Introduction to Callbacks

Callbacks are a useful piece of functionality in Tensorflow that lets you have control over the training process, there's two main flavor of callback. There's the built in callbacks that are pre built in functions that allow you to do things like saving checkpoints early, stopping on this custom callbacks where you can override the callback class to do whatever you want.

In this section, I'm going to look at the built in callbacks, and then later you could learn how to do the custom ones.

Callbacks

- Provides some functionality at various stages of training
- Subclasses `tf.keras.callbacks.Callback`
- Useful in understanding a model's state during training
  - internal states
  - statistics e.g., losses and metrics

So in summary, callbacks are designed to give you some type of functionality, while you're training every epoch, you can effectively have code that executes to perform a task.

What that task is is up to you, there's a tf.keras.callbacks.Callback class that you'll subclass, so the pattern you've been looking at in this course for sub classing existing objects will also work for this. They're particularly useful in helping you understand the model state during training, saving you valuable time is your optimizing your model.

Training specific methods

```
class Callback(object):
  def __init__(self):
    self.validation_data = None
    self.model = None

  def on_epoch_begin(self, epoch, logs=None):
    """Called at the beginning of an epoch during training."""

  def on_epoch_end(self, epoch, logs=None):
    """Called at the end of an epoch during training."""
```

Here's the anatomy of a callback class, as with any class in python, you define it to extend an existing class and you have local variables initialized in the init function.

For callbacks, you have the epoch begin function that you can override, which, as its name suggests, gets called at the beginning of every epoch.

Training specific methods

```
class Callback(object):
  def __init__(self):
    self.validation_data = None
    self.model = None

  def on_epoch_begin(self, epoch, logs=None):
    """Called at the beginning of an epoch during training."""

  def on_epoch_end(self, epoch, logs=None):
    """Called at the end of an epoch during training."""
```

Common methods for training/testing/predicting

```
class Callback(object):
  ...
  def on_(train|test|predict)_begin(self, logs=None):
    """Called at the begin of fit/evaluate/predict."""

  def on_(train|test|predict)_end(self, logs=None):
    """Called at the end of fit/evaluate/predict."""

  def on_(train|test|predict)_batch_begin(self, batch, logs=None):
    """Called right before processing a batch during training/testing/predicting."""

  def on_(train|test|predict)_batch_end(self, batch, logs=None):
    """Called at the end of training/testing/predicting a batch."""
```

So, for example, when you run a prediction, you can have a callback that happens at the beginning or the end by calling the on_predict begin method or the on_predict end method, respectively, you could do similar for on training and on testing.

Common methods for training/testing/predicting

```
class Callback(object):
  ...
  def on_(train|test|predict)_begin(self, logs=None):
    """Called at the begin of fit/evaluate/predict."""

  def on_(train|test|predict)_end(self, logs=None):
    """Called at the end of fit/evaluate/predict."""

  def on_(train|test|predict)_batch_begin(self, batch, logs=None):
    """Called right before processing a batch during training/testing/predicting."""

  def on_(train|test|predict)_batch_end(self, batch, logs=None):
    """Called at the end of training/testing/predicting a batch."""
```

Also, when running batches, you can have on_predict batch begin or on_predict batch end so that you can execute code batch by batch while predicting, and of course, you could do similar for training and testing.

Where can you use them?

Model methods that take callbacks

- `fit(..., callbacks=[...])`
- `fit_generator(..., callbacks=[...])`
- `evaluate(..., callbacks=[...])`
- `evaluate_generator(..., callbacks=[...])`
- `predict(..., callbacks=[...])`
- `predict_generator(..., callbacks=[...])`

This, of course, leads to the question, where would you use callbacks?

Well, the model methods that involve training, evaluation of prediction used them, you simply specify them, using the callbacks equal parameter.

TensorBoard Callback

- Visualize machine learning experiments
- Track metrics (e.g., loss, accuracy)
- View the model graph

```
TensorBoard(log_dir='./logs', update_freq='epoch', **kwargs)
```

https://www.tensorflow.org/tensorboard

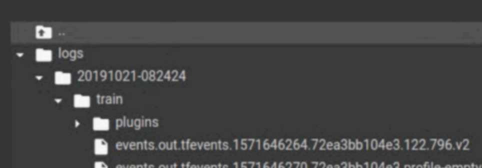
So now let's take a look at some of the built in callbacks, we'll start with TensorBoard, which, if you aren't familiar with it, provides a suite of visualization tools for tensorflow.

Unless you visualize your experiments and track metrics like loss and accuracy, Acela's viewing the model graph you can learn more about attentive load torque slash centerboard

TensorBoard

Define the callback and start training

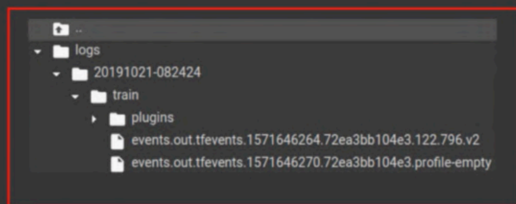
```
log_dir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
tensorboard = tf.keras.callbacks.TensorBoard(log_dir=log_dir)
model.fit(train_batches, epochs=10, callbacks=[tensorboard])
```



To use the TensorBoard callback is super simple. You simply define it and then start training, it's defined by creating an instance of the TensorBoard call back and specifying the desired log directory.

