Using Loss Functions

```
model.compile(loss='mse', optimizer='sgd')
```

or

```
from tensorflow.keras.losses import mean_squared_error
model.compile(loss=mean_squared_error, optimizer='sgd')
```

You've likely seen a lot of loss functions while you've been working in tensorflow

The loss function is called usually when specified as a parameter in model.compile().

Now the last function itself can be declared using either a string with its name, such as MSE here, which stands from mean squared error. Or you can use a loss object.

Using Loss Functions

```
model.compile(loss='mse', optimizer='sgd')
```

or

```
from tensorflow.keras.losses import mean_squared_error
model.compile(loss=mean_squared_error(param=value),
optimizer='sgd')
```

The important difference with using the loss object is that you can add parameters to the object call on.

This means that you can have much better flexibility to do things like tuning your hyper parameters.

Creating a custom loss function

```
def my_loss_function(y_true, y_pred):
    return losses
```

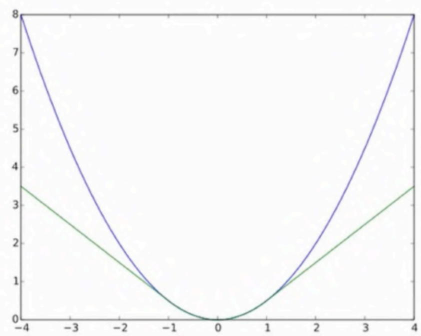
Example

Huber Loss

$$L_{\delta}(a)\begin{cases}\frac{1}{2}a^2 & for|a| \leq \delta \\ \delta \times (|a| - \frac{1}{2}\delta) & \text{otherwise}\end{cases}$$

Threshold

Error




```python
def my_huber_loss(y_true, y_pred):
    threshold = 1
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= threshold
    small_error_loss = tf.square(error) / 2
    big_error_loss = threshold * (tf.abs(error) - (0.5 * threshold))
    return tf.where(is_small_error, small_error_loss, big_error_loss)
```

$$L_{\delta}(a)\begin{cases}\frac{1}{2}a^2 & for|a| \leq \delta \\ \boxed{\delta} \times (|a| - \frac{1}{2}\delta) & \text{otherwise}\end{cases}$$

```python
def my_huber_loss(y_true, y_pred):
    threshold = 1
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= threshold
    small_error_loss = tf.square(error) / 2
    big_error_loss = threshold * (tf.abs(error) - (0.5 * threshold))
    return tf.where(is_small_error, small_error_loss, big_error_loss)
```

$$L_\delta(a) \begin{cases} \frac{1}{2}\boxed{a}^2 & for|\boxed{a}| \le \delta \\ \delta \times (|\boxed{a}| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

```python
def my_huber_loss(y_true, y_pred):
    threshold = 1
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= threshold
    small_error_loss = tf.square(error) / 2
    big_error_loss = threshold * (tf.abs(error) - (0.5 * threshold))
    return tf.where(is_small_error, small_error_loss, big_error_loss)
```

$$L_\delta(a) \begin{cases} \frac{1}{2}a^2 & \boxed{for|a| \le \delta} \\ \delta \times (|a| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

```python
def my_huber_loss(y_true, y_pred):
    threshold = 1
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= threshold
    small_error_loss = tf.square(error) / 2
    big_error_loss = threshold * (tf.abs(error) - (0.5 * threshold))
    return tf.where(is_small_error, small_error_loss, big_error_loss)
```

$$L_\delta(a) \begin{cases} \boxed{\frac{1}{2}a^2} & \boxed{for|a| \le \delta} \\ \delta \times (|a| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

```python
def my_huber_loss(y_true, y_pred):
    threshold = 1
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= threshold
    small_error_loss = tf.square(error) / 2
    big_error_loss = threshold * (tf.abs(error) - (0.5 * threshold))
    return tf.where(is_small_error, small_error_loss, big_error_loss)
```

$$L_\delta(a) \begin{cases} \frac{1}{2}a^2 & for|a| \le \delta \\ \boxed{\delta \times (|a| - \frac{1}{2}\delta)} & \leftarrow \boxed{\text{otherwise}} \end{cases}$$

```python
def my_huber_loss(y_true, y_pred):
    threshold = 1
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= threshold
    small_error_loss = tf.square(error) / 2
    big_error_loss = threshold * (tf.abs(error) - (0.5 * threshold))
    return tf.where(is_small_error, small_error_loss, big_error_loss)
```

