean squared difference predicted values and actual value. MaeMa absolute error or MAE is an MSE variant and is a measure of the difference between two continuous variables, usually referred to as x1 and y1. The mean absolute error is the mean of absolute error = y1-x1, where y1 is the predicted value and x1 is the actual value. The loss calculation formula is defined differently for different loss functions. In some cases, we may need to use a loss calculation formula that is not provided on the fly with Keras. In this case, we may consider defining and using our loss function. This type of user-defined loss function is called a custom loss function. The Keras loss function can improve the performance of the machine learning model the way we want, and can be very useful in solving specific problems more effectively. For example, imagine we are building a model for optimising the stock portfolio. In this case, it will be useful to develop a custom loss function that implements a large penalty for forecasting price fluctuations in the wrong direction. We can create a custom loss function in Keras by typing a function that returns a scalar and accepts two arguments: namely, fair value and predicted value. Then we move the custom loss function to model.compile as a parameter like we did with any other loss function. Let us implement it! Now let's implement a custom loss feature in our Kera model. As a first step, we need to determine our Kera model. The name of our model instance keras_model, and we use kera's sequential() function to create the model. We have included three layers, all dense layers with the shape 64, 64 and 1. We have input form 1, and we use the ReLU activation function (rectified linear unit). Once the model is defined, we need to determine our order loss function. It is implemented as shown below. We go the actual and expected value to this feature. Note that we share the difference between the actual and expected value by 10, which is the custom part of our loss function. In the default loss function, the difference between actual and predicted values is not divided by 10.Remember — it all depends on your specific use of what kind of custom loss function you will need to write. Note that here we share 10, which means that we want to reduce our loss during the calculation. In the default mse scenario, the loss amount will be 10 times greater than this custom implementation. So we can use this type of custom loss feature when the value of our losses becomes very high and the calculations become expensive. Here we return scalar custom loss values from this function. Defining a custom loss function kerasTo use our custom loss function below, we need determine our optimizer. We are going to use RMSProp Optimizer here. RMSprop RMSprop RMSprop the gst mean square. The RMSprop Optimizer is similar to a gradient descent with pulse. Commonly used optimizers are named as rmsprop, Adam, and sgd. We need to put the custom loss function as well as optimizer compilation method, called model instance. Then we print out the model to make sure there is no error in compiling. Keras model optimizer and collect modelNow it's time to train the model and see that it works without errors. To do this, we use a fit method model and put independent variable x and dependent variable y, along with eras = 100.The goal here is to see that the model is a tutorial without errors and that the loss gradually decreases as the number of era increases. You can view the model training leading to the image below. Keras model tutorial with era = 100End notesThis article we learned what the custom loss function is and how to define one keras model. Then we compiled the Kera model using a custom loss feature. Finally, we were able to successfully train the model by introducing a custom loss function. Thanks for reading! Editor's Note: Heartbeat is a collaborative online publication and community designed to explore the new intersection of mobile app development and machine learning. We are committed to supporting and inspiring developers and engineers from all walks of life. Editorially independent, Heartbeat is sponsored and published by Fritz AI, a machine learning platform that helps developers teach devices to see, hear, sense and think. We pay our investors and we don't sell ads. If you want to contribute, head over to our call to investors. You can also sign up to receive our weekly newsletters (Deep Learning Weekly and Fritz AI Newsletter), join us at Slack, and follow Fritz AI on Twitter for all the latest mobile machine learning. The purpose of the loss functions is to calculate