what is a loss function?

In deep learning, the loss is computed to get the gradients with respect to model weights and update those weights accordingly via back propagation. Loss is calculated and the network is updated after every iteration until model updates don’t bring any improvement in the desired evaluation metric.

Let’s get into it!

**Keras loss functions**

In Keras, loss functions are passed during the compile stage, as shown below.

we’re defining the loss function by creating an instance of the loss class. Using the class is advantageous because you can pass some additional parameters.

**from** tensorflow **import** keras

**from** tensorflow.keras **import** layers

model = keras.Sequential()

model.add(layers.Dense(64, kernel\_initializer='uniform', input\_shape=(10,)))

model.add(layers.Activation('softmax'))

loss\_function = keras.losses.SparseCategoricalCrossentropy(from\_logits=**True**)

model.compile(loss=loss\_function, optimizer='adam')

If you want to use a loss function that is built into Keras without specifying any parameters you can just use the string alias as shown below:

model.compile(loss='sparse\_categorical\_crossentropy', optimizer='adam')

You might be wondering how does one decide on which loss function to use?

There are various loss functions available in Keras. Other times you might have to implement your own custom loss functions.

Let’s dive into all those scenarios.

**Which loss functions are available in Keras?**

**Binary Classification**

Binary classification loss function comes into play when solving a problem involving just two classes. For example, when predicting fraud in credit card transactions, a transaction is either fraudulent or not.

**Binary Cross Entropy**

The Binary Cross entropy will calculate the cross-entropy loss between the predicted classes and the true classes. By default, the *sum\_over\_batch\_size* reduction is used. This means that the loss will return the average of the per-sample losses in the batch.

y\_true = [[0., 1.], [0.2, 0.8],[0.3, 0.7],[0.4, 0.6]]

y\_pred = [[0.6, 0.4], [0.4, 0.6],[0.6, 0.4],[0.8, 0.2]]

bce = tf.keras.losses.BinaryCrossentropy(reduction='sum\_over\_batch\_size')

bce(y\_true, y\_pred).numpy()

The sum reduction means that the loss function will return the sum of the per-sample losses in the batch.

bce = tf.keras.losses.BinaryCrossentropy(reduction='sum')

bce(y\_true, y\_pred).numpy()

Using the reduction as none returns the full array of the per-sample losses.

bce = tf.keras.losses.BinaryCrossentropy(reduction='none')

bce(y\_true, y\_pred).numpy()

array([0.9162905 , 0.5919184 , 0.79465103, 1.0549198 ], dtype=float32)

In binary classification, the activation function used is the sigmoid activation function. It constrains the output to a number between 0 and 1.

**Multiclass classification**

Problems involving the prediction of more than one class use different loss functions. In this section we’ll look at a couple:

**Categorical Crossentropy**

The CategoricalCrossentropy also computes the cross-entropy loss between the true classes and predicted classes. The labels are given in an *one\_hot* format.

cce = tf.keras.losses.CategoricalCrossentropy()

cce(y\_true, y\_pred).numpy()

**Sparse Categorical Crossentropy**

If you have two or more classes and  the labels are integers, the SparseCategoricalCrossentropy should be used.

y\_true = [0, 1,2]

y\_pred = [[0.05, 0.95, 0], [0.1, 0.8, 0.1],[0.1, 0.8, 0.1]]

scce = tf.keras.losses.SparseCategoricalCrossentropy()

scce(y\_true, y\_pred).numpy()

**Regression**

In regression problems, you have to calculate the differences between the predicted values and the true values but as always there are many ways to do it.

**Mean Squared Error**

The MeanSquaredError class can be used to compute the mean square of errors between the predictions and the true values.

y\_true = [12, 20, 29., 60.]

y\_pred = [14., 18., 27., 55.]

mse = tf.keras.losses.MeanSquaredError()

mse(y\_true, y\_pred).numpy()

Use Mean Squared Error when you desire to have large errors penalized more than smaller ones.

**Mean Absolute Percentage Error**

The mean absolute percentage error is computed using the function below.

https://latex.codecogs.com/gif.latex?%5Clarge%20loss%20%3D%20100%20*%20abs%28y_%7Btrue%7D%20-%20y_%7Bpred%7D%29%20/%20y_%7Btrue%7D

It is calculated as shown below.

y\_true = [12, 20, 29., 60.]

y\_pred = [14., 18., 27., 55.]

mape = tf.keras.losses.MeanAbsolutePercentageError()

mape(y\_true, y\_pred).numpy()

**Cosine Similarity Loss**

If your interest is in computing the cosine similarity between the true and predicted values, you’d use the CosineSimilarity class. It is computed as:

https://latex.codecogs.com/gif.latex?%5Clarge%20loss%20%3D%20-sum%28l2_%7Bnorm%7D%28y_%7Btrue%7D%29%20*%20l2_%7Bnorm%7D%28y_%7Bpred%7D%29%29

**The result is a number between  -1 and 1**. 0 indicates orthogonality while values close to -1 show that there is great similarity.

y\_true = [[12, 20], [29., 60.]]

y\_pred = [[14., 18.], [27., 55.]]

cosine\_loss = tf.keras.losses.CosineSimilarity(axis=1)

cosine\_loss(y\_true, y\_pred).numpy()