Boolean to check   Value if True   Value if False

$$L_\delta(a) \begin{cases} \frac{1}{2}a^2 & for|a| \le \delta \\ \delta \times (|a| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

```python
model = tf.keras.Sequential([keras.layers.Dense(units=1, input_shape=[1])])
model.compile(optimizer='sgd', loss='my_huber_loss')



def my_huber_loss(y_true, y_pred):
    threshold = 1
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= threshold
    small_error_loss = tf.square(error) / 2
    big_error_loss = threshold * (tf.abs(error) - (0.5 * threshold))
    return tf.where(is_small_error, small_error_loss, big_error_loss)
```

Adding hyperparameters to custom loss functions

```
def my_huber_loss(y_true, y_pred):
    threshold = 1
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= threshold
    small_error_loss = tf.square(error) / 2
    big_error_loss = threshold * (tf.abs(error) - (0.5 * threshold))
    return tf.where(is_small_error, small_error_loss, big_error_loss)
```

The first line sets the threshold to one. But you can modify that to see if the loss function can behave better for your data.

As the threshold is used a lot within the function. It feels like it could be a very powerful hyperparameter for us to tune. Thus, if we made it a parameter that we pass into the function, I think that would be a great way to go.

To do this, you can create a wrapper function that contains the original loss function.

In this case, I've created my huber loss with threshold that accepts a threshold parameter and you can see all of the other code is within that.

This accepts the threshold parameter so that when you call it, you can pass in that value. That threshold then gets used within the function instead of the hard-coded one.

```
def my_huber_loss_with_threshold(threshold):
    def my_huber_loss(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) <= threshold
        small_error_loss = tf.square(error) / 2
        big_error_loss = threshold * (tf.abs(error) - (0.5 * threshold))
        return tf.where(is_small_error, small_error_loss, big_error_loss)
    return my_huber_loss
```

The inner function then gets returned by the outer function. If you call my huber loss with threshold, you'll actually get a loss-function back that implements that threshold.

```
def my_huber_loss_with_threshold(threshold):
    def my_huber_loss(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) <= threshold
        small_error_loss = tf.square(error) / 2
        big_error_loss = threshold * (tf.abs(error) - (0.5 * threshold))
        return tf.where(is_small_error, small_error_loss, big_error_loss)
    return my_huber_loss
```

Now if you want to use the threshold, you call the outer function MyHuberLoss with threshold, which can accept the threshold parameter and then returns a reference to a customize my huber loss function, where the threshold is equal to the chosen parameter.

Notice that this is why we introduced the threshold parameter using a wrapper function rather than just modifying the my huber loss function to take in that third parameter for the threshold.

The model.compile parameter loss expects a function that takes in just y_true and y_pred. By including the threshold by using a wrapper, you can still provide a loss function that just gives the expected parameters y_true and y_pred. Now if you want to do some hyperparameter tuning within the loss function, you can tweak that threshold.

```
model.compile(
    optimizer='sgd', loss=my_huber_loss_with_threshold(threshold=1))
```

```
from tensorflow.keras.losses import Loss

class MyHuberLoss(Loss):
  threshold = 1
  def __init__(self, threshold):
    super().__init__()
    self.threshold = threshold

  def call(self, y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= self.threshold
    small_error_loss = tf.square(error) / 2
    big_error_loss = self.threshold * (tf.abs(error) - (0.5 * self.threshold))
    return tf.where(is_small_error, small_error_loss, big_error_loss)
```

