TensorFlow 2 for Deep Learning Specialization

TensorFlow 2 for Deep Learning Specialization

- course1 Getting started with TensorFlow 2
  - week 1
    - 1.Introduction to the course
    - 2.Welcome to week 1
    - 3.Hello TensorFlow!
    - 4.Coding tutorial--Hello TensorFlow!
    - 5.What's new in TensorFlow2
    - 6.TensorFlow documentation
  - week 2
    - 1.Welcome to week 2 - The Sequential model API
    - 2.What is keras?
    - 3.Building a sequential model
    - 4.Convolutional and pooling layers
    - 5.The compile method
    - 6.The fit method
    - 7.The evaluate and predict methods
  - week3
    - 1.Validation, regularisation and callbacks
    - 2.Validation sets
    - 3.Model regularisation
    - 4.Introduction to callbacks
    - 5.Early stopping and patience
  - week4
    - 1.Welcome to week4-Saving and loading models
    - 2.Saving and loading model weights
    - 3.Model saving criteria
    - 4.Saving the entire model
    - 5.Loading pre-trained Keras models

course1 Getting started with TensorFlow 2

week 1

1.Introduction to the course

大家好，欢迎来到TensorFlow 2的深度学习专论。我叫凯文·韦伯斯特，是伦敦帝国理工学院数学系的高级讲师。我很高兴能在这里教这门课。和我一起的还有一个很棒的研究生助教团队，他们都是帝国理工的博士生。他们将在整个课程中帮助我，指导你通过这个深度学习库可以做的所有神奇的事情。TensorFlow是深度学习最流行的库之一。这是谷歌在2015年发布的开源项目。今天，它在各个层次的研究人员和专业人员中被广泛使用。有几件事我想强调并解释一下，在这节课的开始。首先，这不是一门关于深度学习的课程。我们假设你已经对深度学习模型有了一定的了解。这门课是关于学习TensorFlow 2的。我想强调的第二件事是，这门课是针对之前使用过TensorFlow 1的人，以及完全不熟悉TensorFlow的人。你不需要有任何经验就能上这门课。如果你对本地安装、远程服务器或gpu培训感到害怕，你不需要担心这些，就可以学习如何在TensorFlow中构建、培训和使用深度学习模型。在本课程中，你需要的一切都可以在Coursera平台上找到，你可以完全通过浏览器完成课程。这门课程的结构是结合了讲课视频和实用的编码教程。所以你会把你所学到的所有内容直接运用到实践中。您将为一系列应用开发深度学习模型，包括图像分类、文本、情感分析、回归任务和生成语言模型。在课程的每一周结束时，将会有一个编程作业，你将需要应用你从那周学到的概念。编程作业都在Jupyter笔记本上，你需要通过Coursera平台完成并提交。此外，在本专业的每门课程结束时，都有一个毕业项目。这些毕业项目是扩展项目，它将包括整个课程的材料，以进一步扩展您使用TensorFlow的知识。顶尖级项目是由同学评估的，所以你可以和同学们互相得到反馈。我希望这能让你们清楚地了解这门课的内容，以及你们将会学到什么。我真的希望你会发现这门课很有用，而且很有趣。期待在以后的视频中见到你们。

2.Welcome to week 1

大家好，欢迎回来。在这个视频中，我想为我们这周的课程做一个铺垫。这门课的第一周和接下来的几周会有不同的形式，这是因为在我们开始学习TensorFlow 2之前，我想先概述一下一些重要的概念发展，与TensorFlow 1的不同之处，并介绍一些我认为你们会觉得非常有用的资源和工具，比如谷歌Colab，当然还有TensorFlow文档。这周还有几项我标为可选的。这些内容包括如何在你的本地机器上安装TensorFlow，或者使用TensorFlow的不同方式，或者如何升级你可能用TensorFlow 1编写的代码以兼容TensorFlow 2。现在，你不需要通过这些可选的项目来学习这门课。正如我在介绍视频中提到的，你可以完全通过浏览器完成这门课程。这些可选的项目是作为额外的资源，给那些想要利用它们的人。如果你愿意，可以跳过这些项目。如果您想要准备自己的设置，您可以稍后再回到它们。第一周也没有编程作业，但从下周开始，每个周末都会有一个。话虽如此，我们还是开始吧。

3.Hello TensorFlow!

从下周开始，这门课的结构将由我的讲座视频组成，我将解释并展示TensorFlow中的不同特征和对象，以及由研究生助教或gta提供的编码教程视频，在Coursera平台上，我将要展示的材料将会和示例一起被编码到一个教程笔记本中。

在这些编码教程视频中，你可以在课程的书架上单独找到笔记本。你可以在一个单独的窗口中打开这个笔记本，这样你就可以在跟随教程视频中GTA的过程中自己编写示例代码。这是一个很好的练习，可以让你自己在TensorFlow中编写代码。

紧接着这个视频的编码教程视频和笔记本会给你一个如何工作的框架。这将是这周课程内容的主要形式。

让我们继续吧，打开下面的编码教程视频和笔记本，Jerome会带你们看TensorFlow 2中的第一个hello world例子。

4.Coding tutorial--Hello TensorFlow!

大家好，欢迎来到深度学习的TensorFlow 2。我的名字叫杰罗姆。我是伦敦帝国理工学院数学系的博士生。正如Kevin已经告诉过你的，在这节课中，我们将在TensorFlow中通过一个hello world的例子。我们会用Coursera的笔记本导入TensorFlow，我们会检查它的版本，然后我们会运行一个训练神经网络的小脚本。让我们开始吧。

打开Coursera笔记本学习这一课。你应该看看我现在在屏幕上显示的东西。

首先导入TensorFlow并检查它的版本。要导入TensorFlow，写import TensorFlow stf。然后按Shift+Enter来运行单元格。

在下面的单元格中，输入tf**version**。按Ctrl+Enter运行单元格。

```
In [1]: # Import TensorFlow

        import tensorflow as tf
```

```
In [2]: # Check its version

        tf.__version__
```

```
Out[2]: '2.0.0'
```

你应该会发现你已经导入了TensorFlow版本2。本笔记本底部的单元格包含创建和训练神经网络的代码，该神经网络可以对取自MNIST数据集的手写数字图像进行分类。

```
In [*]: # Train a feedforward neural network for image classification

        import numpy as np

        print('Loading data...\n')
        data = np.loadtxt('./data/mnist.csv', delimiter=',')
        print('MNIST dataset loaded.\n')

        x_train = data[:, 1:]
        y_train = data[:, 0]
        x_train = x_train/255.

        model = tf.keras.models.Sequential([
          tf.keras.layers.Dense(16, activation='relu'),
          tf.keras.layers.Dense(10, activation='softmax')
        ])

        model.compile(optimizer='adam',
                      loss='sparse_categorical_crossentropy',
                      metrics=['accuracy'])

        print('Training model...\n')
        model.fit(x_train, y_train, epochs=3, batch_size=32)
```

暂时不要担心这段代码的细节，下周的课程结束时您就会理解它了。它所做的就是加载一个数据集，声明一个模型，然后训练这个模型。通过单击单元格并按Ctrl+Enter来运行代码。

数据集的加载大约需要30秒。所以耐心点，给自己泡杯茶。

一旦数据集被加载并且模型开始训练，您将看到一个打印输出，通知您模型的训练进度及其性能。

```
Loading data...

MNIST dataset loaded.

Training model...

Train on 60000 samples
Epoch 1/3
60000/60000 [==============================] - 11s 189us/sample - loss: 0.4137 - accuracy: 0.8855
Epoch 2/3
60000/60000 [==============================] - 11s 178us/sample - loss: 0.2386 - accuracy: 0.9331
Epoch 3/3
60000/60000 [==============================] - 10s 175us/sample - loss: 0.2094 - accuracy: 0.9420
Model trained successfully!
```

当模型完成训练后，打印输出会告诉您模型训练成功。恭喜你，你已经用TensorFlow 2训练了你的第一个模型。如果您不熟悉本笔记本中的代码，请不要担心，它的概念将在以后的课程中讨论。在这节课中，我们只是检查你是否能在Coursera上运行TensorFlow 2。下节课再见。

5.What's new in TensorFlow2

欢迎回来。在这个视频中，我想谈谈TensorFlow 2版本中所做的一些重大改变。正如我在介绍视频中提到的，TensorFlow 2是TensorFlow 1的一大进步。TensorFlow开发团队的重点是，在不丧失TensorFlow 1范例给我们带来的好处的情况下，让它真正易于使用。我还想重复一下那个视频中的一点，那就是你不需要在这门课上成功地使用过TensorFlow 1。但是，即使你以前没有使用过TensorFlow 1，你可能会了解一些关于你的代码在这个早期版本中是如何组织的以及为什么?因为它将阐明TensorFlow 2的一些优点，并让你对幕后发生的事情有更多的了解。

```python
x = tf.placeholder(tf.float32, [None, 20])
y = tf.placeholder(tf.float32, [None, 5])

W = tf.get_variable("W", shape=(20, 5), initializer=tf.initializers.glorot_normal())
b = tf.get_variable("b", shape=(5,), initializer=tf.initializers.zeros())
h = tf.matmul(x, W) + b
loss = tf.losses.mean_squared_error(h, y)
opt = tf.train.GradientDescentOptimizer(0.001)
train_op = opt.minimize(loss)

max_steps = 1000
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for step in range(max_steps):
        x_batch, y_batch = next(train_batch)
        _, batch_loss = sess.run([train_op, loss], feed_dict={x: x_batch, y: y_batch})

```

如果你以前使用过TensorFlow 1，那么你在这里看到的代码可能对你来说很熟悉。

在TensorFlow 1中，乍一看，它看起来相当复杂和混乱

```python
x = tf.placeholder(tf.float32, [None, 20])
y = tf.placeholder(tf.float32, [None, 5])

W = tf.get_variable("W", shape=(20, 5), initializer=tf.initializers.glorot_normal())
b = tf.get_variable("b", shape=(5,), initializer=tf.initializers.zeros())
h = tf.matmul(x, W) + b
```

因为你首先必须设置变量和定义模型的操作，以及损失函数和优化器。

```python
loss = tf.losses.mean_squared_error(h, y)
opt = tf.train.GradientDescentOptimizer(0.001)
train_op = opt.minimize(loss)
```

你希望用来训练模型，以便在实际运行任何东西之前，一切都准备就绪。
所以你不能立即使用我在上面创建的变量w和b。

```
W = tf.get_variable("W", shape=(20, 5), initializer=tf.initializers.glorot_normal())
b = tf.get_variable("b", shape=(5,), initializer=tf.initializers.zeros())
```

相反，这些变量和操作将定义一个计算图形，它基本上说明了模型的输入、输出和参数是如何连接起来的。你还需要定义这些叫做占位符的东西作为你计算的入口点在你可以输入数据的图上。

```
x = tf.placeholder(tf.float32, [None, 20])
y = tf.placeholder(tf.float32, [None, 5])
```

一旦图被构建，你就可以在一个叫做TensorFlow Session的东西里运行图。

```
with tf.Session() as sess:
```

但是首先，您需要在使用变量之前对它们进行初始化。

```
sess.run(tf.global_variables_initializer())
```

然后，您可以运行训练操作，并使用称为提要字典的东西计算传入数据的损失。

```
x_batch, y_batch = next(train_batch)
_, batch_loss = sess.run([train_op, loss], feed_dict={x: x_batch, y: y_batch})
```

TensorFlow 2消除了很多这些并发症。所以我想总结一下TensorFlow 2中的一些主要的发展，它们让你的工作流程变得更加简单。
动态图机制已经存在于TensorFlow 1中，但在TensorFlow 2中，这是默认行为。

```
New in Tensorflow 2.0

    - Eager execution by default
```