TensorBoard — Define the callback and start training

```
log_dir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
tensorboard = tf.keras.callbacks.TensorBoard(log_dir=log_dir)
model.fit(train_batches, epochs=10, callbacks=[tensorboard])
```



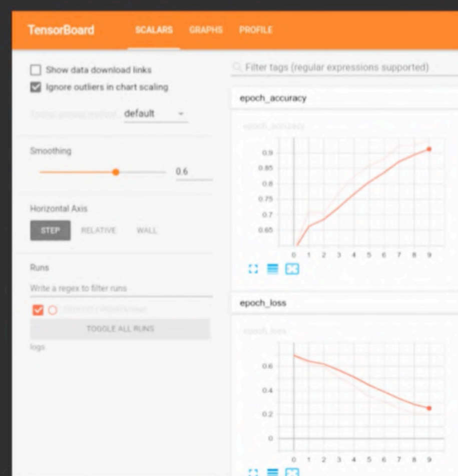
Tensorflow then saves the details to that directory

TensorBoard — TensorBoard in Colab

Then, when Tensorboard is pointed at the logs directory, it does it's thing such as plotting accuracy and loss. It can even be used by co lab by loading it as an extension, as you can see here.

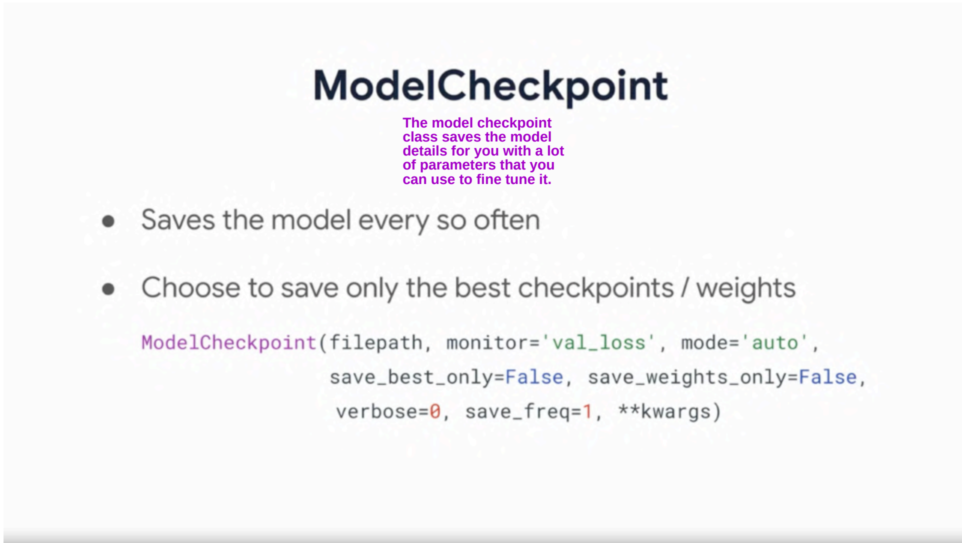
```
# Load the extension
%load_ext tensorboard

# Run TensorBoard
%tensorboard --logdir logs
```