This syntax means that MyHuberLoss inherits from Loss. This lets use use it as a loss

```
from tensorflow.keras.losses import Loss

class MyHuberLoss(Loss):
  threshold = 1
  def __init__(self, threshold):
    super().__init__()
    self.threshold = threshold

  def call(self, y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= self.threshold
    small_error_loss = tf.square(error) / 2
    big_error_loss = self.threshold * (tf.abs(error) - (0.5 * self.threshold))
    return tf.where(is_small_error, small_error_loss, big_error_loss)
```

Then within a class, you should have two functions. An __init__() function that initializes a function from the class.

The second is the call function that gets executed when an object is instantiated from a class.

The __init__() function gets the threshold, and the call function gets the y_true and y_pred parameters

```python
from tensorflow.keras.losses import Loss

class MyHuberLoss(Loss):
  threshold = 1
  def __init__(self, threshold):
    super().__init__()
    self.threshold = threshold

  def call(self, y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= self.threshold
    small_error_loss = tf.square(error) / 2
    big_error_loss = self.threshold * (tf.abs(error) - (0.5 * self.threshold))
    return tf.where(is_small_error, small_error_loss, big_error_loss)
```

So we will declare threshold as a class variable which allows us to give it an initial value. S mention earlier, the threshold parameters when passed into an object will be received in an init function.

So within an init function, we can set the threshold class variable to be the parameterized one.

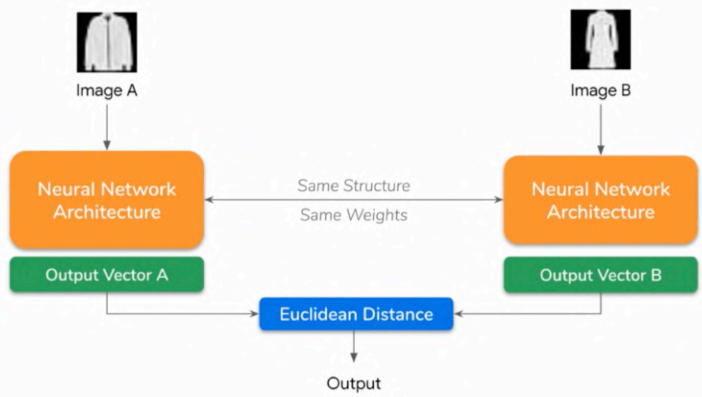
```python
from tensorflow.keras.losses import Loss

class MyHuberLoss(Loss):
  threshold = 1
  def __init__(self, threshold):
    super().__init__()
    self.threshold = threshold

  def call(self, y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= self.threshold
    small_error_loss = tf.square(error) / 2
    big_error_loss = self.threshold * (tf.abs(error) - (0.5 * self.threshold))
    return tf.where(is_small_error, small_error_loss, big_error_loss)
```

And the threshold class variable will be referred to within the class function as self.threshold.

Then when we want to specify the loss function in our model.compile, we can do so by specifying the MyHuberLossClass and we can pass in our threshold value as a parameter

```python
model.compile(optimizer='sgd', loss=MyHuberLoss(threshold=1))
```

Siamese Network for Image Similarity



Contrastive Loss

- If images are similar, produce feature vectors that are very similar
- If images are different, produce feature vectors that are dissimilar.
- Based on the paper
  "Dimensionality Reduction by Learning an Invariant Mapping"
  by R. Hadsell ; S. Chopra ; Y. LeCun
  http://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.pdf

To calculate the loss in this, we needed a new type of loss function that wasn't in our tool care. We called it contrastive loss as we wanted to contrast the images against each other.

The idea is that if two images are similar, we want to produce a feature vector for each image where the vectors are very similar.