即时执行意味着可以直接使用TensorFlow变量和张量。不需要运行初始化器或启动这些会话对象中就可以获取它们的值

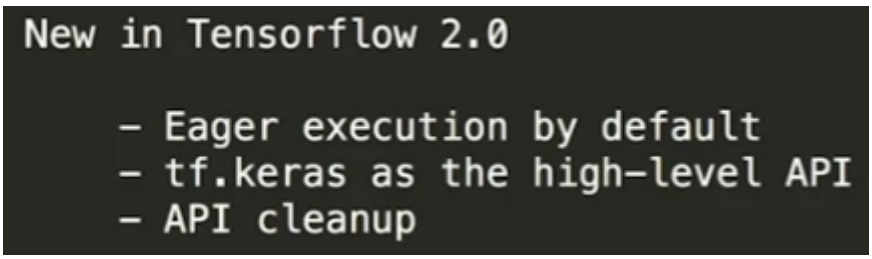
```
1  import tensorflow as tf
2
3  x = tf.Variable([1., 2.], name='x')
4  print(x)
```

听起来很简单。但这确实是一个很大的变化，从整个图表构建和运行在一个会话范式。

```
New in Tensorflow 2.0

    - Eager execution by default
    - tf.keras as the high-level API
```

在TensorFlow 2中，keras已经成为TensorFlow默认的高级API。因此，完整的keras API现在被包装成TensorFlow安装的一部分，并与TensorFlow无缝集成。

keras的优点我们将在稍后的课程中看到，它真的很容易使用，而且仍然非常灵活。即使我说它是TensorFlow的高级API，你也可以使用keras来完成大部分模型开发。



最后我想说的是TensorFlow的API做了很多工作。在此之前，有一些不一致的地方，比如函数在不同的地方做非常相似的事情，等等。在TensorFlow 2中，这个问题已经得到了很大的解决。当我们稍后浏览文档时，您将看到，它被很好地组织成模块，使您更容易找到需要的东西。

当然还有很多新的关于TensorFlow 2得改变，我在这里没有提到，但我想强调一些主要的变化。就像我说的，即使你以前从未使用过TensorFlow 1，我也希望这能给你一点关于TensorFlow 2的知识，让你对在后台发生的事情有个有一个认识。

记住，对于图构建范式，其主要目的是优化代码，使其运行得更快。在TensorFlow 2中，这实际上仍在发生，但它发生在后端。例如，当你使用高级的keras API时，你就不用担心它了。

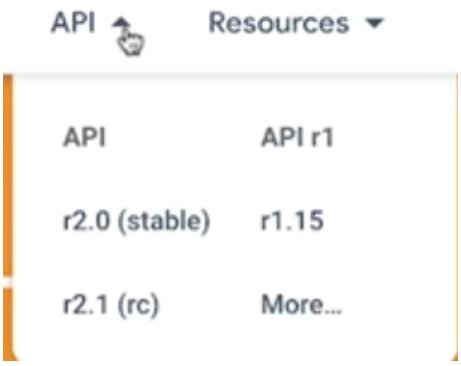
但是，当我们更加熟练得使用TensorFlow并开始编写更低级别的代码时，您将在稍后的课程中看到，您仍然可以通过将部分代码编译成单个图，以获得性能收益。所以至少了解一下图表编译的这个阶段是有帮助的，这是在TensorFlow的后端进行的。下个视频再见。

6.TensorFlow documentation

欢迎回来。我会鼓励你们在这门课中使用的一个重要资源是TensorFlow文档。我们将在这门课中涉及到TensorFlow库中的很多不同部分，这门课的结构方式意味着它是非常独立的，但我相信你仍然会发现很多有用得东西，比如回到TensorFlow文档，把它作为参考来提醒你语法或函数名，或者在库的特定部分探索更多可用的选择。

所以经常查阅文档是一个很好的习惯。要浏览TensorFlow文档，请访问TensorFlow.org。在顶部，你可以看到API文档的链接，顺便说一下，有一些很棒的教程和TensorFlow指南，你可以通过这个链接找到，所以一定要看看这个链接。

但现在我们想看一下文档如果你把鼠标悬停在API链接上，你可以看到API文档的不同版本。我们将看看最新的稳定版本，在这个记录的时候是2.0版本。



在第一页，你可以得到一个概述，你可以浏览库中的所有模块，你会看到这里有相当多的模块。在页面的下方，类和函数都列出了，如果我滚动到底部，你可以看到其他成员，你会发现像TensorFlow类型这样的东西。

好吧。在页面的左边，你可以找到所有这些按字母顺序排列的类和函数，你可以点击查看任何一个的文档。

现在如果我最小化这个列表，

你现在可以看到TensorFlow中许多模块的概述。我特别想让你们注意两个模块因为我们在这门课上会经常用到它们。第一个是tf.keras模块，我们马上就会用到。如果我们打开这个模块,看看里面都有什么,你会看到上面有一些类,输入、模型和顺序,你就会明白这些都是在今后的课程中,有更多的模块我可以看一看在TF.keras模块。同样，你可以点击其中任何一个打开文档。

我想指出的另一个模块，如果我最小化TF.keras模块，就是TF.data模块。这是TensorFlow中的另一个重要模块，当我们谈到数据管道时,我们会在课程中进一步讨论。所以，请花一些时间来研究文档，只是暂时的，只是为了熟悉它的布局。你应该把它看作是我们这门课要讲的内容的参考资料。我总是鼓

励你们去查阅文档了解你们在整个课程中学习到的不同对象。我希望在下一周的课程中，能开始在TensorFlow中构建我们的第一个深度学习模型。

week 2

1.Welcome to week 2 - The Sequential model API

欢迎来到本周的TensorFlow 2课程。现在你已经设置好了所有的东西，你已经准备好进入TensorFlow并开始构建你的第一个模型。就像我之前说的，在TensorFlow 2中，一个大的驱动力是易用性。在最高层次上，在TensorFlow中开始构建深度学习模型的最简单的方法是使用Keras API，这也是我们将要开始关注的。在这周，我们将介绍keras sequential API并使用它来建立我们的第一个模型，训练它们并从中做出预测。在此过程中，您将看到如何根据任务选择不同的优化器来训练神经网络中不同的损失函数，以及如何在训练期间跟踪任意数量的指标。在整个课程中，会有一系列的讲座视频，我会向你们展示一些新的东西，并用幻灯片来演示。还有一些编写教程的视频，由一个研究生助教，或者叫GTA，来使用我介绍过的API并将它们付诸实践在一个教程笔记本上。每周都会有一个不同的GTA作为编码教程。这周，Sergio将和你们一起学习sequential model API。在这周末，你将拥有所有你需要的东西，从你自己的TensorFlow深度学习模型中设计、构建、训练和预测。你们会在编程作业中把这些都付诸实践你们会训练一个计算机视觉模型来使用数据集对"手写数字"进行分类。让我们开始吧。

2.What is keras?

在整个课程中，我们将使用TensorFlow库中的Keras API来开发模型。我们先来谈谈Keras是什么。Keras项目是由Francois Chollet编写的，它是一个高级的神经网络API，开发的具体目标是让它更容易构建巨大的深度的学习模型。它是一个没有后端的API，但实际上支持多个后端实现。在TensorFlow 2中，它被用作高级API。尽管它是高级的，易于使用的，但它仍然非常灵活，可能99%的情况，你都可以使用Keras API来实现。所以我们会在这门课中频繁地使用它。你可以在Keras.io访问Keras的主页，在那里我们可以阅读更多关于Keras项目及其开发原则的信息。这里有各种不错的指南和教程，看看吧。所以当你安装TensorFlow 2时，你会把所有这些打包成命名空间TensorFlow. keras中的库的一部分。这就是我们这门课要用到的东西。如果你浏览TensorFlow.org上的API，你也可以看到这个。这里你可以看到tf.keras，里面是keras库。虽然你有两种文档你可以参考，但我还是建议使用TensorFlow文档来作为你的参考。

3.Building a sequential model

欢迎回来。在这个视频中，我们将介绍keras序列类，并使用它来构建简单的模型。事实上，您可能会发现，您所使用的大多数神经网络，都可以使用顺序类来构建。这是一种非常简单直观的构建深度学习模型的方法。我们将在这门课中使用sequential model。那么让我们来看看如何使用它。

首先，看看我们的导入。我们从tensorflow.keras.models导入 sequential class。

```
from tensorflow.keras.models import Sequential
```

在这里，我们还用tensorflow.keras.layers.导入了dense layer层

```
from tensorflow.keras.layers import Dense
```

我们构建模型的方法非常简单。我们只是创建了一个顺序类的实例。

```
model = Sequential([
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

这里我调用实例模型。我们所做的就是传递一个keras层的列表。这个模型是一个只有一个隐藏层的前馈网络。你可以看到，当你创建一个密集的层时，你总是需要说明该层中应该有多少神经元。

```
64
10
```

这是我们在第一个论点中得到的。你也可以像我在这里做的那样，有选择地传递一个激活函数。

```
activation='relu'
```

所以隐藏层有一个relu激活。如果您没有为激活传递任何信息，那么致密层默认情况下将有一个线性激活或没有激活函数。输出层有10个单元，并使用softmax激活函数。

```
Dense(10, activation='softmax')
```

你可以想象这可能是一个分类任务的模型其中有10个类。你可能注意到的一件事是我们没有说输入的大小或形状是什么。所以如果你考虑这个模型定义，我们还没有足够的信息来创建所有模型的权重和偏差。这样做实际上是可以的，并把它留到训练阶段，当你把数据输入到模型中，在权重和偏差产生之前。

```
Dense(64, activation='relu', input_shape=(784,)),
```

或者，您可以显式地告诉模型，在构建阶段输入数据的形状是什么。像这里一样，我指定每个输入数据示例都是一个大小为784的一维向量。

```
input_shape=(784,)
```

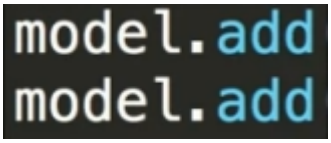
在这种情况下，权重和偏差将被创建并直接初始化。

这是另一种构建完全相同模型的方法。

```
model = Sequential()

model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

您可以使用模型，而不是传递一个层列表。

使用add method以向模型追加其他层。



这张幻灯片上的模型和前一张幻灯片上的模型是等价的。选择哪种方法，我认为在很大程度上只是个人喜好的问题，但也可能取决于代码的结构。例如，你可能会发现像我在这里做的那样一次添加一个层是很有用的，如果你想要模型构建代码依赖于一些条件或循环或类似的东西。在你付诸实践之前，这里还有最后一个例子。

```
model = Sequential([
    Flatten(input_shape=(28, 28)),  # (784, )
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

在这个模型中，我导入了两种图层类型;flatten and dense。

```
from tensorflow.keras.layers import Flatten, Dense
```

你可以看到我在做第一层，the flatten layer。

```
Flatten(input_shape=(28, 28))
```

同样，我在我的模型中给第一层输入形状，以便在定义模型实例时创建权重和偏差。请注意，这里的输入形状是二维的，这就是为什么我要将每个输入数据展平，将其展开成一个尺寸为784的一维向量，然后再将其发送到第一个dense layer。现在您已经了解了如何构建sequential models。现在是时候自己编写一些示例了。正如我在介绍视频中提到的，在本课程中会有编码的教程视频，GTA会带你通过教程笔记本来练习我在这些讲座视频中涉及的内容。在Coursera shell中，您将看到我的讲座视频后面的教程项被标记为编码教程。笔记本下面总是会有一个视频。在视频中，助手将带你学习教程笔记本。正如您本周所看到的，Sergio正在做编码教程。每周有几个编码教程视频项目。在每个视频中，GTA将带你学习编码教程笔记本的一部分。在一个单独的窗口中打开编码教程笔记本是一个好主意，这样您就可以跟随屏幕播放并自己编写示例代码。当然，你可以在任何时候暂停视频，在笔记本上尝试一些东西，你可以用它来做实验。所以，现在开始，打开这周的笔记本，准备好学习下面的编码教程

4.Convolutional and pooling layers

Hi and welcome back. Now you know how to build simple neural networks using the sequential model. The models we've built so far were all feed-forward networks. Which are also called Multilayer Perceptrons or MLPs. In this video we'll see how to build convolutional neural networks by including convolutional and pooling layers in our model.

Here's a typical setup for a convolutional network.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Conv2D, MaxPooling2D

model = Sequential([
    Conv2D(16, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D((3, 3)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

I'm importing the Flatten and Dense layers that we use last time.