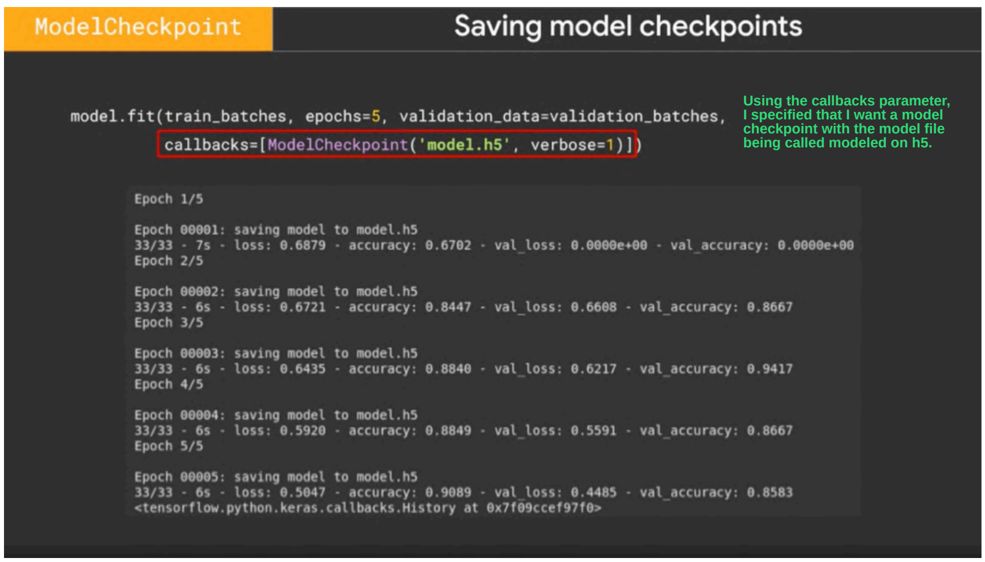


Model Checkpoints

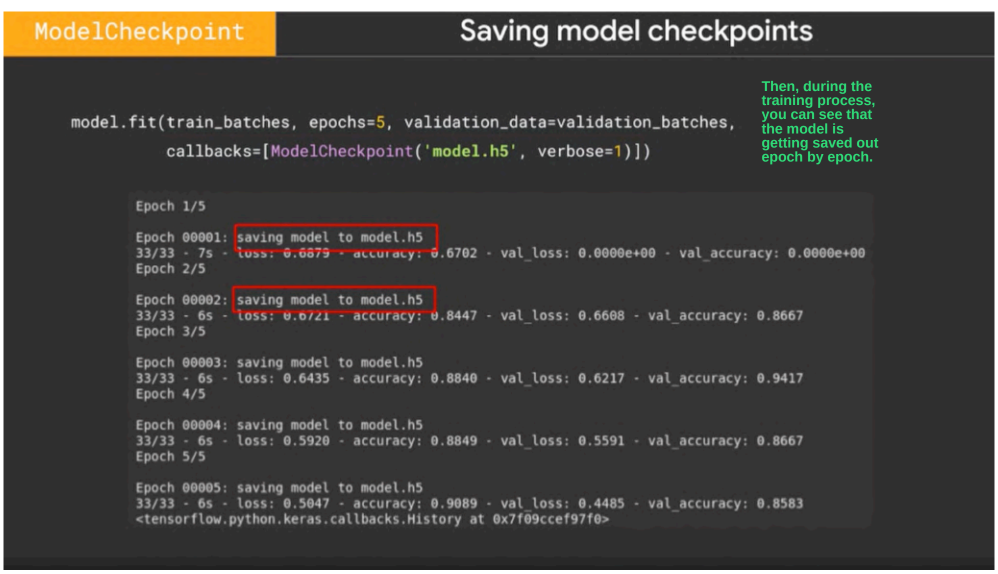
Next, take a look at the model checkpoints where the models details can be saved out, epoch by epoch for later inspection, or we can monitor progress through them.



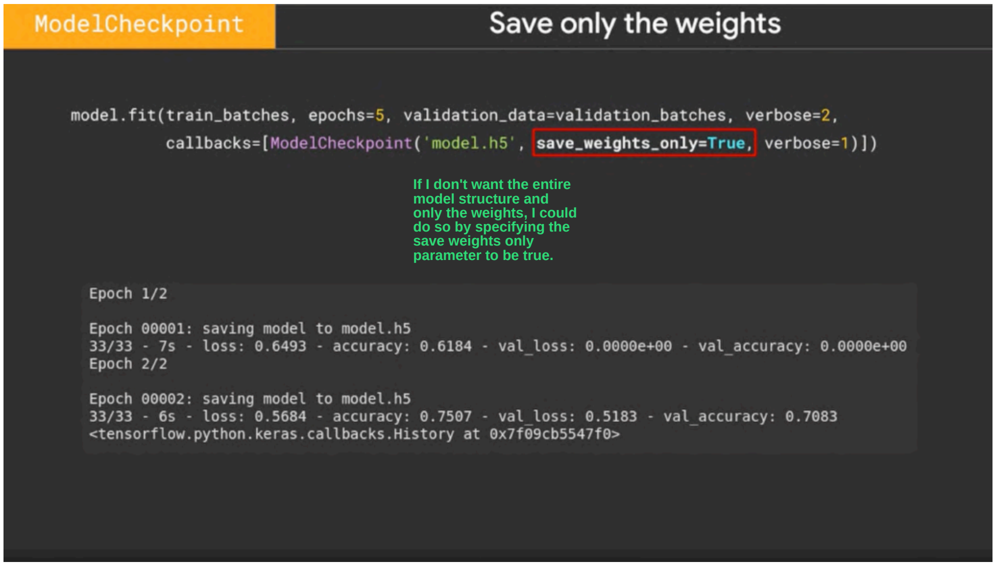
The model checkpoint class saves the model details for you with a lot of parameters that you can use to fine tune it.



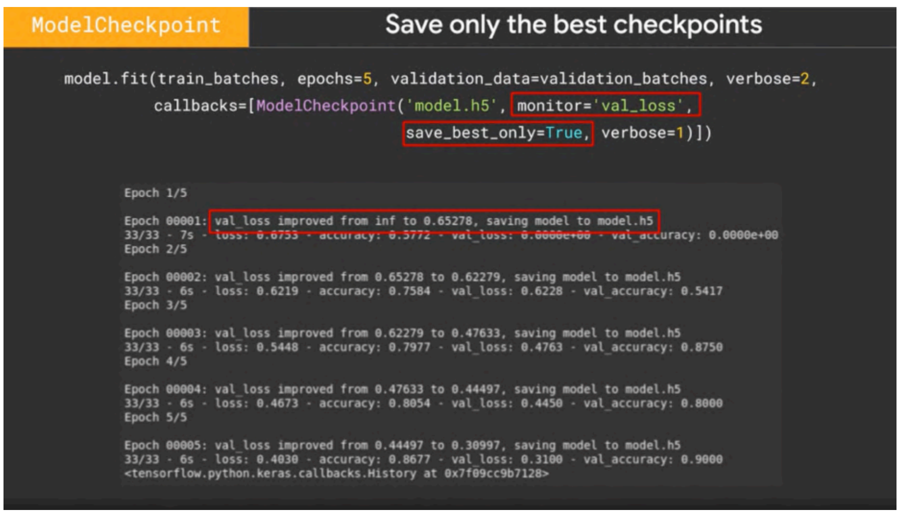
Using the callbacks parameter, I specified that I want a model checkpoint with the model file being called modeled on h5.