If the images are different, we want their respective feature vectors to also be different.

The paper dimensionality reduction by learning an invariant mapping is the basis for loss like this.

Contrastive Loss - Formula

$$\boxed{Y} * D^2 + (1 - \boxed{Y}) * max(margin - D, 0)^2$$

The formula for contrastive loss is here. It's Y times D squared plus one minus Y times the max of a margin value minus D or zero squared. Now there's a lot to breakdown here, so let's look at each of these elements in turn.

Here, Y is the tensor of details about image similarities. They are one if the images are similar and they are zero if they're not.

Contrastive Loss - Formula

$$Y * \boxed{D}^2 + (1 - Y) * \max(\text{margin} - \boxed{D}, 0)^2$$

D is the tensor of Euclidean distances between the pairs of images.

Contrastive Loss - Formula

$$Y * D^2 + (1 - Y) * \max(\boxed{\text{margin}} - D, 0)^2$$

Margin is a constant that we can use to enforce a minimum distance between them in order to consider them similar or different.

Contrastive Loss - Formula

$$Y * D^2 + (1 - Y) * \max(\text{margin} - D, 0)^2$$

$$\Downarrow$$

$$1 * D^2 + (1 - 1) * \max(\text{margin} - D, 0)^2$$

$$\Downarrow$$

$$D^2$$

Margin is a constant that we can use to enforce a minimum distance between them in order to consider them similar or different.

Contrastive Loss - Formula

$$Y * D^2 + (1 - Y) * max(margin - D, 0)^2$$

$$0 * D^2 + (1 - 0) * max(margin - D, 0)^2$$

$$max(margin - D, 0)^2$$

When Y is zero, and we sub this in for Y, then our value instead of D squared will be the max between the margin minus D or zero, which is then squared, and this should be a much smaller value than D squared. You can think of the Y and one minus Y in this loss function as weights that will either allow the D squared of the max part of the formula to dominate the overall loss. When Y is close to one, it gives more weight to the D squared term and less weight on the max term. The D squared term will dominate the calculation of the loss. Conversely, when Y is closer to zero, this gives much more weight to the max term and less weight to the D squared term, so the max term dominates the calculation of the loss.

Contrastive Loss - Formula

$$Y * D^2 + (1 - Y) * max(margin - D, 0)^2$$

$$Y_{true} * Y_{pred}^{\,2} + (1 - Y_{true}) * max(margin - Y_{pred}, 0)^2$$

When we take into account the parameters that TensorFlow expects for a loss function, let's rewrite it like this.

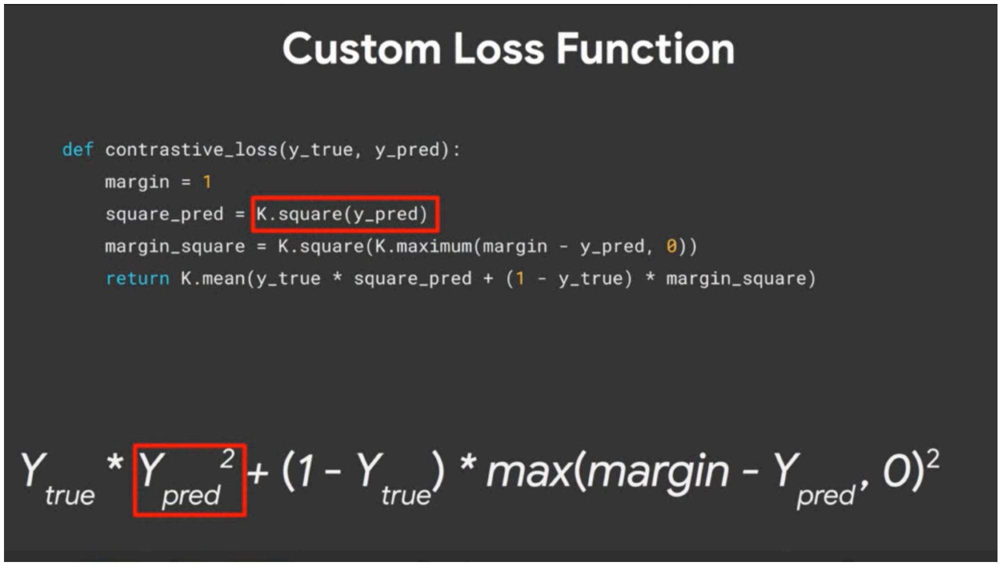
The Y in the original formula becomes the Y true value.