```
from tensorflow.keras.layers import Flatten, Dense
```

Plus, I'm also using the Conv2D and MaxPooling2D layers.

```
from tensorflow.keras.layers import Flatten, Dense, Conv2D, MaxPooling2D
```

And here's the model definition.

```
model = Sequential([
    Conv2D(16, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D((3, 3)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

The first layer is a convolutional layer.

```
Conv2D(16, (3, 3), activation='relu', input_shape=(32, 32, 3))
```

And these layers have to required arguments. The first is the number of filters.

```
(16,
```

So, we have 16 filters in this layer. And the second is the shape of the convolutional kernel.

```
(3, 3),
```

So, let's take a look at our input shape.

```
input_shape=(32, 32, 3))
```

This input could well be an image with 32 by 32 pixels and 3 color channels. So, our kern el is convolving over the image with a window of 3 by 3 pixels using 16 filters. We're also passing the output through a relu activation function.

```
activation='relu'
```

The second layer is a MaxPooling layer.

```
MaxPooling2D((3, 3))
```

This layer has one required argument, which is the pooling window size. And that's also 3 by 3 pixels. This is followed by a Flatten layer.

```
Flatten()
```

Which, remember, is unrolling each data example into a long one dimensional vector ready to be passed through to the final two Dense layers. So, just to make sure we understand what's going on here. Let's take a look at the tensor shapes after each of these layers.

```
Conv2D(16, (3, 3), activation='relu', input_shape=(32, 32, 3)),  # (None, 30, 30, 16)
MaxPooling2D((3, 3)),                                            # (None, 10, 10, 16)
Flatten(),                                                       # (None, 1600)
Dense(64, activation='relu'),                                    # (None, 64)
Dense(10, activation='softmax')                                  # (None, 10)
```

You'll notice that the first dimension of every tensor has a value of none.

```
None
None
None
None
None
```

And that's because the first dimension will always be the batch size. And that batch size is flexible. We can pass any number of examples in a batch to the model. Tensor flow represents this with the value none in the tensor shape. So, let's ignore the batch size for the moment and look at the remaining shape dimensions. Remember, the input shape is 32 by 32 by 3. After being processed by the convolutional layer, this becomes a tensor of shape 30 by 30 by 16.

```
(None, 30, 30, 16)
```

The 16 is because we created the layer with 16 filters. The shape 30 by 30 is because these convolutional layers have no 0 padding by default. Sometimes you'll hear that referred to as valid padding. And also, the default stride is one. So, with a kernel of shape 3 by 3, that results in a 30 by 30 by 16 shaped tensor output. Likewise, the pooling layer has a 3 by 3 window. And these are non-overlapping. So, that downsamples the input to a 10 by 10 by 16 shaped tensor.

```
(None, 10, 10, 16)
```

The Flatten layer just unrolls this into a one dimensional vector, ignoring the batch size. So, 10 times 10 times 16 is 1,600.

```
(None, 1600)
```

From here on, the shapes are easy to understand. This is just a Dense layer with 64 units.

```
(None, 64)
```

And the final Dense layer has 10 units.

```
(None, 10)
```

Here's how it would look like if we wanted SAME padding.

```
model = Sequential([
    Conv2D(16, kernel_size=(3, 3), padding='SAME',
           activation='relu', input_shape=(32, 32, 3)),  # (None, 32, 32, 16)
    MaxPooling2D(pool_size=(3, 3)),                      # (None, 10, 10, 16)
    Flatten(),                                           # (None, 1600)
    Dense(64, activation='relu'),                        # (None, 64)
    Dense(10, activation='softmax')                      # (None, 10)
])
```

That just makes sure that the spatial dimensions don't change. So, the 32 by 32 by 3 shaped input goes to a 32 by 32 by 16 output.

```
(None, 32, 32, 16)
```

And the rest of the shapes for the other layers of the same. And the last thing I want to show you is just a shortcut. Often the window shapes of convolutional and pooling layers have the same size in each dimension. In that case, you can just write the kernel size or the pool size as a single integer. So here, this is the same as writing 3 by 3 explicitly, as we did before.

```
 kernel_size=3,
ivation='relu',
2D(pool_size=3)
```

In this lecture video, we've seen how to build convolutional and pooling layers into our network. With just these few layers we've seen so far, you can already develop some very powerful deep learning models. So, now it's time for you to practice coding up some convolutional neural networks with Sergio in the next coding tutorial.

5.The compile method

Welcome back. So far, we've seen how to build feedforward and convolutional neural networks in TensorFlow using the Keras sequential class. But to start training our network on data, we're also going to need to define a loss function that will give us a measure of our model's performance and the optimization algorithm. In this video, we're going to talk about how to define these things so we're ready for training. Let's take a look at a simple binary classification network.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(64, activation='elu', input_shape=(32,)),
    Dense(1, activation='sigmoid')
])

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

This model takes a one-dimensional input of size 32, passes it into a fully connected layer with 64 units and an exponential linear unit activation function.

```
input_shape=(32,)
```

```
Dense(64, activation='elu', input_shape=(32,))
```

That's followed by a final dense layer of just one neuron with a sigmoid activation function.

```
Dense(1, activation='sigmoid')
```

Okay, down here is the new bit of code that we're interested in.

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

This is what is defining the loss function and the optimizer for the network. And as you can see, we're defining the loss function to be the binary_crossentropy which makes sense for this network and task.

```
loss='binary_crossentropy'
```

And the optimizer is sgd or stochastic gradient descent.

```
optimizer='sgd'
```

These options are being passed into the compile method of our sequential model object. We can also optionally define a set of metrics that we want to keep track of as the model is training.

```
model.compile(
    optimizer='sgd',
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

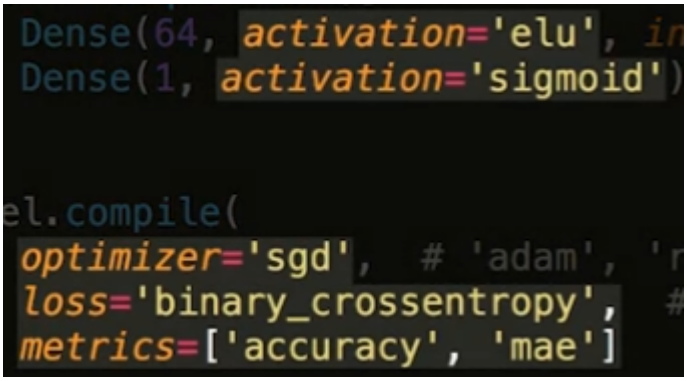
These metrics will be calculated for each epoch of training along with the evaluation of the loss function on the training data. We'll see this in action later when we start training some models. For now though, we just need to know that we can pass
a list of metrics into the compile method. So here we're just passing in a list with one metric that is the accuracy. Of course, there are lots of options for each one of these keyword arguments.

```
# 'adam', 'rmsprop', 'adadelta'
entropy',  # 'mean_squared_error', 'categorical_crossentropy'
```

For example, we could instead choose the adam optimizer or rmsprop or adadelta. For the loss function, you could choose
the mean_squared_error although that might be more appropriate for a regression task or categorical cross entropy, for example. In the list of metrics, I've added one more.

```
metrics=['accuracy', 'mae']
```

mae here stands for mean absolute error. So with this call to the compile method would be keeping track of two performance metrics in addition to the binary_crossentropy loss. One thing that you'll be noticing is that Keras is giving us a really nice usable interface for setting up these models. And there are a lot of readable strings that we can pass in to many of the options in the Keras API like we see here in the activation functions and the options in the compile method.



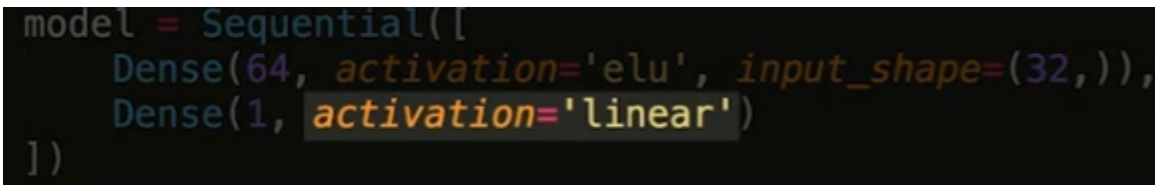
It's definitely worth knowing though that each of these strings is a reference to another object or function and we can always use that object or function directly. For example, all the options I have here in the compile function could be given like this.

```
model.compile(
    optimizer=tf.keras.optimizers.SGD(),
    loss=tf.keras.losses.BinaryCrossentropy(),
    metrics=[tf.keras.metrics.BinaryAccuracy(), tf.keras.metrics.MeanAbsoluteError()]
)
```

Instead of passing in the string SGD, I'm directly passing in an SGD object that comes from the tf.keras.optimizers module. Similarly for the loss, the BinaryCrossentropy loss is available in the tf.keras.losses module. And the metrics I'm using can be given like this, the BinaryAccuracy and the MeanAbsoluteError metrics are in the tf.keras.metrics module. The reason why you might want to do this is because it gives you greater flexibility as many of these objects themselves have options that you might want to have control over. This slide shows you what I mean.

```
model.compile(
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9, nesterov=True),
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.BinaryAccuracy(threshold=0.7), tf.keras.metrics.MeanAbsoluteError()]
)
```

With stochastic gradient descent, an important parameter is the learning_rate. By default, the learning_rate is set to 0.01 but here I'm creating in the SGD optimizer objects with learning_rate 0 .001. I'm also using momentum with a value of 0.9. By default, the momentum value is 0. So there is no momentum. And an extra option I can choose to set is whether or not to use nesterov momentum which here I'm setting to be True. Let's take a look now at the BinaryCrossentropy function. That also takes some options and here I'm setting the option from_logits=True. If you are watching closely, you might have also seen that I've changed the activation function in the last layer of the network from sigmoids to linear.



In other words, now, there is no activation function and I could as well have left this argument out as the linear activation is the default. And so the network is now outputting the logits which is any real value before it is squeezed through the sigmoid activation function. The from_logits=True option tells the model that it should take the output of the network. And the loss function itself should handle the squeezing of the output through the sigmoid activation. Mathematically, there's no difference between this and what I had before but this way turns out to be a more numerically stable approach. Finally, the BinaryAccuracy also has an option and we can choose what the threshold is for predictions to be classed as positive.

```
tf.keras.metrics.BinaryAccuracy(threshold=0.7)
```

The default is 0.5 but here I'm setting it to be 0.7. So now you've seen how to build neural networks as sequential models and set the optimizer loss function and metrics using the compile method. Once this has been done, you've got everything you need to start training the model on data. And so in the next lecture video, we'll see how to do just that.

6.The fit method

In the previous videos, you've seen how to build and compile models. So now a deep-learning model is fully defined. We have the architecture, the number of layers, activation functions, and so on. We've also assigned an optimizer to the model,

possibly with some options of its own, and loss function, and maybe some metrics to track. So now it's time to train the model.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(64, activation='elu', input_shape=(32,)),
    Dense(100, activation='softmax')
])

model.compile(
    optimizer='rmsprop',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

model.fit(X_train, y_train)
```

Here we can see each step outlined in code. We've defined the feed-forward network with 100 way softmax output for a classification task,

```
model = Sequential([
    Dense(64, activation='elu', input_shape=(32,)),
    Dense(100, activation='softmax')
])
```

and we've compiled the model with the RMSProp optimizer, the categorical_crossentropy loss function, and we're going to track the categorical accuracy metric.

```
model.compile(
    optimizer='rmsprop',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

The last line here shows how simple it is to launch the training.

```
model.fit(X_train, y_train)
```

You just need to call model.fit and pass in the training inputs, X_train and the labels, y_train. We're going to assume for now that both X_train and y_train are Numpy arrays, where all of the dataset inputs have been stacked together into a single array,

X_train, and all the targets or outputs are in y_train.

```
#  X_train: (num_samples, num_features)
#  y_train: (num_samples, num_classes)
```

So the first dimension of each array corresponds to the number of examples in the training set.

```
(num_samples,
(num_samples,
```

For example, if each input data point is a one-dimensional array, X_train would be a two-dimensional array with the number of samples in the first dimension and the number of features in the second.