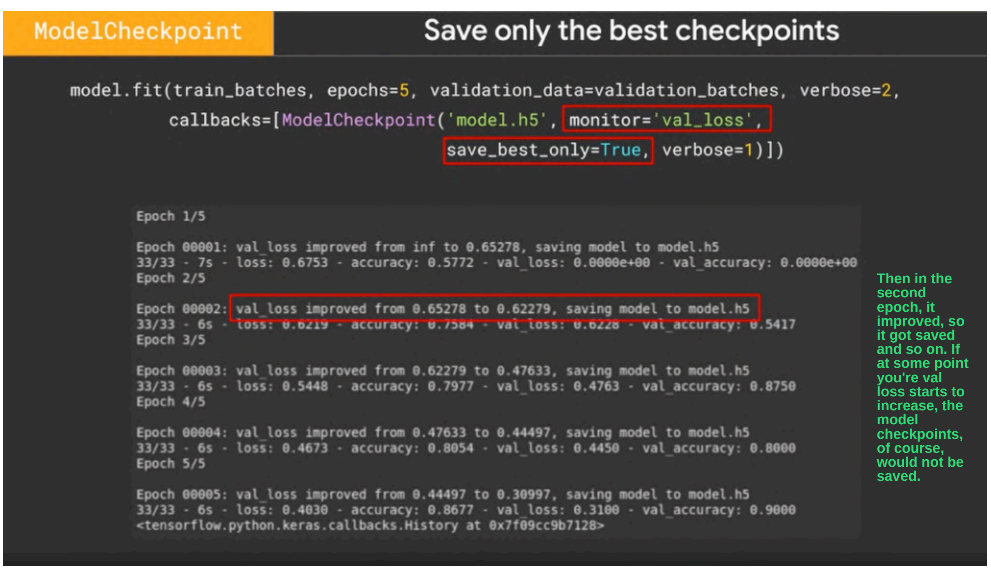
Then, during the training process, you can see that the model is getting saved out epoch by epoch.



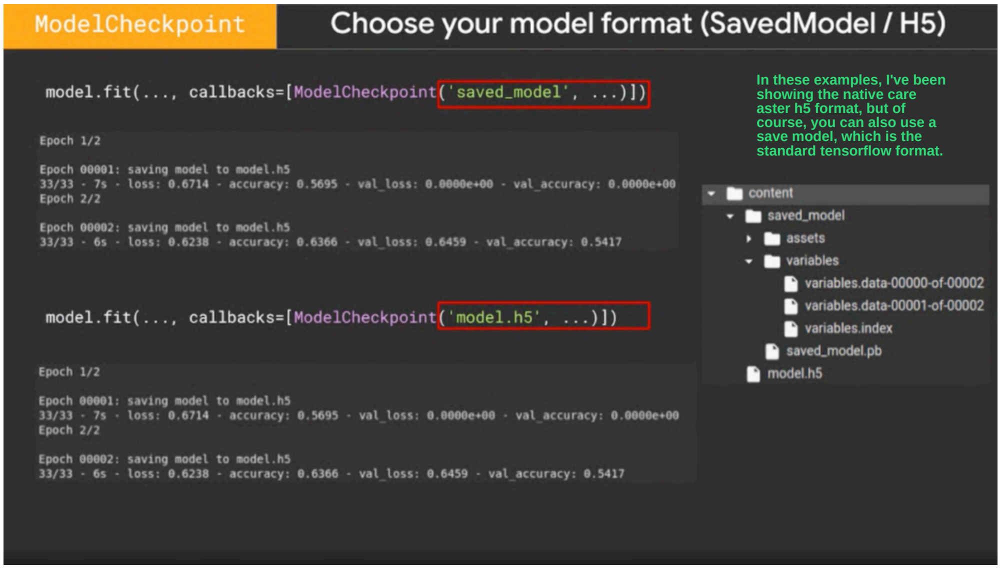
If I don't want the entire model structure and only the weights, I could do so by specifying the save weights only parameter to be true.