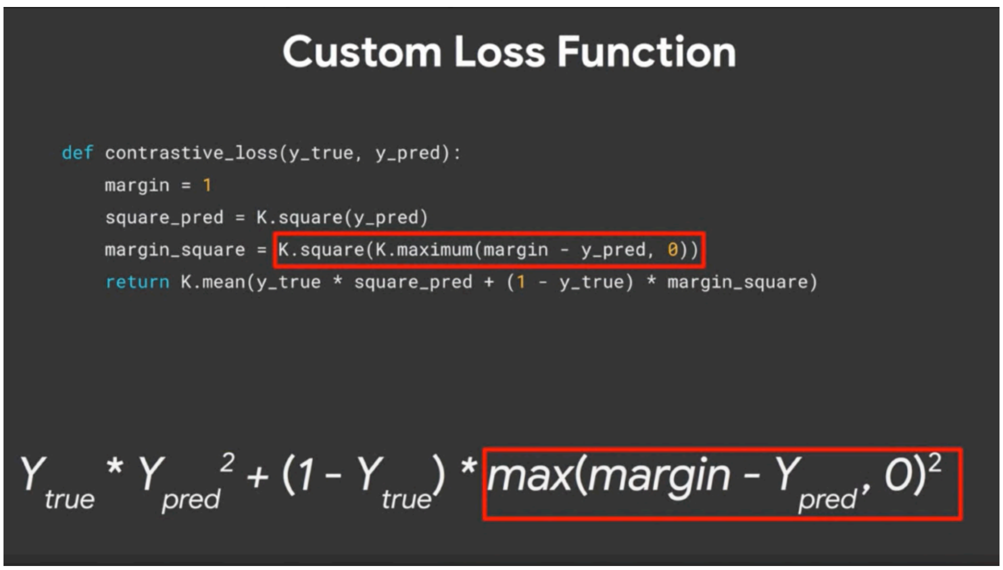
Contrastive Loss - Formula

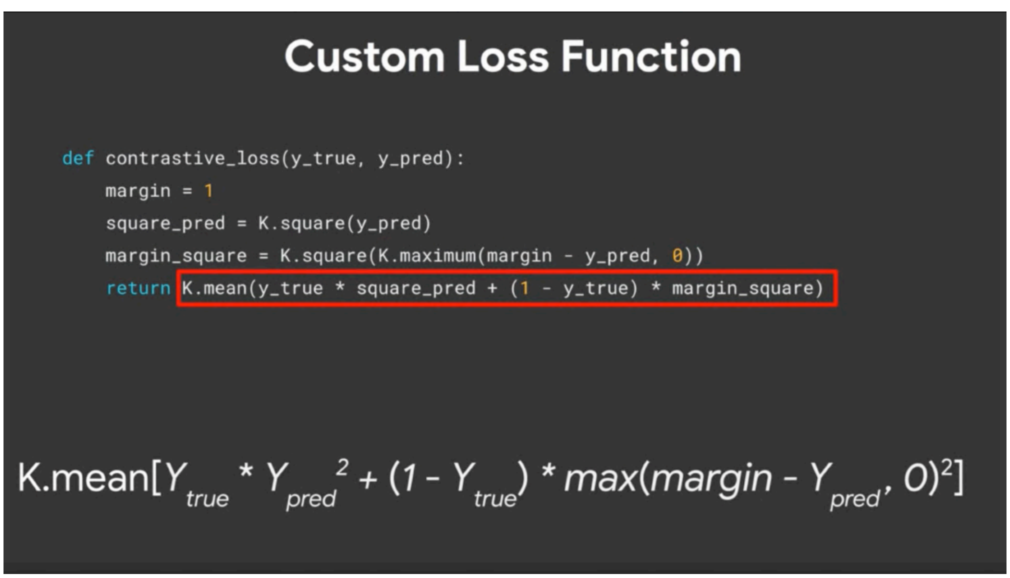
$$Y * D^2 + (1 - Y) * max(margin - D, 0)^2$$