```
(num_samples, num_features)
```

Y_train would be a two-dimensional array with the number of samples in the first dimension and the number of classes in the second.

```
(num_samples, num_classes)
```

Here, I'm assuming that the labels have been represented as a one-hot vector, so each row of y_train is a vector of length num_classes which is all zeros, except for a one in the place corresponding to the correct class.

```
model.compile(
    optimizer='rmsprop',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

model.fit(X_train, y_train)

#  X_train: (num_samples, num_features)
#  y_train: (num_samples,)
```

Or if all the labels have a sparse representation, so just a single integer for each label, then y_train could be

a one-dimensional array with length equal to the number of samples.

```
(num_samples,)
```

Notice that in this case, we should choose the sparse_categorical_crossentropy loss function.

```
loss='sparse_categorical_crossentropy'
```

Passing these arrays into model.fit, we will then train the model for one pass through the training data or one epoch. You'll probably want to train a model for more than just one epoch, though, and so here, I'm passing in the optional keyword argument, epochs equals 10.

```
model.fit(X_train, y_train, epochs=10)
```

So the training will make 10 complete passes through the dataset. Another option I want to mention is the batch size.

By default, the batch size is set to 32 whenever you pass in the dataset inputs and outputs as single Numpy arrays as we're doing here.

```
model.fit(X_train, y_train, epochs=10, batch_size=16)
```

If you want to change that to use a different batch size, you can just use the batch size argument. Like here, I'm setting the model to train on minibatches of size 16. There are quite a few more optional keyword arguments for model.fit, and we'll come back to these in future videos where we start to look at more sophisticated features in our training. Finally, an important point is that calling model.fit actually returns something, and it returns something called a TensorFlow history object.

```
history = model.fit(X_train, y_train, epochs=10, batch_size=16)
```

This object contains a record of the progress of the network during training in terms of the loss and the metrics that we defined when we compiled the model. This object is actually an example of something called a callback, which we'll look at later in the course. In the following coding tutorial, GTA will show you how to access the record of the loss and metrics for each epoch of training in the history object.

So you're already at the stage of being able to build neural network models, compile them with a given loss function and

optimizer and any additional metrics, and now train them using model.fit. Thanks to the Keras API, we can do all this in just a few lines of code. So now let's open up the Notebook again and in the next coding tutorial, you'll start training your first deep-learning models.

7.The evaluate and predict methods

Welcome back. By now, you've learned how to build deep learning models using the sequential API, how to specify optimizers, loss functions, and metrics using the compile method. In the last lecture video, you've seen how to train models using model.fit. In this video, we're now going to see how well our network has learned by evaluating it's performance on a held out test set. We're also going to be putting our network into action by getting it's predictions on unseen input data. Let's just recap what we've done so far.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([Dense(1, activation='sigmoid', input_shape=(12,))])
model.compile(optimizer='sgd', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train)
```

We defined the model using the sequential class by passing in a list of layers.

```
model = Sequential([Dense(1, activation='sigmoid', input_shape=(12,))])
```

Or remember, you could use the model.add method to add layers one-by-one. Then we compile it using model.compile and then we passed in options for the optimizer, the loss function, and a list of metrics to evaluate the performance via network with.

```
model.compile(optimizer='sgd', loss='binary_crossentropy', metrics=['accuracy'])
```

We train the model using model.fit., and here, remember, we're passing in the input data as one big NumPy array, X train and the labels also as one NumPy array, Y train.

```
model.fit(X_train, y_train)
```

For both of these arrays, the first dimension corresponds to the number of examples in the dataset. Once this has finished training, we have our trained model and the first thing we'll want to do is test it. So let's suppose that we have a separate held-out test set of data that we can test it on.

```
model.evaluate(X_test, y_test)
```

This is data that wasn't used in the training so it will give us a good measure of how well the network has learned to generalize beyond the training data. So then let's say that X tests to the test inputs and Y test to the test labels. Again, each of these arrays contains a number of data examples. The first dimension of each array corresponds to the number of examples in the test set. Running model.evaluate, will iterate over the test set and calculate the loss and the metrics on that test set. So for example, if we had defined binary cross-entropy loss function when we compiled the model,

```
loss='binary_crossentropy'
```

as well as an accuracy metric,

```
metrics=['accuracy']
```

then this loss function and this metric will be evaluated on the test set and returned by the model.evaluate call. So in that case, we can save those returned values like this.

```
loss, accuracy = model.evaluate(X_test, y_test)
```

If we had included more metrics, then they would all just be returned by model.evaluate and we could save them like this.

```
metrics=['accuracy', 'mae'])
```

```
loss, accuracy, mae = model.evaluate(X_test, y_test)
```

Hopefully, if our model has trained well, then these loss of metric values won't be too far off what we achieved on the training set. So we should use the evaluate method whenever we have a held-out test set like this to evaluate the performance of the network. But if we want to use our trained model to get predictions on unseen data, then we use the model.predict method.

```
pred = model.predict(X_sample)
```

This method doesn't take any labels of course, it just takes in an array of inputs. Here we're going to let X sample be those inputs. So example is a NumPy array of stack data inputs.

```
X_sample: (num_samples, 12)
```

The first dimension corresponds to the number of examples that we want to get predictions for and the rest of the dimensions will be the data features. In the example, I'm showing here, the number of features is 12 and notice how that corresponds to the input shape that I set when I defined the model. Model.predict will then return the outputs of the network for these given inputs. One point that often catches people out, if we're getting predictions in only one example, then we still have to have a dummy first dimension that will be equal to one. So for example, if X sample is a NumPy array, with just one input sample

```
X_sample: (1, 12)
```

and our network is a binary classifier where the final layer has just one neuron passed through a sigmoid activation,

```
[Dense(1, activation='sigmoid', input_shape=(12,))]
```

then model.predict will return just one number that we interpreted as a probability that the input we passed in belongs to the positive class.

```
pred = model.predict(X_sample)  # [[0.07713523]]
```

Here, the probability is very low. So the model prediction would be that, this input belongs to the negative class. But see how the output is a two-dimensional NumPy array.

```
# X_sample: (2, 12)

pred = model.predict(X_sample)  # [[0.07713523],
                                #  [0.94515101]]
```

The first dimension of this array, again, corresponds to the number of examples that we're getting predictions for. So if we were passing in two examples in a stack in NumPy array, and so example now is an array where the first dimension is equal to two, then we would get out a two-dimensional array where the first dimension is also equal to two.

```
pred = model.predict(X_sample)  # [[0.07713523],
                                #  [0.94515101]]
```

As a final example, let's say, a network is a multi-class classification model where there are three classes.

```
model = Sequential([Dense(3, activation='softmax', input_shape=(12,))])
model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy', 'mae'])
model.fit(X_train, y_train)

loss, accuracy, mae = model.evaluate(X_test, y_test)

# X_sample: (2, 12)

pred = model.predict(X_sample)
```

So our final layer has three neurons and softmax activation.

```
[Dense(3, activation='softmax', input_shape=(12,))]
```

You'll see I've also changed the loss-function from binary cross entropy to categorical cross entropy.

```
loss='categorical_crossentropy'
```

I'm supposing that the number of input features is the same as before, 12.

```
input_shape=(12,)
```

Let's also say that example again contains two data inputs.

```
X_sample: (2, 12)
```

So that the first dimension of this array is equal to two. Then the output of model.predict will look something like this. The array of output predictions will be a two-dimensional array as before, the first dimension will be two, which matches the first dimension of X sample as the number of examples we're getting predictions for and the second dimension will be three for the number of classes.

```
pred = model.predict(X_sample)  # [[0.93957397, 0.0189931, 0.04143293],
                                #  [0.01211542, 0.0907736, 0.89711098]]
```

Each row of this array is the network output for that data input. The output layer is a softmax layer, remember, so each layer is a set of output probabilities emitted by the softmax function and you can see how they add up to one in each case. So here, it looks like the model is predicting class one for the first example and class three for the second.

Now, you've seen how to both evaluate and train models as well as make predictions from them. In the final coding tutorial of this week, you'll apply this to the model that you've trained on the fashioned eminence dataset. So let's see how good this predictions are.

week3

1.Validation, regularisation and callbacks

Hello and welcome to this week of the course. Previously, you learned an end-to-end workflow for building, training, evaluating, and predicting from deep learning models. In this week, we're going to look at adding some important features into our training. We'll be revisiting the same workflow and methods that you've learned before, but we'll be including additional options into those methods.

A key goal in machine learning is to develop models that generalize beyond the data that they were trained on. We want models that can make good and reliable predictions in the real world. An important concept that helps us with this is model validation and selection. In this week, we'll look at how we can include held out validation sets in training rooms, as well as implementing regularization techniques such as weight decay, dropout, and early stopping.

In this week, we'll also introduce callbacks, which are really useful objects that help us to perform certain functions during training and evaluation runs. You can use callbacks for example to monitor the model's performance on a held out a validation set, and terminate the training run according to some criteria that we specify.

By the end of the week, you'll be able to properly validate and regularize your models to avoid issues like over fitting and help them generalize better to unseen data. Just as in the previous week, you'll build up your expertise in applying these concepts in a series of coding tutorials. This week, Melissa will walk you through those tutorials. In the programming assignment for this week, you'll put the full workflow into practice and implement validation, regularization and callbacks in a deep learning model for the well-known Iris dataset. So let's make a start.

2.Validation sets

In this video, we'll see how to introduce validation sets into the training room. Validation sets is useful as a measure of how well our model is performing outside of the training set. So the model never uses validation set to train on but its performance is evaluated on the validation set during training. So here, I've built a simple feedforward network.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

model = Sequential()
model.add(Dense(128, activation='tanh'))
model.add(Dense(2))

opt = Adam(learning_rate=0.05)
model.compile(optimizer=opt, loss='mse', metrics=['mape'])

model.fit(inputs, targets, validation_split=0.2)
```

It's got a hidden layer with a 128 neurons with tanh activation and linear output layer with two neurons.

```
model.add(Dense(128, activation='tanh'))
model.add(Dense(2))
```

So this could be a regression model where the target output is two-dimensional. We're compiling the model with the Adam optimizer, a mean squared error loss, and we've chosen mean absolute percentage error as a performance metric.

```
opt = Adam(learning_rate=0.05)
model.compile(optimizer=opt, loss='mse', metrics=['mape'])
```

Just as a reminder from earlier, I'm creating the optimizer from the tensorflow.keras.optimizers module, so that I can change

the learning rate for the Adam optimizer to 0.05.

```
from tensorflow.keras.optimizers import Adam

model = Sequential()
model.add(Dense(128, activation='tanh'))
model.add(Dense(2))

opt = Adam(learning_rate=0.05)
```

Now let's say that our dataset inputs are stacked into one big NumPy array which here we're calling inputs.

```
model.fit(inputs, targets, validation_split=0.2)
```

So remember the first dimension of this array would be the number of examples in the training sets and the remaining dimensions would be for the features. Similarly, the first dimension of the NumPy array targets will be the same, number of samples in the dataset.

```
model.fit(inputs, targets, validation_split=0.2)
```

The second dimension here should be equal to two, as this needs to match up with the dimension of the output layer of our network.

```
model.add(Dense(2))
```

Let's say then that we want to track the performance of our model on a separate held-out validation set. One way we can do this is to pass in the validation split keyword argument.

```
model.fit(inputs, targets, validation_split=0.2)
```

What this does is to automatically split the inputs in the targets into separate training and validation sets. The 0.2 you see here means that 20 percent of the data will be held back for validation. The model's performance is then recorded on both the training and validation sets. So you remember that the model.fit method returns the history object.

```
history = model.fit(inputs, targets, validation_split=0.2)
```

Before you saw how that history object records the training set loss and metrics over the course of the model training, now this history object also records the performance on the validation set. You might also remember that these values were stored inside the history attributes of the history object and this attribute is a Python dictionary.

```
print(history.history.keys())  # dict_keys(['loss', 'mape', 'val_loss', 'val_mape'])
```

If we were to take a look at this dictionary, would see that it now not only has keys for the training set loss and metric values,

but also for the validation loss of metric values.