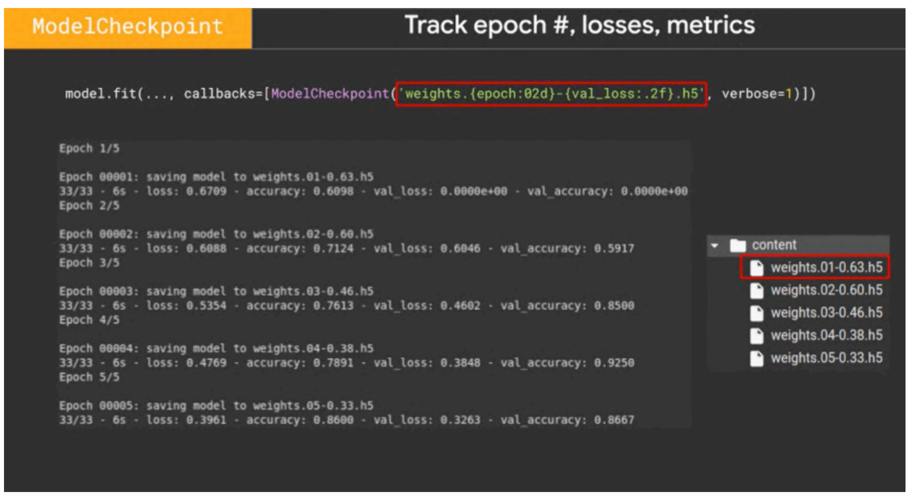
Or if I only want to save when I reach optimal values, I could do so by specifying save best, only to be true, then whatever value I specify in the monitor parameter will be saved whenever it's optimized. As you can see here in the first epoch, the value started as infinite, and it ended at 0.65278 so get safe



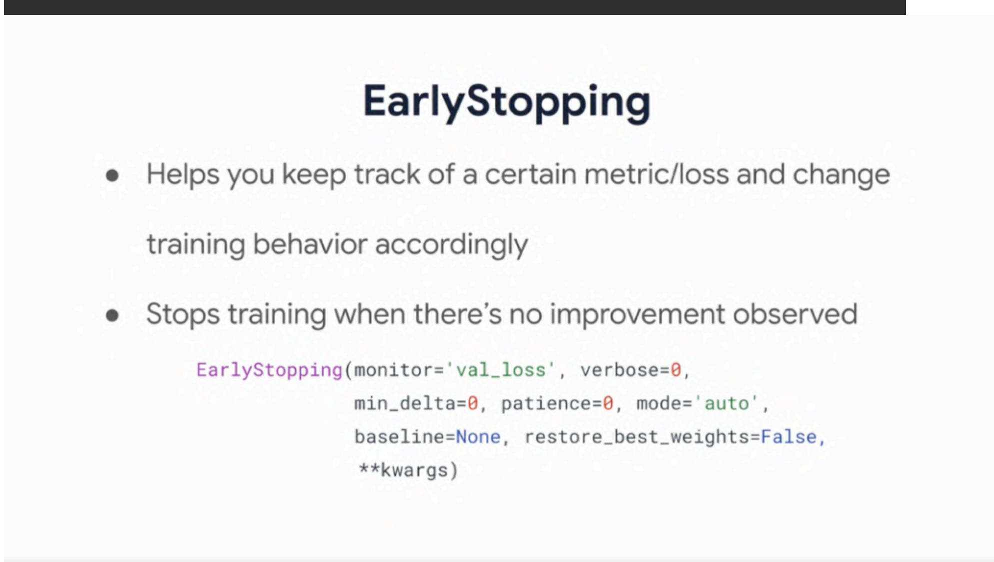
Then in the second epoch, it improved, so it got saved and so on. If at some point you're val loss starts to increase, the model checkpoints, of course, would not be saved.



In these examples, I've been showing the native care aster h5 format, but of course, you can also use a save model, which is the standard tensorflow format.



As the name of the file is just specified using text, you can actually form at the values within the name, so you could have separate weights saved out pair epoch in separate h5 files. Simply by using the epoch value or other metrics such as the validation lost value you conform at the file name so you can see here the last two digits of the epoch are used, so the file is wait 01 wait 02 and so on.



EarlyStopping — Prevent your model from overfitting

Set patients to three, the idea here is that
once we hit the best value will log that
and we'll wait for this number of epochs
ie three to see if the values improve.

```
model.fit(train_batches, epochs=50, validation_data=validation_batches,
          callbacks=[EarlyStopping(patience=3, monitor='val_loss')])
```

```
Epoch 11/50
33/33 [==============================] - 6s 184ms/step - loss: 0.2423 - accuracy: 0.9319 - val_loss: 0.1474 - val_accuracy: 0.9500
Epoch 12/50
33/33 [==============================] - 6s 184ms/step - loss: 0.1521 - accuracy: 0.9607 - val_loss: 0.1990 - val_accuracy: 0.9000
Epoch 13/50
33/33 [==============================] - 6s 182ms/step - loss: 0.1571 - accuracy: 0.9511 - val_loss: 0.1176 - val_accuracy: 0.9500
Epoch 14/50
33/33 [==============================] - 6s 186ms/step - loss: 0.1409 - accuracy: 0.9569 - val_loss: 0.1071 - val_accuracy: 0.9583
Epoch 15/50
33/33 [==============================] - 6s 185ms/step - loss: 0.1450 - accuracy: 0.9626 - val_loss: 0.0953 - val_accuracy: 0.9583
Epoch 16/50
33/33 [==============================] - 6s 184ms/step - loss: 0.2023 - accuracy: 0.9396 - val_loss: 0.1413 - val_accuracy: 0.9583
Epoch 17/50
33/33 [==============================] - 6s 184ms/step - loss: 0.1443 - accuracy: 0.9655 - val_loss: 0.1771 - val_accuracy: 0.9167
Epoch 18/50
33/33 [==============================] - 6s 183ms/step - loss: 0.1442 - accuracy: 0.9521 - val_loss: 0.1201 - val_accuracy: 0.9333
Epoch 00018: early stopping
```