the amount that the model should try to reduce during training. Available losses Note that all losses are available through both the class handle and the function handle. Class handles allow you to pass configuration arguments to a constructor (for example, loss_fn = CategoricalCrossentropy(from_logits=True)), and they perform a reduction by default when used in a standalone way (see details below). Probabilistic loss Regression losses Viru losses maximum margin classification losses Loss of loss Loss Of loss Use of losses with compilation() & fit() A loss function is one of the two arguments necessary for compiling the Kera model: import keras from tensorflow.keras import layer model = keras. Sequential() model.add (layers. Dense(64, kernel_initializer =uniform, input_shape =(10,))) model.add(layers. Activation(softmax)) loss_fn = model.compile(loss=loss_fn, optimizer='adam') All built-in loss functions can also be passed using their string identifier: # pass optimizer by name: Default parameters will be used for model.compile(loss='sparse_categorical_crossentropy', optimizer='adam') Loss functions are usually generated by instilling loss class (e.g. keras.losses.SparsegoricalCrossenCrotropy). All losses are also provided as function handles (e.g. keras.losses.sparse_categorical_crossentropy). Using class lets put configuration arguments during the moment, such as: loss_fn = keras.losses.SparseCategoricalCrossentropy (from_logits = True) Standalone usage loss is callable with arguments loss_fn (y_true, y_pred, sample_weight = None): y_true: Earth values, shapes (batch_size, d0, ... (dN). For rare loss features such as a rare categorical cross, the shape would be (batch_size, d0, ... dN-1) y_pred: shapes (batch_size, d0, .. dN estimated values). sample_weight: sample_weight shall, optionally, act as a reduction weighting factor for losses of each sample. If a scalar is provided, the loss is simply scaled by the given value. If sample_weight the value of tenos [batch_size], the total loss of each sample of the batch shall be recalculated with the corresponding element in sample_weight vector. If the sample_weight (batch_size, d0, ... dN-1) (or can be transmitted to this form), then each y_pred loss element is scaled with the corresponding sample_weight. (Note to ondN-1: all loss functions are reduced by 1 dimension, usually axis = -1.) By default, loss functions return one scalar loss value for each input sample, for example >>> tf.keras.losses.mean_squared_error(tf.ones((2,2,)), tf.zeros((2,2))), However, in loss class cases, there is <tf. Tensor: shape=(2,), dtype=float32, numpy=array([1st, 1st],=dtype=float32)>reduction constructor argument, which by default is sum_over_batch_size (i.e. average). The permissible values are sum_over_batch_size, sum and none: sum_over_batch_size means that the event of loss will return the mean of the lot losses per sample. amount is a loss instance that will return the amount of lot losses per sample. does not mean that the loss instance will return the entire sample loss array. >>> loss_fn = tf.keras.losses.MeanSquaredError(reduction='sum_over_batch_size') >>> loss_fn(tf.ones((2,2,)), tf.zeros((2,2))) >>> loss_fn <tf. Tensor: shape=(), dtype=float32, numpy=1.0>= tf.keras.losses.MeanSquaredError(reduction='sum') >>> loss_fn(tf.ones((2,2,)), tf.zeros((2,2)) >>loss_fn(tf.ones((2, 2,)), tf.zeros((2, 2)) <tff. Tensor: shape=(), dtype=float32, numpy=2.0>>>>> loss_fn = tf.keras.losses.MeanSquaredError(reduction='none') >>>> loss_fn(tf.ones((2,2,)), tf.zeros((2,2))) Note that this is an important difference between loss features such as tf.keras.losses.mean_squared_error and default <tf. Tensor: shape=(2,), numpy=array([1, 1.],= dtype=float32)>loss class instances such as tf.keras.losses.mean_squared_error and default loss class instances, such as tf.keras.losses.mean_squared_error and default loss class instances, for example, tf.keras.losses.mean_squared_error and default loss class instances, such as tf.keras.losses.mean_squared_error and default loss class instances, such as tf.keras.losses.mean_squared_error and default loss class instances, such as tf.keras.losses.mean_squared_error and default loss class instances, such as tf.keras.losses.mean_squared_error and default loss class instances such as tf.keras.losses.mean_squared_error and default loss class instances such as tf.keras.losses.mean_squared_error and default loss class instances, such as tf.keras.losses.mean_squared_error and default loss class instances such as tf.keras.losses.mean_squared_error and default loss class instances, such as tf.keras.losses.mean_squared_error