```
print(history.history.keys())  # dict_keys(['loss', 'mape', 'val_loss', 'val_mape'])
```

So then we can take a look at how our model performed on the validation set. Sometimes, datasets have already been packaged up for us with the training and test split.

```
import tensorflow as tf

(X_train, y_train), (X_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
```

You've seen this already before when you loaded the fashion_mnist dataset with the tf.keras.datasets module. You could use this split as a ready-made training and validation split. So instead of getting the model.fit method to make this split for us, we explicitly give it the validation set.

```
model.fit(X_train, y_train, validation_data=(X_test, y_test))
```

If we want to do that, then we can use the validation data keyword argument.

```
validation_data=(X_test, y_test)
```

This argument takes a tuple of inputs and outputs, as before the model will then record the performance on this validation set during the training room. Finally, we might also want to make a training and validations with our ourselves, before feeding them to the model.fit method.

```
from sklearn.model_selection import train_test_split

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.1)

model.fit(X_train, y_train, validation_data=(X_val, y_val))
```

For example, we could use the train test split function from sklearn to make a training and validation split and then feed them into the fit method, again using the validation data keyword argument. So now we've seen how to include validation sets into our training room using either the validation split keyword argument or the validation data keyword argument. This is a really important feature to build into our training workflow.

In the first coding tutorial of this week, you'll put this into practice yourself on a simple dataset, so you can see how well the model performs on a held out validation set.

3.Model regularisation

Welcome back. So we've now seen how to monitor the progress and performance of our model on a held out the validation set using two different keyword arguments in the model.fit method. Now, we'll see how you can include regularization techniques into the model training that have the effect of constraining the model capacity in preventing overfitting. In particular, we're going to look at using L2 weight regularization, which is also known as weight decay in a context of neural networks, as well as L1 weight regularization and you'll see how to include Dropouts in your models. Let's start with adding in weight decay into our model.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(64, activation='relu',
          kernel_regularizer=tf.keras.regularizers.l2(0.001)),
    Dense(1, activation='sigmoid')
])
```

Here we have a feedforward network with a single hidden layer.

```
Dense(1, activation='sigmoid')
```

This is a binary classification model. As you can see, the output has just one neuron and the sigmoid activation function. We can add in weight decay when we define our model. Dense layers as well as convolutional layers have an optional kernel regularizer, keyword argument.

```
kernel_regularizer=tf.keras.regularizers.l2(0.001))
```

To add in weights decay or L2 regularization. We set the kernel regularizer argument equal to tf.keras.regularizes.L2 object. This object is created with one required arguments, which is the coefficient that multiplies the sum of squared weights in this layer. Remember that a dense layer has weights and biases, and the weight matrix is sometimes called the kernel. Here we've set the weight decay coefficients to be equal to 0.001.

```
kernel_regularizer=tf.keras.regularizers.l2(0.001))
```

Once we've built the model together with the weight decay, we can compile and fit the model is normal.

```
model.compile(optimizer='adadelta', loss='binary_crossentropy', metrics=['acc'])
model.fit(inputs, targets, validation_split=0.25)
```

The weight decay penalty term is automatically added to the loss-function when we compile the model. So here our loss-function would be the binary cross entropy calculated using the model predictions

```
model.compile(optimizer='adadelta', loss='binary_crossentropy', metrics=['acc'])
```

and the true labels plus the sum of squared weights of the first layer around model multiplied by the weight decay coefficient of 0.001. This sum of squared weights term has the effect of penalizing large values of the weights, which encourages the model to find the simpler function that fits the data. This means the model is less likely to overfit to the training set. Instead of L2 regularization, we could use L1 regularization.

```
model = Sequential([
    Dense(64, activation='relu',
          kernel_regularizer=tf.keras.regularizers.l1(0.005)),
    Dense(1, activation='sigmoid')
])
```

In this case, the penalty term is a sum of absolute weight values, instead of a sum of squared weight values, we can use L1 regularization in a very similar way. This time we're creating a tf.keras.regularizers. L1 object and passing that to the kernel regularizer argument. The regularization coefficient this time is 0.005.

```
kernel_regularizer=tf.keras.regularizers.l1(0.005)),
```

L1 regularization has the effect of pacifying the network weights. Or in other words, it results in some of those weights being set to zero. In fact, there's no reason why you couldn't use both L1 and L2 regularization.

```
model = Sequential([
    Dense(64, activation='relu',
          kernel_regularizer=tf.keras.regularizers.l1_l2(l1=0.005, l2=0.001)),
    Dense(1, activation='sigmoid')
])
```

Here we've created a tf.keras.regularizers.l1l2 object and pass that to the kernel regularizer argument. The coefficients for the regularizers can be set independently.

```
kernel_regularizer=tf.keras.regularizers.l1_l2(l1=0.005, l2=0.001)),
```

So now the regularizer object has got two keyword arguments, one the L1 regularization and one for the L2 regularization.

```
(l1=0.005, l2=0.001)),
```

In all of these examples so far, we've only been applying regularization to the weights matrix of the first dense layer. It's quite typical to only apply regularization to the weight matrix or the kernel of a dense or convolutional layer. However, it is perfectly possible to include regularization for the buyers as well. Here we can see how to do just that.

```
model = Sequential([
    Dense(64, activation='relu',
          kernel_regularizer=tf.keras.regularizers.l1_l2(l1=0.005, l2=0.001),
          bias_regularizer=tf.keras.regularizers.l2(0.001)),
    Dense(1, activation='sigmoid')
])
```

In the constructor of a dense or convolutional layer, there is also an optional bias regularizer argument.

```
model = Sequential([
    Dense(64, activation='relu',
          kernel_regularizer=tf.keras.regularizers.l1_l2(l1=0.005, l2=0.001),
          bias_regularizer=tf.keras.regularizers.l2(0.001)),
    Dense(1, activation='sigmoid')
])
```

We can pass this argument a regularizer object in just the same way as we did before. Again, this will add a penalty term to the loss function, this time for the bias parameters. The final thing I'd like to show you is how to include Dropout in your network.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

model = Sequential([
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])
```

Dropouts also has a regularizing effect in neural networks and intensive flow, we can include Dropouts as just another layer.

So here, you can see we're importing not only the dense layer as before, but also the Dropouts layer.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
```

Now in our sequential model, we've included the Dropouts layer in our list of layers.

```
model = Sequential([
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])
```

The Dropouts layer has a required arguments, which is the Dropouts rate. Here the rate has been set to 0.5 and that means that each weight connection between these two dense layers is set to zero with probability 0.5. This is sometimes referred to as Bernoulli Dropout, since the weights are effectively being multiplied by a Bernoulli random variable. Each of the weights are randomly dropped out independently from one another and Dropout has also applied independently across each element in the batch at training time. Again, we can compile and fit our model as normal, with Dropouts layers included in our model.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

model = Sequential([
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

model.compile(optimizer='adadelta', loss='binary_crossentropy', metrics=['acc'])

model.fit(inputs, targets, validation_split=0.25)
model.evaluate(val_inputs, val_targets)
model.predict(test_inputs)
```

Remember that when we're using Dropouts, we typically have two different modes for how we run the network. At training time, we randomly Dropout weights in the way that I've described and we can think of this as Training mode. However, when we're evaluating our model or making predictions from it, we stopped randomly dropping out the weights and we can think of this as testing mode.

```
model.fit(inputs, targets, validation_split=0.25)  # Training mode, with dropout
model.evaluate(val_inputs, val_targets)            # Testing mode, no dropout
model.predict(test_inputs)                         # Testing mode, no dropout
```

These two modes are automatically handled behind the scenes by the model.fit, model of evaluate, and model.predict methods. But it's worth bearing in mind that this is happening in the Dropout layer and later on we'll look at ways of getting more lower level control over the network and you'll see then how these two modes of operation for Dropouts can be controlled.

We're now starting to increase the sophistication of our models by including important concepts such as validation and regularization into our training workflow. You can also now include Dropout as an additional layer, and later on in the week, you'll see how to include batch normalization also as an additional layer type. In the following coding tutorial, you'll implement regularization into your model and see for yourself firsthand how these techniques help your model to generalize beyond the training data.

4.Introduction to callbacks

Welcome back. You now know how to include a validation set in your training in order to monitor the performance of your model on data that isn't used for training. You've also seen how to include regularization into your model to help avoid over fitting and improve the performance of the model on the validation set. You've implemented extra weight to regularization penalty terms that are added to the loss-function and added dropout into the network. But even now we're monitoring the performance of our model on the validation set, we're still having to fix the number of epochs that the model is training for ahead of time. What we would like to do is to have the ability to not only monitor the performance of the network, but also perform certain actions depending on those performance measures.

That's where callbacks come in. Callbacks are an important type of object TensorFlow and Keras that are designed to be able to monitor the loss in metrics at certain points in the training run and perform some action that might depend on those loss in metric values. Let's take a look at how callbacks are constructed. In TensorFlow, all callbacks are stored in the tensorflow.keras.callbacks module. Inside that module, there's a base class called callback which all other callbacks inherit from. You can also subclass the callback based class yourself to create your own callbacks, and that's what we're going to start to look at first as this is a really nice way to get a feel for how these objects work. Here I'm creating a new class called 'my_callback', and you can see that I'm sub-classing the callback base class.

```
from tensorflow.keras.callbacks import Callback

class my_callback(Callback):

    def on_train_begin(self, logs=None):
        # Do something at the start of training
```

This base class has a series of methods that can be overridden in our new class definition. These methods will then be called at different points in the training. For example, one of the methods of the callback base class is called on_train_begin.

```
def on_train_begin(self, logs=None):
```

This method optionally uses the logs keyword argument, and you'll find out more about that in a notebook reading coming up in this week. For now though, you just need to know that as the name suggests, this method will be called once at the start of training. So we can override this method in our new callback to perform some action at the beginning of the training run. Similarly, there are more methods of the base class that are called at other points in the training.

```
def on_train_begin(self, logs=None):
    # Do something at the start of training

def on_train_batch_begin(self, batch, logs=None):
    # Do something at the start of every batch iteration

def on_epoch_end(self, epoch, logs=None):
    # Do something at the end of every epoch
```

You can pretty much just read it from the name on_train_batch_begin is called at the start of every batch or every iteration of the training.

```
def on_train_begin(self, logs=None):
    # Do something at the start of training

def on_train_batch_begin(self, batch, logs=None):
    # Do something at the start of every batch iteration

def on_epoch_end(self, epoch, logs=None):
    # Do something at the end of every epoch
```

You can see that this method has a batch argument. So when this method is called, the batch number will be passed in using this argument. So if you want you can use the batch number during training to perform some action within this method. You'll see an example of this in the next coding tutorial. The train part of on_train_begin and on_train_batch_begin means that these methods will only be used in the training run.

```
def on_train_begin(self, logs=None):
    # Do something at the start of training

def on_train_batch_begin(self, batch, logs=None):
    # Do something at the start of every batch iteration
```

We can actually also use callbacks in evaluation and prediction runs as well. Again you'll see examples of this in the following coding tutorial. On_epoch_end is the method that's called at the end of every epoch.

```
def on_epoch_end(self, epoch, logs=None):
    # Do something at the end of every epoch
```

This method has an epoch arguments.

```
def on_epoch_end(self, epoch, logs=None):
    # Do something at the end of every epoch
```

So similar to before, when this method is called, it gets past the epoch number, so we could use this number within the method if we want to. Once we've defined the actions we want our callback to take at these various parts of the training run, all we need to do is to pass in the callback to model.fit.

```
model.fit(X_train, y_train, epochs=5, callbacks=[my_callback()])
```

Model.fit has a callbacks keyword argument, and this takes a list of callback objects. So here we're passing in an instance of our callback.