EarlyStopping — Restoring best weights

If you don't want to lose the weight values from the
best epoch you can set, restore best waits to be true.
So in our case, even though we stopped at 18 will
have the weight restored to where they were at 15.

```
model.fit(...,
          callbacks=[EarlyStopping(patience=3, restore_best_weights=True,
                                   monitor='val_loss', verbose=1)])
```

```
Epoch 11/50
33/33 - 6s - loss: 0.1380 - accuracy: 0.9616 - val_loss: 0.0968 - val_accuracy: 0.9750
Epoch 12/50
33/33 - 6s - loss: 0.1202 - accuracy: 0.9655 - val_loss: 0.0741 - val_accuracy: 0.9917
Epoch 13/50
33/33 - 6s - loss: 0.1716 - accuracy: 0.9434 - val_loss: 0.1083 - val_accuracy: 0.9750
Epoch 14/50
33/33 - 6s - loss: 0.1331 - accuracy: 0.9626 - val_loss: 0.0861 - val_accuracy: 0.9667
Epoch 15/50
Restoring model weights from the end of the best epoch.
33/33 - 6s - loss: 0.1393 - accuracy: 0.9578 - val_loss: 0.0771 - val_accuracy: 0.9750
Epoch 00015: early stopping
```

EarlyStopping — More customization

```
model.fit(...,
          callbacks=[EarlyStopping(
                                   patience=3,
                                   min_delta=0.05,
                                   baseline=0.8,
                                   mode='min',
                                   monitor='val_loss',
                                   verbose=1
                     )])
```

There's other parameters
you can play with, but the
mode one is crucial to
ensure that you're following
your monitor values
correctly for loss that you
want to minimize. So you
would then set the mode to
min, for others, they might
require you to maximize the
value so you could change
the mode with this property.

CSVLogger

Logging training results

```
model.fit(..., callbacks=[CSVLogger('training.csv')])
```

epoch	accuracy	loss	val_accuracy	val_loss
0	0.574305	0.682536	0.775000	0.655427
1	0.760307	0.633610	0.675000	0.595201
2	0.758389	0.573186	0.850000	0.503174
3	0.835091	0.472031	0.808333	0.416691
4	0.854267	0.419491	0.916667	0.309128

Another super useful callback is the CSVlogger which, as its name suggests, will log your training results out to a CSV file. So, for example, when using it like this, you'll have a file containing the epoch number, accuracy, loss, validation, accuracy and validation lost stored for you.

Build a simple model

```
model = tf.keras.Sequential()

model.add(tf.keras.layers.Dense(units=1,
                                activation='linear',
                                input_dim=(784,)))

model.compile(optimizer=tf.keras.optimizers.RMSprop(lr=0.1),
              loss='mean_squared_error', metrics=['mae'])
```

How a custom callback looks

```
import datetime

class MyCustomCallback(tf.keras.callbacks.Callback):
  def on_train_batch_begin(self, batch, logs=None):
    print('Training: batch {} begins at {}'
          .format(batch, datetime.datetime.now().time()))

  def on_train_batch_end(self, batch, logs=None):
    print('Training: batch {} ends at {}'
          .format(batch, datetime.datetime.now().time()))
```

Now let's define a custom callback, which we'll name my custom callback. We'll do this by subclassing the tf.keras.callbacks.callbackclass.

Our custom callback will print the timestamp each time when a training batch starts or ends. To achieve this we'll override the untrained batch begin and the untrained batch end methods to include our business logic, which just displays the timestamps.

To use the custom callback that we've just defined we'll instantiate an instance of this class. Here we save the instance of the custom callback in a variable named my_custom _callback and as usual, we'll train our model using model.fit.

```
my_custom_callback = MyCustomCallback()

model.fit(x_train, y_train, batch_size=64, epochs=1, verbose=0,
          callbacks=[my_custom_callback])
```

```
class DetectOverfittingCallback(tf.keras.callbacks.Callback):
  def __init__(self, threshold):
    super(DetectOverfittingCallback, self).__init__()
    self.threshold = threshold

  def on_epoch_end(self, epoch, logs=None):
    ratio = logs["val_loss"] / logs["loss"]
    print("Epoch: {}, Val/Train loss ratio: {:.2f}".format(epoch, ratio))

    if ratio>threshold:
      print("Stopping training...")
      self.model.stop_training = True

model.fit(..., callbacks=[DetectOverfittingCallback(threshold=1.3)])
```

To use the custom callback that we've just defined we'll instantiate an instance of this class. Here we save the instance of the custom callback in a variable named my_custom_callback and as usual, we'll train our model using model.fit.

```
class DetectOverfittingCallback(tf.keras.callbacks.Callback):
  def __init__(self, threshold):
    super(DetectOverfittingCallback, self).__init__()
    self.threshold = threshold

  def on_epoch_end(self, epoch, logs=None):
    ratio = logs["val_loss"] / logs["loss"]
    print("Epoch: {}, Val/Train loss ratio: {:.2f}".format(epoch, ratio))

    if ratio>threshold:
      print("Stopping training...")
      self.model.stop_training = True

model.fit(..., callbacks=[DetectOverfittingCallback(threshold=1.3)])
```

Let's explore a call back where we measure the ratio between our validation loss and our training loss.