$$Y_{true} * Y_{pred}^{\,2} + (1 - Y_{true}) * max(margin - Y_{pred}, 0)^2$$

The D in the original formula becomes the Y pred value, and now we have the two values you need.


Custom Loss Function
def contrastive_loss(y_true, y_pred):
margin = 1
square_pred = K.square(y_pred)
margin_square = K.square(K.maximum(margin - y_pred, 0))
return K.mean(y_true * square_pred + (1 - y_true) * margin_square)
$Y_{true} * Y_{pred}^{2} + (1 - Y_{true}) * max(margin - Y_{pred}, 0)^2$


Custom Loss Function
def contrastive_loss(y_true, y_pred):
margin = 1
square_pred = K.square(y_pred)
margin_square = K.square(K.maximum(margin - y_pred, 0))
return K.mean(y_true * square_pred + (1 - y_true) * margin_square)
$Y_{true} * Y_{pred}^{2} + (1 - Y_{true}) * max(margin - Y_{pred}, 0)^2$


Custom Loss Function
def contrastive_loss(y_true, y_pred):
margin = 1
square_pred = K.square(y_pred)
margin_square = K.square(K.maximum(margin - y_pred, 0))
return K.mean(y_true * square_pred + (1 - y_true) * margin_square)
$K.mean[Y_{true} * Y_{pred}^{2} + (1 - Y_{true}) * max(margin - Y_{pred}, 0)^2]$

Usage of Custom Loss

```
model.compile(loss=constrastive_loss, optimizer=RMSprop())
```

Custom Loss Function with Arguments

```
def contrastive_loss_with_margin(margin):
  #Original Loss Function
  def contrastive_loss(y_true, y_pred):
      square_pred = K.square(y_pred)
      margin_square = K.square(K.maximum(margin - y_pred, 0))
      return K.mean(y_true * square_pred + (1 - y_true) * margin_square)
  return contrastive_loss
```

Usage of Wrapper Loss Function

```
model.compile(loss=contrastive_loss_with_margin(margin=1), optimizer=rms)
```

Contrastive Loss - Object Oriented

```
class ContrastiveLoss(Loss):
  margin = 0
  def __init__(self,margin):
    super().__init__()
    self.margin = margin

  def call(self, y_true, y_pred):
    square_pred = K.square(y_pred)
    margin_square = K.square(K.maximum(self.margin - y_pred, 0))
    return K.mean(y_true * square_pred + (1 - y_true) * margin_square)
```

Contrastive Loss - Object Oriented

```
class ContrastiveLoss(Loss):
  margin = 0
  def __init__(self,margin):
    super().__init__()
    self.margin = margin

  def call(self, y_true, y_pred):
    square_pred = K.square(y_pred)
    margin_square = K.square(K.maximum(self.margin - y_pred, 0))
    return K.mean(y_true * square_pred + (1 - y_true) * margin_square)
```

Usage of Object Oriented Loss

```
model.compile(loss=ContrastiveLoss(margin=1), optimizer=rms)
```

Huber Loss:

- https://en.wikipedia.org/wiki/Huber_loss

Dimensionality Reduction by Learning an Invariant Mapping:

- http://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.pdf