```
model.fit(X_train, y_train, epochs=5, callbacks=[my_callback()])
```

Remember again that model.fit returns a history object.

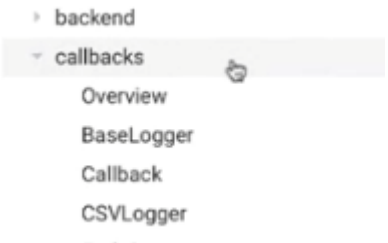
```
history = model.fit(X_train, y_train, epochs=5, callbacks=[my_callback()])
```

So by this point, it should make sense that this object is actually an example of a callback. It's a callback that is automatically included into every training run whenever we call model.fit. The action that this callback takes is simply to record the loss and metric values and store them as a dictionary in its history attribute.

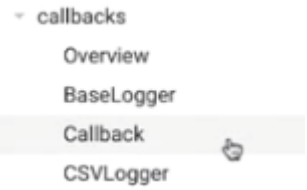
There are quite a few built-in callbacks in TensorFlow and we are going to be covering several of them in the later weeks of the course, but you can take a look at the list of built-in callbacks, if you go to the API documentation on TensorFlow.org.

So here's the tf.keras module,

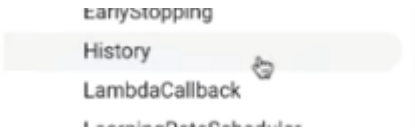
and inside there is the callbacks module and here's the list of built-in callbacks.



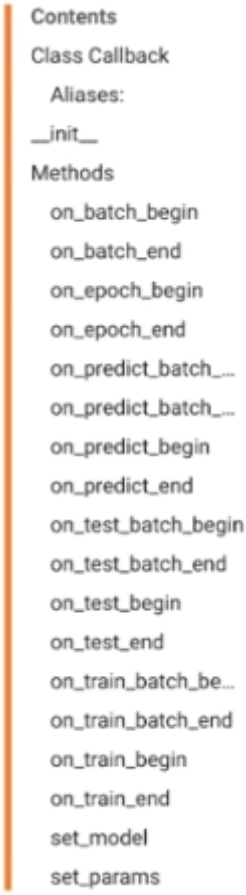
A couple of these names you'll recognize already.



Here's the base callback class that we can subclass to create our own callbacks, and here's the history callback that I was just talking about that's automatically added into every training run.



If we just take a look at the callback class, on the right-hand side here you can see the list of methods that you can override when you create your own callback that are called at different parts of a training, evaluation or prediction run.



The names are quite self-explanatory, so down the bottom here you can see four methods with the word train in the name and these methods will be called when you run model.fit. These four with test in the name will be called by model.evaluate and these four will predict and the name would be called by model.predict.

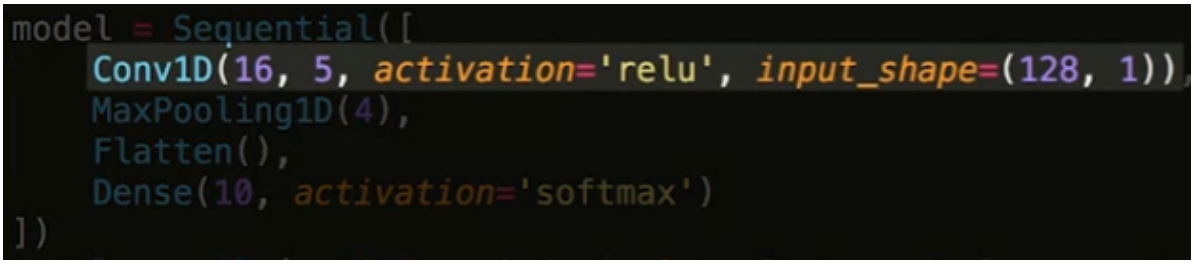
So now you know the purpose of callbacks and have a rough idea of how they're used. In the following coding tutorial, you'll consolidate this by putting it into practice and creating custom callbacks to perform certain actions. For now these actions will be simple and just serve to demonstrate how to use callbacks, but we'll soon be using callbacks for several important aspects of our model chain.

5.Early stopping and patience

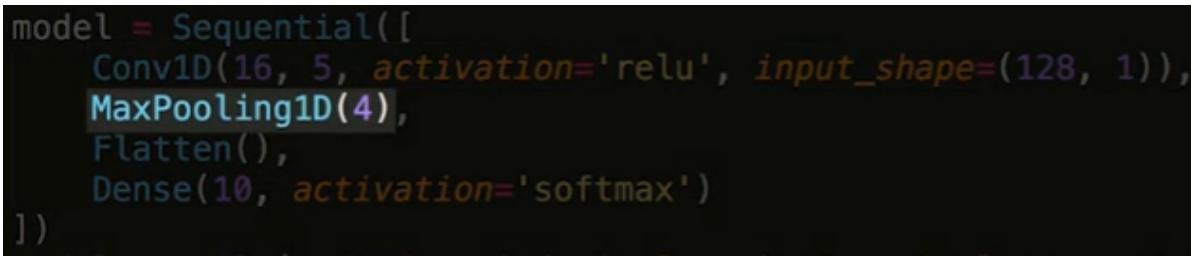
We've seen how to include validation sets and regularization techniques into our model training. We took a first look at callbacks. In this video, we'll use a callback to implement another regularization approach called early stopping. Early stopping is a technique that monitors the performance of the network for every epoch on a held out validation set during the training run, and terminates the training conditional on the validation performance. The Keras module contains a built-in callback designed for this purpose called the early stopping cutback. Let's see how it works.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, Flatten, Dense, MaxPooling1D
from tensorflow.keras.callbacks import EarlyStopping

model = Sequential([
    Conv1D(16, 5, activation='relu', input_shape=(128, 1)),
    MaxPooling1D(4),
    Flatten(),
    Dense(10, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

early_stopping = EarlyStopping()

model.fit(X_train, y_train, validation_split=0.2, epochs=100,
          callbacks=[early_stopping])
```

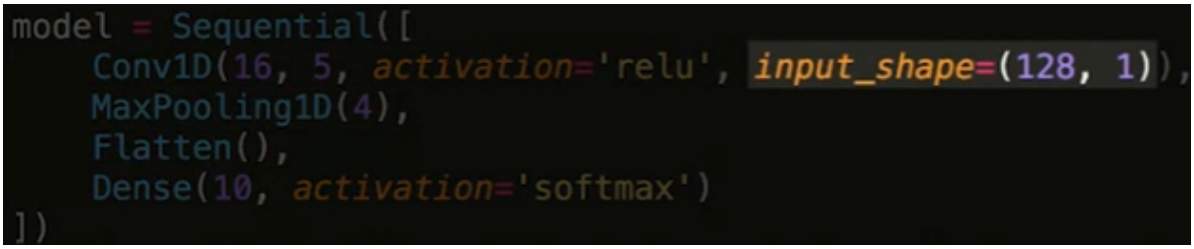
In this example, I've built a convolutional network starting with a Conv1D layer,



followed by a MaxPoolinglayer.



Note that the input is two-dimensional,



so this could be a univariate time series of length 128, for example. Then we've flatten the input by unrolling into one long vector,

```
Flatten(),
Dense(10, activation='softmax')
```

and finally, we have a dense layer with a 10 layer softmax.

```
Dense(10, activation='softmax')
```

We're compiling the model, with the AdamOptimizer, categorical crossentropy loss, and we're tracking the categorical accuracy metric.

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

So here's where the callback comes in. At the top, you can see that I'm importing the early stopping callback from the tensorflow.keras.callbacks module.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, Flatten, Dense, MaxPooling1D
from tensorflow.keras.callbacks import EarlyStopping
```

In this line, I'm creating an early stopping callback object.

```
13  early_stopping = EarlyStopping()
```

This callback has a few options that we'll look at in a moment. But for this instance, we're using the defaults, then we just need to pass in the callback to the model.fit method. Here, you'll see we're passing this callback in a list to the callbacks keyword argument.

```
15  model.fit(X_train, y_train, validation_split=0.2, epochs=100,
16            callbacks=[early_stopping])
```

It's a list because in practice you might be passing in a number of callbacks, all of which are performing different tasks during the training run. So what the early stopping callback is doing, is monitoring the performance of the network on the validation set, which you'll see we've created here with the validation_split keyword argument.

```
model.fit(X_train, y_train, validation_split=0.2, epochs=100,
          callbacks=[early_stopping])
```

It stops the training depending on how that performance progresses. So a natural question is, which measure of performance is the callback monitoring? The early stopping callback constructor takes a keyword argument called monitor, which can be used to set which performance metric to use. You can see here that I've passed in val_loss for this keyword argument, to say that I want to use the validation loss as the performance measure to decide when to terminate the training.

```
early_stopping = EarlyStopping(monitor='val_loss')
```

This is actually the default setting for the early stopping callback. In this model, the loss is the categorical_crossentropy loss.

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

As an example, you can instead choose to use the validation accuracy as the performance measure to decide when to terminate the training.

```
early_stopping = EarlyStopping(monitor='val_accuracy')
```

Remember that we said to track the accuracy metric when we compile the model. If we had more metrics that we'd use when we compile the model, then we could have used one of them instead. By the way, the actual string name that you see being passed into the monitor arguments here is the same as the string that's used as one of the keys in the history object that's returned from model.fit. So that's one way that you can check what string you should be using here. Another keyword arguments that we can sets in the early stopping callback, is the patience argument.

```
early_stopping = EarlyStopping(monitor='val_accuracy', patience=5)
```

By default, patience is set to zero. That means that as soon as the performance measure gets worse from one epoch to the next, then the training is terminated. Now, this might not be ideal since of course, the model's performance is noisy, and it might go up or down from one epoch to the next. What we really care about is that the general trend should be improving. That's why will often set the patience to some number of epochs, like here, I've set it to five epochs.

```
early_stopping = EarlyStopping(monitor='val_accuracy', patience=5)
```

In this case, the training will terminate only if there is no improvement in the monitor performance measure, the five epochs, in a row. The early stopping callback also has a min_delta argument.

```
early_stopping = EarlyStopping(monitor='val_accuracy', patience=5, min_delta=0.01)
```

This option can be used to define what qualifies as an improvement in the monitor performance measure. For example, if I set here the min_delta to be equal to 0.01,

```
early_stopping = EarlyStopping(monitor='val_accuracy', patience=5, min_delta=0.01)
```

that means that the validation accuracy has to improve by at least 0.01 for it to count as an improvement. If the validation accuracy improved by a smaller amount, say by 0.001, then that would be treated the same as a worsening of the performance by the early stopping callback and the patience counter would increase by one. By default, min_delta is zero, which means that any improvement in the performance is enough to reset the patience. The last thing I want to show you is one more option that we can use in the early stopping callback, and that is the mode keyword argument.

```
early_stopping = EarlyStopping(monitor='val_accuracy', patience=5, min_delta=0.01, mode='max')
```

You might have noticed that depending on the choice of quantity that we're monitoring, an improvement could be either an increase or a decrease in that quantity. If we're monitoring the validation loss, then we want the loss to go down. But if we're monitoring the validation accuracy, then we'd want that to go up. How does the early stopping callback know which direction is better? Well, by default, the mode keyword arguments is set to auto, which means that the direction is automatically inferred by the quantity name. If you're tracking the loss or the accuracy, then the callback knows what this is and whether increasing or decreasing is better. However, you can explicitly set the direction in the early stopping callback. For example, here I'm setting the mode to be max.