If the ratio gets too high, we could have an over-fitting scenario because the validation loss may no longer be decreasing while the training loss continues to decrease, making the ratio of validation loss divided by training loss higher.

We should in this case, stop training to avoid overfitting. We'll do this by defining a class called Detect Overfitting Callback and this subclass is the Keras callback base class.

We'll need a class level variable to hold the threshold so that we can override the init function to take the threshold as a parameter and then store it in self.threshold.

```python
class DetectOverfittingCallback(tf.keras.callbacks.Callback):
  def __init__(self, threshold):
    super(DetectOverfittingCallback, self).__init__()
    self.threshold = threshold

  def on_epoch_end(self, epoch, logs=None):
    ratio = logs["val_loss"] / logs["loss"]
    print("Epoch: {}, Val/Train loss ratio: {:.2f}".format(epoch, ratio))

    if ratio>threshold:
      print("Stopping training...")
      self.model.stop_training = True

model.fit(..., callbacks=[DetectOverfittingCallback(threshold=1.3)])
```

We can then implement the on epoch end class method overriding the base class. In this method we'll compute the ratio at the end of every epoch.

```python
class DetectOverfittingCallback(tf.keras.callbacks.Callback):
  def __init__(self, threshold):
    super(DetectOverfittingCallback, self).__init__()
    self.threshold = threshold

  def on_epoch_end(self, epoch, logs=None):
    ratio = logs["val_loss"] / logs["loss"]
    print("Epoch: {}, Val/Train loss ratio: {:.2f}".format(epoch, ratio))

    if ratio>threshold:
      print("Stopping training...")
      self.model.stop_training = True

model.fit(..., callbacks=[DetectOverfittingCallback(threshold=1.3)])
```

If that ratio was higher than our threshold value, we can stop training.

```python
class DetectOverfittingCallback(tf.keras.callbacks.Callback):
  def __init__(self, threshold):
    super(DetectOverfittingCallback, self).__init__()
    self.threshold = threshold

  def on_epoch_end(self, epoch, logs=None):
    ratio = logs["val_loss"] / logs["loss"]
    print("Epoch: {}, Val/Train loss ratio: {:.2f}".format(epoch, ratio))

    if ratio>threshold:
      print("Stopping training...")
      self.model.stop_training = True

model.fit(..., callbacks=[DetectOverfittingCallback(threshold=1.3)])
```

To use this, we'll just then specify detect overfitting callback when performing the model.fit and pass in our desired threshold.

Notice that we're creating an instance of the class by invoking the detect overfitting callback class constructor and placing it directly into the list that gets passed to the callbacks parameter.

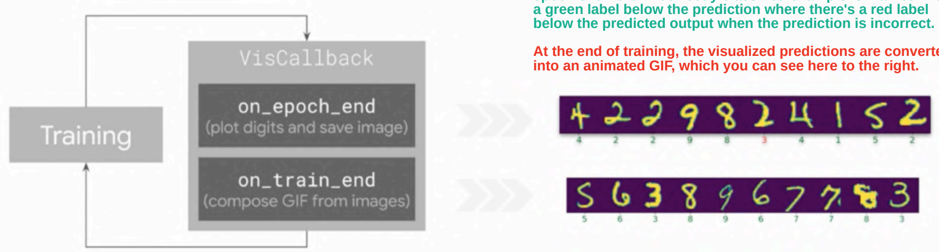
You could have also stored an instance of the callback class in a variable and passed in that variable, as you saw in the earlier example with my custom callback.

Let's look at another example of customizing a callback. In this example, we'll train an endless classifier and we'll define a custom callback function called VisCallback.

At the end of every epoch, the custom callback generates a visualization of the classified outputs and it will save that image out to disk.

You can see an example of this saved image on the right of an epoch end here. A correctly classified example is marked with a green label below the prediction where there's a red label below the predicted output when the prediction is incorrect.

At the end of training, the visualized predictions are converted into an animated GIF, which you can see here to the right.




```python
class VisCallback(tf.keras.callbacks.Callback):
  def __init__(self, inputs, ground_truth, display_freq=10,
                                           n_samples=10):
    self.inputs = inputs
    self.ground_truth = ground_truth
    self.images = []
    self.display_freq = display_freq
    self.n_samples = n_samples
```

First we'll define a custom callback called VisCallback subclassing tf.keras.callbacks.callback and we'll have a set of class variables.

self.input holds the inputs to run in the model.

self.ground_truth holds the ground truth or the true labels for those input images. That allows us to compare each prediction with the actual label.

```python
class VisCallback(tf.keras.callbacks.Callback):
  def __init__(self, inputs, ground_truth, display_freq=10,
                                           n_samples=10):
    self.inputs = inputs
    self.ground_truth = ground_truth
    self.images = []
    self.display_freq = display_freq
    self.n_samples = n_samples
```

Self.images will hold the visualize comparison of the prediction against the ground truth.

```python
class VisCallback(tf.keras.callbacks.Callback):
  def __init__(self, inputs, ground_truth, display_freq=10,
                                          n_samples=10):
    self.inputs = inputs
    self.ground_truth = ground_truth
    self.images = []
    self.display_freq = display_freq
    self.n_samples = n_samples
```

Self.display frequency allows you to choose how frequently you want to display these plots. For instance, you might decide to display a visual at every 10 epochs instead of every epoch.

```python
class VisCallback(tf.keras.callbacks.Callback):
  def __init__(self, inputs, ground_truth, display_freq=10,
                                          n_samples=10):
    self.inputs = inputs
    self.ground_truth = ground_truth
    self.images = []
    self.display_freq = display_freq
    self.n_samples = n_samples
```

Self.n_samples then determines how many samples to be plotted each time a visualization is generated.