and default loss class instances, such as tf.keras.losses.mean_squared_error and default loss class instances, such as tf.keras.losses.mean_squared_error and default loss class instances, such as tf.keras.losses.mean_squared_error and default loss class instances, such as,</tf. Tensor:> </tf. Tensor:> </tf. Tensor:> </tf. Tensor:> </tf. Tensor:> the version of the function does not reduce, but by default, the class instance is used. >>> loss_fn = tf.keras.losses.mean_squared_error >>> loss_fn(tf.ones((2, 2,)), tf.zeros((2, 2))) <tf. Tensor: shape=(2,), dtype=float32, numpy=array([1,1.],= dtype=float32)>>>>>>loss_fn = tf.keras.losses.MeanSquaredError() >>>>loss_fn(tf.ones((2,2,)), tf.zeros((2, 2))<tff. Tensor: shape=(), dtype=float32, numpy=1.0>Using fit(), this difference is not significant because the reduction is made by the system. Here's how you could use the loss class, for example, as part of a simple learning loop: loss_fn = tf.keras.losses.CategoricalCrossentropy (from_logits =True) Optimizer = tf.keras.optimizers.Adam() # Iterate over dataset lots. for X,y Data Set: With tf. GradientTape() as ribbon: logits = model(x) # Calculate the loss value of this batch. loss_value = loss_fn(y, logits) # Update model scales to reduce loss value. gradients = tape.gratient(loss_value, model.trainable_weights) optimizer.apply_gradients (zip (gradients, model.trainable_weights)) Creating custom loss Any callable with signature loss_fn (y_true, y_pred) that returns an array of losses (one of the sample input lots) can be put to compile () as loss. Note that sample weighting is automatically supported for such loss. Here is a simple example: def my_loss_fn (y_true, y_pred): squared_difference = tf.square (y_true - y_pred) returns tf.reduce_mean(squared_difference, axis = -1) # Note 'axis = -1 ' model.compile(optimizer = adam, loss = my_loss_fn) add_loss API Loss functions that are used for model output are not the only way to cause damage. As you type in a custom layer or subclassized model call method, you may want to calculate the scalar quantities that you want to reduce during training (for example, regularization losses). You can use add_loss() layer method to follow the following loss conditions. Here's an example of a layer that adds a sparsity regularization loss based on L2 norm input: from tensorflow.keras.layers to import Layer class MyActivityRegularizer (Layer): Layer that causes operational vim regularization loss. def __init__ (self, rate = 1e-2): super (MyActivityRegularizer, self) .__init__() self.rate = rate def call (self, raw): # We use add_loss to cause regularization loss # which depends on the input. self.add_loss (self.rate * tf.reduce_sum (tf.square (inputs))) returns input loss values added by using add_loss that can be retrieved in the .losses list property of any layer or model (they are recursively retrieved from each basic level): from tensflow.keras import layers to the SparseMLP(): Layer Stackar liner layers with vim regularity loss. def __init__ (self, output_dim): super (SparseMLP, self.dense_1 = slāņi. Blīvs (32, activation = tf.nn.relu) self.regularization = MyActivityRegularizer (1e-2)</tf. Tensor:> </tf. Tensor:> </tf. Tensor:> = layers. Dense(output_dim) def call (self, input): x = self.dense_1 (input) x = self.regularization (x) return self.dense_2(x) mlp = SparseMLP(1) y = mlp (tf.ones ((10, 10))) print (mlp.losses) # List containing one float32 scalar These losses are cleared by the top layer at the beginning of each front pass - they do not accumulate. So layer.losses always contain only losses incurred in the last forward pass. Typically, these losses are used by summing them before calculating gradients by writing a training loop. # Losses correspond * last * front pass mlp = SparseMLP (1) mlp (tf.ones ((10, 10))) defend len (mlp.losses) == 1 mlp (tf.ones ((10, 10))) defend len (mlp.losses) == 1 # No accumulation. Using model.fit(), such loss terms are processed automatically. When writing a custom training loop, you should retrieve these terms manually from model.losses such as: loss_fn = tf.keras.losses.CategoricalCrossentropy (from_logits = True) Optimizer = tf.keras.optimizers.Adam() # Iterate over dataset batches. for X,y Data Set: With tf. GradientTape() as ribbon: # Forward pass. loss_value = loss_fn(y, logits) # Add additional loss in terms of loss value. loss_value += sum(model.losses) # Update model scales to reduce loss value. gradient = tape.gradient (loss_value, model.trainable_weights) optimizer.apply_gradients (zip (gradients, model.trainable_weights)) For more information, see add_loss() documentation. Details.