```
mode='max')
```

Which means that we're aiming to maximize the monitored performance measure, which is the validation accuracy.

In this lecture video, we've looked at an example of how callbacks are really useful objects to incorporate into our model training. There's more that you can do with callbacks, of course, and you'll see more examples of that in this week and later on in the course. For now though, you ready to go through the final coding tutorial of the week and implement the early stopping into your training.

week4

1.Welcome to week4-Saving and loading models

Welcome to this week of the course. So far in this course, we've covered some pretty fundamental aspects of developing deep learning models from building, training and testing models using the sequential class to things like including validation sets and regularization.

In this week we'll round off our basic workflow by covering saving and loading our models. There are actually a few ways of saving and loading models TensorFllow such as only saving the parameters or the weights of a model or saving the entire model including the architecture. You can also save the architecture only without the weights. As well as that you'll see how you can include model saving as part of the training run which of course is really useful in case the training gets interrupted for some reason and you want to restart it but you can also do some more sophisticated things like saving a model depending on certain performance measures.

As you might expect from what you saw previously we can use callbacks to do this for us and there's a built-in call back TensorFlow just for this purpose and so we'll spend some time looking at that in detail. Finally as well as saving and loading your own models, you'll see that there are many pre-trained TensorFlow models that can be downloaded and used in your applications. You'll see where you can find these models and how to use them later on in this week. In the programming assignment this time, you'll practice these different ways of saving and loading models as part of a training run for an image classification task on satellite image data. So I'll see you in the next lecture video.

2.Saving and loading model weights

Welcome back. In this lecture video, we're going to take a look at saving TensorFlow model parameters, unloading them from previously saved files. We'll see how to save model weights automatically during a training run, and for this we'll use a built-in callback called the model checkpoint callback. It's also possible to manually save the weights of a model, for example, after a training run has completed. So you also see how to do this. An extra detail is that there are two different formats available for saving and loading files due to there being a native TensorFlow format as well as the HDF5 format used by Keras. For the moment, we're just going to be saving model weights only. We'll cover saving the weights and architecture later on in the week. So here's an example of how to save a model during a training run.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.callbacks import ModelCheckpoint

model = Sequential([
    Dense(64, activation='sigmoid', input_shape=(10,)),
    Dense(1)
])
model.compile(optimizer='sgd', loss=BinaryCrossentropy(from_logits=True))

checkpoint = ModelCheckpoint('my_model', save_weights_only=True)

model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint])
```

As I mentioned, we're using a built-in callback here called the model checkpoint callback.

```
from tensorflow.keras.callbacks import ModelCheckpoint

model = Sequential([
    Dense(64, activation='sigmoid', input_shape=(10,)),
    Dense(1)
])
model.compile(optimizer='sgd', loss=BinaryCrossentropy(from_logit

checkpoint = ModelCheckpoint('my_model', save_weights_only=True)
```

These first few lines should be pretty familiar by now.

```
model = Sequential([
    Dense(64, activation='sigmoid', input_shape=(10,)),
    Dense(1)
])
model.compile(optimizer='sgd', loss=BinaryCrossentropy(from_logits=True))
```

You can see that I'm building a simple feed forward network using two dense layers. The output is a single neuron with a linear activation.

```
model = Sequential([
    Dense(64, activation='sigmoid', input_shape=(10,)),
    Dense(1)
])
```

Here's a reminder of something you learned previously.

```
model.compile(optimizer='sgd', loss=BinaryCrossentropy(from_logits=True))
```

I'm compiling the model using the stochastic gradient descent optimizer, and for the loss-function, I've instantiated at BinaryCrossentropy object and fed in the argument from_ logits=True. The binary cross-entropy class is imported up here from the tensorflow.keras.losses module.

```
from tensorflow.keras.losses import BinaryCrossentropy
```

So this network is actually a binary classifier, and the linear unit output will be squeezed through a sigmoid activation function that is being handled by the loss-function itself. We're importing the model checkpoint callback from the tensorflow.keras.callbacks module,

```
from tensorflow.keras.callbacks import ModelCheckpoint
```

and down here, we're creating a model check point object, which I'm calling checkpoint.

```
checkpoint = ModelCheckpoint('my_model', save_weights_only=True)
```

The constructor of this callback has one required argument, which is the file path.

```
checkpoint = ModelCheckpoint('my_model', save_weights_only=True)
```

This is the name that callback will use to save the model. You can see we're also using another keyword argument here.

I've set save_weights_only to be true.

```
checkpoint = ModelCheckpoint('my_model', save_weights_only=True)
```

As you might have guessed, this means that only the model weights will be saved by this callback and not the architecture.

The final line should again look familiar.

```
model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint])
```

We're calling model.fit, parsing in the training inputs and outputs, setting it to train for 10 epoch. Just as we did previously, here we're using the callbacks keyword argument and parsing in a list, which contains our model checkpoint callback object.

```
model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint])
```

This callback will save the model weights after every epoch.

```
# checkpoint
# my_model.data-00000-of-00001
# my_model.index
```

Now, because we're using the same file name to save them model, the saved weights will get overwritten every epoch. We'll see how we can change that later on in the week. But what you'll see after the weights have been saved is that there are three files created in the current working directory. These are the files that contain the model parameters. You can see here that the first part of these filenames is my model,

```
# checkpoint
# my_model.data-00000-of-00001
# my_model.index
```

which is the file name we specified in the callback constructor.

```
checkpoint = ModelCheckpoint('my_model', save_weights_only=True)
```

There's also an alternative formats that you can use to save the model weights.

```
checkpoint = ModelCheckpoint('keras_model.h5', save_weights_only=True)
```

The formats used by Keras is the HDF5 format. These file formats often use the extension.h5.

```
checkpoint = ModelCheckpoint('keras_model.h5', save_weights_only=True)
```

If you give the file path argument, the h5 extension, like I've done here, then the model will save the weights in the Keras HDF5 format.

```
# keras_model.h5
```

What you'll see after running the training is that the model weights will have saved to a single HDF5 file in the current working directory. For most models, it doesn't really make a difference which format you use but in general, I'd recommend using the native TensorFlow format. So we've saved the model weights during a training run, how do we load weights that have been previously saved?

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import ModelCheckpoint

model = Sequential([
    Dense(64, activation='sigmoid', input_shape=(10,)),
    Dense(1)
])

model.load_weights('my_model')
```

So remember that we've only saved the weights and not the model architecture. That means to use the saved weights, we still have to have the code to rebuild the model. So here is the same code that we used before to define this model, and let's say we've just built this model again from scratch.

```
model = Sequential([
    Dense(64, activation='sigmoid', input_shape=(10,)),
    Dense(1)
])
```

So the weights have been randomly initialized. We can then load the weights we've saved from a previous training run by calling model.load_weights and parsing in the same file name that we used to save the weights.

```
model.load_weights('my_model')
```

If we had saved the model weights in the HDF5 format, it's exactly the same, just parsing the file name together with the h5 extension.

```
model.load_weights('keras_model.h5')
```

The final thing I'd like to show you is how to save the model weights manually, that is, without using the model checkpoint callback.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping

model = Sequential([
    Dense(64, activation='sigmoid', input_shape=(10,)),
    Dense(1)
])
model.compile(optimizer='sgd', loss='mse', metrics=['mae'])

early_stopping = EarlyStopping(monitor='val_mae', patience=2)

model.fit(X_train, y_train, validation_split=0.2, epochs=50,
          callbacks=[early_stopping])

model.save_weights('my_model')
```

So for example, let's say we've built and compiled our model like we can see here, and maybe we've used an early stopping callback, which you already know about from earlier.

```
model = Sequential([
    Dense(64, activation='sigmoid', input_shape=(10,)),
    Dense(1)
])
model.compile(optimizer='sgd', loss='mse', metrics=['mae'])
```

You can see in this line, I've created there a stopping callback object and just as a reminder, notice that it's monitoring the mean absolute error metric on the validation set and the patience is set two.

```
early_stopping = EarlyStopping(monitor='val_mae', patience=2)
```

That means that the training will be terminated if the validation mean absolute error performance doesn't improve after two successive epochs. In this line, we're launching the training.

```
model.fit(X_train, y_train, validation_split=0.2, epochs=50,
          callbacks=[early_stopping])
```

Again, as a reminder, you can see that I've parsed in the training inputs and targets and created a validation set by holding back 20 percent of the data, and the maximum number of epochs that this model will train for is 50. Here, I'm parsing in the early_stopping callback.

```
model.fit(X_train, y_train, validation_split=0.2, epochs=50,
          callbacks=[early_stopping])
```

So depending on how the training progresses, the model might reach an optimal performance on the validation metric and the training is terminated. We can then manually save the model weights like this just by calling model.save_weights and again parsing in the file path that should be used to save the weights.

```
model.save_weights('my_model')
```

So now you've seen how easy it is to save the parameters of your model, either using a callback during the training run or manually by using the model.save_weights method. You can also load the weights that you've previously saved, assuming that you have the code to build the same model architecture that the weights were saved from.

In the next lecture video, we'll take a look at some of the options that are available when saving models during a training run using the model checkpoint callback. But first, it's time to code some examples yourself. In this week, Adrian is going to be walking you through the coding tutorial notebook. So have fun with that and I'll see you in the next lecture video.

3.Model saving criteria

So now you've seen how to use the ModelCheckpoint callback to save your model every epoch during a training run. But you might want to have more flexibility when it comes to saving your model. For example, you might want to save your model with a different frequency or according to some criteria. The ModelCheckpoint callback comes with a few extra options that help you to do this. So let's take a look at what they are.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import ModelCheckpoint

model = Sequential([
    Dense(16, activation='relu'),
    Dropout(0.3),
    Dense(3, activation='softmax')
])
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy',
              metrics=['acc', 'mae'])

checkpoint = ModelCheckpoint('training_run_1/my_model', save_weights_only=True)

model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=10,
          batch_size=16, callbacks=[checkpoint])
```

So at this time we have a model with two dense layers and a dropout layer with dropout rate of 0.3.

```
model = Sequential([
    Dense(16, activation='relu'),
    Dropout(0.3),
    Dense(3, activation='softmax')
])
```

The output layer is a three-way softmax. So this looks like a classification model with three classes. We are compiling the model with the rmsprop optimizer and we can see from the loss function that the target data is in sparse form, or in other words, each target values an integer equal to zero, one or two.

```
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy',
              metrics=['acc', 'mae'])
```

We're tracking two metrics: the categorical accuracy and the mean absolute error, and here's our ModelCheckpoint callback.

```
checkpoint = ModelCheckpoint('training_run_1/my_model', save_weights_only=True)
```

This time the file path points to the folder training_run_1, which will be created if it doesn't exist and inside that, the name used to save the model is my_model.

```
checkpoint = ModelCheckpoint('training_run_1/my_model', save_weights_only=True)
```

Again, we're saving weights only.

```
save_weights_only=True)
```

The final line here is just training the model for 10 epochs, using a batch size of 16 and passing in the checkpoint callback.

```
model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=10,
          batch_size=16, callbacks=[checkpoint])
```

As we've seen, the ModelCheckpoint callback will save the weights at the end of every epoch by default.

```
checkpoint = ModelCheckpoint('training_run_1/my_model', save_weights_only=True,
                             save_freq='epoch')
```

Here's a new keyword arguments that can be used when you create the callback.

```
checkpoint = ModelCheckpoint('training_run_1/my_model', save_weights_only=True,
                             save_freq='epoch')
```

These arguments sets the save_frequency and the default value is the string epoch, which is shown here. But you can instead set this argument to an integer value and this sets the frequency of saving in terms of the number of samples that have been seen by the model since the last time the weights were saved.

```
checkpoint = ModelCheckpoint('training_run_1/my_model', save_weights_only=True,
                             save_freq=1000)
```

So just to emphasize that point, the save_frequency argument specifies the number of samples and not the number of training iterations. So here my_model would save the weights every 1,000 samples and I have a batch size of 16 and that means that my_model would save every 62 or 63 iterations.

```
checkpoint = ModelCheckpoint('training_run_1/my_model', save_weights_only=True,
                             save_freq=1000)

model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=10,
          batch_size=16, callbacks=[checkpoint])
```

Here's another option. You can pass into the callback constructor.

```
checkpoint = ModelCheckpoint('training_run_1/my_model', save_weights_only=True,
                             save_best_only=True, monitor='val_loss')
```

The keyword argument save_best_only defaults to false. But if you set it to be true, as I've done here, then the model will only save the weights according to a performance measure criteria. That criterion is set by their monitor keyword argument.

```
monitor='val_loss')
```

The default value of this argument is val_loss, which I've shown here. In this case, the callback only saves the weights if the validation loss is the best value seen so far in the training run. This looks quite similar to what we saw before with the early stopping callback. The callback behaves in a certain way according to how the training is progressing. Just in the same way as we've seen with the other stopping callback, you can choose which performance measure the callback is monitoring. Here I've changed the monitor performance measure to be the validation accuracy,

```
checkpoint = ModelCheckpoint('training_run_1/my_model', save_weights_only=True,
                             save_best_only=True, monitor='val_acc')
```

which is one of the metrics I added when I compiled the model. This should also look familiar. A related keyword arguments is mode and again, as before,

```
checkpoint = ModelCheckpoint('training_run_1/my_model', save_weights_only=True,
                             save_best_only=True, monitor='val_acc', mode='max')
```

these arguments tells the callback whether we're trying to maximize or minimize the monitor to performance measure. Here we're monitoring the validation accuracy, which we wants to maximize. So mode is set to max. The default for mode is auto, which will automatically infer from the name of the performance measure which direction we want it to go in. One more thing I'd like to show you is that the filename itself can be formatted with whatever keys are available in the logs dictionary.