At the end of every epoch we'll randomly sample data from the list of input images and then classify them.

Accordingly, we'll mark the label predictions green or red based on whether they're classified correctly or not.

We'll start by using np.random.choice passing it the number of inputs and the size of samples that we want. This will give us a list of random indexes back and we can use these to pick the images from our training data completely at random.

```python
class VisCallback(tf.keras.callbacks.Callback):
  ...
  def on_epoch_end(self, epoch, logs=None):
    # Randomly sample data
    indexes = np.random.choice(len(self.inputs), size=self.n_samples)
    X_test, y_test = self.inputs[indexes], self.ground_truth[indexes]
    predictions = np.argmax(self.model.predict(X_test), axis=1)
```

We can then get our x values from the test, from the inputs and using those randomly generated indexes, the corresponding y values from the ground truth.

```
class VisCallback(tf.keras.callbacks.Callback):
  ...
  def on_epoch_end(self, epoch, logs=None):
    # Randomly sample data
    indexes = np.random.choice(len(self.inputs), size=self.n_samples)
    X_test, y_test = self.inputs[indexes], self.ground_truth[indexes]
    predictions = np.argmax(self.model.predict(X_test), axis=1)
```

To get our predictions, you can use model.predict and pass it the X test values. You'll use argmax to get the most likely value.

Recall that argmax will take a list, find the maximum value on that list, and then return the index position of that maximum value.

```
class VisCallback(tf.keras.callbacks.Callback):
  ...
  def on_epoch_end(self, epoch, logs=None):
    # Randomly sample data
    indexes = np.random.choice(len(self.inputs), size=self.n_samples)
    X_test, y_test = self.inputs[indexes], self.ground_truth[indexes]
    predictions = np.argmax(self.model.predict(X_test), axis=1)
```

Now that we have our predictions, we can pass our data to a display function to draw them in our plotter.

```
class VisCallback(tf.keras.callbacks.Callback):
  ...
  def on_epoch_end(self, epoch, logs=None):
    # Randomly sample data
    indexes = np.random.choice(len(self.inputs), size=self.n_samples)
    X_test, y_test = self.inputs[indexes], self.ground_truth[indexes]
    predictions = np.argmax(self.model.predict(X_test), axis=1)

    # Plot the digits
    display_digits(X_test, predictions, y_test, epoch, n=self.display_freq)
    ...
```

```python
class VisCallback(tf.keras.callbacks.Callback):
  ...
  def on_epoch_end(self, epoch, logs=None):
    ...
    # Save the figure
    buf = io.BytesIO()
    plt.savefig(buf, format='png')
    buf.seek(0)
    image = Image.open(buf)
    self.images.append(np.array(image))

    # Display the digits every now and then
    if epoch % self.display_freq == 0:
      plt.show()
```

Then when we save those plotted images in a list by reading the results of the plot into a buffer, which can then be appended to the list called self.images.

```python
class VisCallback(tf.keras.callbacks.Callback):
  ...
  def on_epoch_end(self, epoch, logs=None):
    ...
    # Save the figure
    buf = io.BytesIO()
    plt.savefig(buf, format='png')
    buf.seek(0)
    image = Image.open(buf)
    self.images.append(np.array(image))

    # Display the digits every now and then
    if epoch % self.display_freq == 0:
      plt.show()
```

If we want to show the plots occasionally, we can do so with code like this, where for example, if the frequency is 10, it will render every tenth epoch.

Here's how you can create an animated GIF that's composed of several saved images. The imageio library for Python contains a mimsave method that lets you specify an array of images and it will write out an animated GIF of them. As you've kept a list of images in self.images, this is now really easy to do.

```python
import imageio

class VisCallback(tf.keras.callbacks.Callback):
  ...
  def on_train_end(self, logs=None):
    imageio.mimsave('animation.gif', self.images, fps=1)

# Train the model
model.fit(..., callbacks=[VisCallback(x_test, y_test)])
```

To get this visualization, you specify this custom class in the callbacks when you call your model.fit.

You'll pass in the x test and y test so that the visualizations are created on the test set and not on the training set.

```python
import imageio

class VisCallback(tf.keras.callbacks.Callback):
  ...
  def on_train_end(self, logs=None):
    imageio.mimsave('animation.gif', self.images, fps=1)

# Train the model
model.fit(..., callbacks=[VisCallback(x_test, y_test)])
```

TensorBoard: TensorFlow's visualization toolkit

- https://www.tensorflow.org/tensorboard