```
checkpoint = ModelCheckpoint('training_run_1/my_model.{epoch}.{batch}',
                             save_weights_only=True, save_freq=1000)
```

So for example, if we're saving the model every 1,000 samples,

```
save_freq=1000)
```

we could make the filename include the epoch and batch number.

```
('training_run_1/my_model.{epoch}.{batch}',
```

If we do this, then no saved files would be overwritten as they would all get a unique filename. Here's another example of that.

```
checkpoint = ModelCheckpoint('training_run_1/my_model.{epoch}-{val_loss:.4f}',
                             save_weights_only=True)
```

In this case, we're saving the model every epoch, so we're including the epoch number in the filename and we could also add the validation loss into the filename as well. So you can now save your network model during the training run and with more flexible options. Just as we saw with model validation, callbacks are again really useful objects to dynamically perform actions according to certain criteria. Here, the ModelCheckpoint callback can be set to only save your model if its performance during the training has improved.

In the next coding tutorial, you'll put this into practice to see how this works for yourself. After that, we'll be taking a look at saving and loading the entire model, including the architecture as well as the weights. So I'll see you then.

4.Saving the entire model

Hi, and welcome back. So far in this week, you've seen how we can include model saving into our training pipeline using
the ModelCheckpoint callback. And how you can also manually save the weights of a model using the model.save_weights method. All of this supposed that you would still have the code to build the model when it came to reloading the weights. Of course, you might also want to save the entire model, weights and architecture combined. That's what we'll talk about in this video. Here's the same model example we looked at last time. Everything looks the same.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import ModelCheckpoint

model = Sequential([
    Dense(16, activation='elu'),
    Dropout(0.3),
    Dense(3, activation='softmax')
])
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy',
              metrics=['acc', 'mae'])

checkpoint = ModelCheckpoint('my_model', save_weights_only=False)

model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint])
```

We're again using the ModelCheckpoint callback, and we're passing it in to the model.fit method. This callback will save the model at the end of every epoch. The only difference is in the argument we're passing into the callback constructor. Now you can see we're setting save_weights_only to be false.

```
checkpoint = ModelCheckpoint('my_model', save_weights_only=False)
```

So the callback will save the entire model at the end of every epoch and not just the weights. In fact, this is the default value for this keyword argument, so we could have just left it out altogether. Whenever we save the entire model like this, the files that get saved will change.

```
17  # my_model/assets/
18  # my_model/saved_model.pb
19  # my_model/variables/variables.data-00000-of-00001
20  # my_model/variables/variables.index
```

Remember before, the callback used the file path my_model in the name of the files that were saved in the current working directory. Now, because we're saving the entire model and not just the weights, the file path my_model is used to create a subdirectory within the current working directory. Inside that, you'll find two more subdirectories, assets and variables. The assets folder is where files can be stored that are used by the TensorFlow graph.

```
my_model/assets/
```

In our simple example, this folder would be empty. The variables folder contains the saved weights of the model.

```
# my_model/variables/variables.data-00000-of-00001
# my_model/variables/variables.index
```

These file names should look familiar from when we were saving weights only. The last file here, saved_model.pb, is the file that stores the TensorFlow graph itself.

```
# my_model/saved_model.pb
```

You can think of this as storing the model architecture. So it contains all the information from when we built and compiled the model, including the optimizer state, in case we want to resume training from a saved model. Again, we can also save the entire model in the Keras HDF5 format. It looks just the same. We're again setting save_weights_only = false.

```
checkpoint = ModelCheckpoint('keras_model.h5', save_weights_only=False)
```

The only difference is we're including the h5 extension, as we did before.

```
('keras_model.h5',
```

In this case, just one file will be saved, which here is the keras_model.h5 file.

```
keras_model.h5
```

This looks the same as when we saved weights only, but this HDF5 file now contains the architecture as well as the weights. We could also manually save the entire model, just as we manually saved the weights of a model after a training run.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout


model = Sequential([
    Dense(16, activation='elu'),
    Dropout(0.3),
    Dense(3, activation='softmax')
])
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy',
              metrics=['acc', 'mae'])

model.fit(X_train, y_train, epochs=10)

model.save('my_model')  # SavedModel format
```

If you remember, when we saved the weights, we used model.save_weights. The only difference here is
that we're using model.save.

```
model.save('my_model')  # SavedModel format
```

This method will now save the entire model, architecture and weights, in the same directory structure as before for
the native TensorFlow saved model format. And again, we can manually save the entire model, this time in HDF5 format, just by adding the h5 extension to the file path.

```
model.save('keras_model.h5')  # HDF5 format
```

So what about loading an entire model? Well, this time, we don't need the original code that was used to build the model, so
the loading code looks a lot simpler.

```
from tensorflow.keras.models import load_model

new_model = load_model('my_model')
```

In fact, this is all there is to it. Here, I'm importing the load_model function from the tensorflow.keras.models module.

```
from tensorflow.keras.models import load_model
```

And to load the model, I just pass this function, the path to the model folder, and it will return the complete model instance, architecture and saved weights.

```
new_model = load_model('my_model')
```

We could now do everything that we've done before with this loaded model instance.

```
from tensorflow.keras.models import load_model

new_model = load_model('my_model')

new_model.summary()
new_model.fit(X_train, y_train, validation_data=(X_val, y_val),
              epochs=20, batch_size=16)
new_model.evaluate(X_test, y_test)
new_model.predict(X_samples)
```

We could print out the model summary. We could resume training, using the fit method. We could run new_model.evaluate to evaluate the loss in metrics on a held-out test set. Or get model predictions from unseen examples using new_model.predict. Finally, the code for loading an entire model from the Keras HDF5 format also looks the same.

```
from tensorflow.keras.models import load_model

new_keras_model = load_model('keras_model.h5')
```

We use the same function, load_model, and this time we just pass in the file path, including the h5 extension. Now we have a pretty comprehensive set of tools for saving models in different ways and according to different criteria, as well as loading models. It's also possible to only save the model architecture without the weights, and you'll see how to do that in a reading notebook later in this lesson. In the next coding tutorial, you'll practice saving and loading entire models using the methods we've just looked at.

5.Loading pre-trained Keras models

In this week, you've now seen how we can easily save and load models in TensorFlow, either as part of the training loop or manually. But there are also many pre-trained models available out there that you can download and experiment with in your applications. In this lecture video, we'll take a look at one resource for loading pre-trained image classification models which is through the Keras API. If you go to keras.io/applications, you'll see a number of different architectures that are available for you to download. These are all image classification models that have been trained on the ImageNet dataset. There are also examples
of how you can use these models for different purposes, including a straightforward classifier models or as image feature extractors. Later on in the course, you'll see an example of how to use one of these pre-trained models as a feature extractor, as well as implementing a transfer learning application. Documentation is also available for individual model architectures, so you can see how to use them and what the options are for downloading them.

Let's take a look at an example.

```
from tensorflow.keras.applications.resnet50 import ResNet50

model = ResNet50(weights='imagenet')
```

One of the models available through the Keras API is the ResNet-50 model.

```
from tensorflow.keras.applications.resnet50 import ResNet50
```

To load this model, you import the ResNet-50 class from the tensorflow.keras.applications.resnet50 module. Here I'm creating a ResNet-50 object, and when this line of code executes, the model architecture in weights will be downloaded to your local machine if it hasn't already been downloaded previously.

```
model = ResNet50(weights='imagenet')
```

When you do this yourself, it might take a few minutes to load in as these models can be pretty big. Just so you know, the model is downloaded into a hidden folder in your home directory inside a subdirectory called models. You can also see I've passed in the keyword arguments, weights equals ImageNet.

```
(weights='imagenet')
```

That means that the model will be loaded with weights that have been learned from training on the ImageNet dataset. Otherwise, if you set this to none, then the weights will be randomly initialized for a fresh training. Another interesting
option you can use is the include top keyword argument.

```
model = ResNet50(weights='imagenet', include_top=False)
```

By default, this is equal to true in which case the complete classifier model is downloaded and instantiated. However, if you set include top to be false, the fully connected layer at the top of the network isn't loaded. So what you end up with is a headless model that you can use for things like transfer learning applications. Here's an example of how you can use a downloaded model to make class predictions on any image you might have available locally.

```
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input
import numpy as np

model = ResNet50(weights='imagenet', include_top=True)

img_input = image.load_img('my_picture.jpg', target_size=(224, 224))
img_input = image.img_to_array(img_input)
img_input = preprocess_input(img_input[np.newaxis, ...])
```

First of all, here I've instantiated the model using weights that have been learned from training on ImageNet,

```
model = ResNet50(weights='imagenet', include_top=True)
```

and I've set include top to be true, so I'm getting the complete model including the final classification layer. In the Keras module, there are some really useful tools for handling images and here I'm importing the image module from tensorflow.keras.preprocessing.

```
from tensorflow.keras.preprocessing import image
```

I'm using the load image function in this line to load in an image I've got stored in the current working directory called my_picture.jpg.

```
img_input = image.load_img('my_picture.jpg', target_size=(224, 224))
```

I'm also passing the target size keyword and setting that to 224 by 224.

```
target_size=(224, 224))
```

If you take a look at the documentation where we were, you'll see that this is the required input size for the ResNet-50 classifier model, and the load image function can handle any necessary resizing for us. The model it's self needs to take in a numpy array as an input. Again, the image module can take care of this force with the image to array function.

```
img_input = image.img_to_array(img_input)
```

The final thing we need to do to get the image ready for the model is to preprocess it according to the model requirements. You can see here I'm importing the preprocess input function from the tensorflow.keras.applications.resnet50 module,

```
from tensorflow.keras.applications.resnet50 import preprocess_input
```

and the last line of code down here uses that to do that last stage of preprocessing.

```
img_input = preprocess_input(img_input[np.newaxis, ...])
```

You'll notice that we're also adding an extra dummy dimension to the input.

```
(img_input[np.newaxis, ...])
```

This might look familiar from earlier on in the course. Remember that the model is expecting a batch of inputs where the first dimension is equal to the number of examples in the batch. We're just feeding into one example here but we still need to add the batch dimension for the model. Here is where we get the model prediction, and I'm sure this looks familiar.

```
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions
import numpy as np

model = ResNet50(weights='imagenet', include_top=True)

img_input = image.load_img('my_picture.jpg', target_size=(224, 224))
img_input = image.img_to_array(img_input)
img_input = preprocess_input(img_input[np.newaxis, ...])

preds = model.predict(img_input)
decoded_predictions = decode_predictions(preds, top=3)[0]

# List of (class, description, probability)
```

The image is preprocessed and ready, so we just need to pass it into the model.predict method to get the model predictions.

```
preds = model.predict(img_input)
```

These predictions will be a numpy array of probabilities, but of course what we want to know is what the classes are that the model is predicting for this image. So the final function that we're importing is the decode predictions function.

```
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions
```

Again, from the tensorflow.keras.applications.resnet50 module, this function will take the model predictions

```
decoded_predictions = decode_predictions(preds, top=3)[0]
```

and return a list of tuples corresponding to the class code, its description in plain English that we can understand, and the network probability. The top equals 3 arguments just means that we're only retrieving the top three model predictions.

```
# List of (class, description, probability)
```

Now you know how to load pre-trained models using the Keras API and use them to get classification predictions on your own images. There are many more models that can be downloaded and used in a similar way through the Keras API. So be sure to take a look again at the website we saw earlier to see all the options. In the next coding tutorial, you'll download and use pre-trained Keras models yourself. So have fun with that and in the next lecture video, I'll show you another source of pre-trained models we can use with TensorFlow Hub.