

CORE ML 3 SURVIVAL GUIDE

Core ML Survival Guide

More than you ever wanted to know about mlmodel files and the Core ML and Vision APIs

Matthijs Hollemans

This book is for sale at <http://leanpub.com/coreml-survival-guide>

This version was published on 2019-09-29



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2018 - 2019 M.I. Hollemans

Contents

Introduction	i
Who Is This Book For?	ii
Useful Links	iii
Part 1: The Core ML Ecosystem	1
What is Core ML — and What is It Not?	2
coremltools	6
Part 2: Converting Models	10
Image Preprocessing	11
Part 3: Examining Models	19
Viewing Models With Netron	20
The mlmodel File Format	21
Using the Spec to Edit Models	32
Part 4: Model Surgery	39
Saving the Weights as 16-bit Floats	40
Part 5: Inside the App	42
The Neural Engine	43
Transposing an MLMultiArray	45

CONTENTS

Part 6: Advanced Topics	49
Working With Video	50

Introduction

This guide is a collection of tips and tricks for working with Core ML and `mlmodel` files. I collected these during my work as machine learning consultant for iOS and macOS apps, which often involves making models suitable for use with Core ML.

At first sight, the Core ML API appears very simple to use. Xcode even automatically generates a class for you that makes using machine learning models as easy as:

```
let model = YourModel()  
let result = try? model.prediction(input: someInput)
```

Such a high-level API is really convenient for the developer when everything goes smooth and works as intended. Unfortunately, things don't always go smooth...

Core ML hides away a lot of the complexity involved in running machine learning models. But that complexity is still there and — like it or not — you're going to have to deal with it when the bits start hitting the fan.

I know for a fact that a lot of developers struggle with the finer points of Core ML because for the past couple of years I've been keeping a close eye on the Core ML topics on [StackOverflow](https://stackoverflow.com/questions/tagged/coreml)¹ and the [Apple Developer Forums](https://forums.developer.apple.com/community/system-frameworks/machine-learning)². I also receive a lot of questions on my GitHub repos and by email.

It's clear that using Core ML isn't always as easy as it's supposed to be...

That's why I wrote this guide. It contains pretty much everything I know about Core ML. As I learn new things, I'll keep updating this guide.

I primarily work with deep learning and computer vision, so most of the tips in this guide are about neural networks that take image inputs — but a lot of the advice applies to other model types and data types too. The code examples are in Python and Swift, but should be easy enough to translate to Objective-C.

Let me know if anything is still unclear, or if you have a question about Core ML that is not answered in this book. Happy reading!

¹<https://stackoverflow.com/questions/tagged/coreml>

²<https://forums.developer.apple.com/community/system-frameworks/machine-learning>

Who Is This Book For?

I wrote this book to help out developers who have questions about Core ML, or who want to learn how to get the best performing Core ML models in their apps.

Just to be clear: This book does *not* teach machine learning!

You're already supposed to know what a model is, the difference between training and inference, and so on. If you're completely new to machine learning, please check out my other book [Machine Learning by Tutorials](#)³.

This book also isn't a beginner's guide to Core ML. The intended audience is people who want to get more out of Core ML, or who are stuck trying to get their models to work.

If you haven't used Core ML before, I suggest you first follow one of the many [beginner tutorials](#)⁴ available online and then come back to this book if you run into any problems — or if you want to learn more, of course!

³<https://store.raywenderlich.com/products/machine-learning-by-tutorials>

⁴<https://www.raywenderlich.com/577-core-ml-and-vision-machine-learning-in-ios-11-tutorial>

Useful Links

You may find the following links useful in your Core ML journey.

Official Core ML documentation:

- Apple’s [coremltools documentation](#) and [source code](#).
- The [Core ML specification](#), although it can be a little out of date.
- The most up-to-date specification is in the [proto files](#), which you’ll look at in great detail in this book.

Source code:

- [The GitHub repo](#) with the accompanying source code files for this book.
- [CoreMLHelpers](#) makes it a little easier to work with Core ML in Swift.

Tutorials and blogs:

- A [curated list of machine learning resources](#), with a focus on Apple technologies.
- [Machine learning tutorials](#) at raywenderlich.com.
- [Heartbeat by fritz.ai](#), a great blog about machine learning on the edge.
- [My own blog](#), which covers Core ML but also Metal Performance Shaders and other topics related to machine learning in Swift.

Questions and answers:

- [Core ML questions on StackOverflow](#).
- [The machine learning section](#) of the Apple Developer Forums.

Part 1: The Core ML Ecosystem

What can you do with Core ML and where does it fit into the world of machine learning?

What is Core ML — and What is It Not?

Apple’s marketing department has done a good job of hyping up Core ML, but it’s important to understand exactly what Core ML is and also what it isn’t. The goal of this chapter is to set realistic expectations.

Core ML is really two things:

1. The **mlmodel** file format definition. This is an open file format for machine learning models.
2. The **CoreML.framework** that lets you perform inference in your iOS, tvOS, watchOS and macOS apps using the model from an mlmodel file.

Most Core ML tutorials focus on the API from the framework but to make the most of Core ML you also need to understand how the mlmodel format works. In this guide you’ll find a lot of tips for working with mlmodel files.

Core ML and the machine learning community

So where does Core ML fit into the larger picture?

The Core ML framework and the mlmodel format are on the fringes of the ML ecosystem. Literally, because Core ML does machine learning “on the edge” as opposed to “in the cloud” — but also because to the vast majority of machine learning practitioners, Core ML is not important at all.

Most of the effort in machine learning goes into designing and training the models, while Core ML is just one technology of many for deploying the trained models into production. And mlmodel is just another file format.

It’s good that Apple opened up the mlmodel specification. This has made it easier for other tools to adopt the format. However, mlmodel is *not* the industry standard for the interchange of models between platforms. It looks like [ONNX](http://onnx.ai)⁵, the Open Neural Network eXchange format, is going to win that race.

⁵<http://onnx.ai>

Perhaps things will change as machine learning on mobile becomes more popular, but right now Core ML isn't having much of an impact on the industry. One reason may be that Core ML, unlike pretty much all other machine learning tools, is not open source and therefore does not get to benefit from an active community of users and developers.

Don't get me wrong... Core ML is great for iOS and macOS developers who want to add machine learning to their apps, but it's just one technology among many competitors that all try to do the same thing. Even on Apple platforms, Core ML is not the only choice for adding machine learning to your apps.

Benefits of using Core ML

The following are good reasons for using Core ML in your app:

- The API is easy to use, especially with the tips from this guide. ;-)
- You only need to provide the `mlmodel` file. You don't need to implement any of the machine learning algorithms yourself.
- Core ML can run your model on the CPU, on the GPU, or on the Neural Engine, depending on what hardware is available in the user's device. This allows it to make optimal use of the available resources. Core ML can even split up the model to only perform certain parts on the GPU (tasks that need to do a lot of computation) and the other parts on the CPU.
- Because Core ML can run models on the CPU, you can develop and test your apps on the iOS Simulator. This is not the case for apps that directly use the Metal APIs, since the Simulator does not allow Metal apps. Nor do Metal apps play well with unit tests. Core ML doesn't have these problems. (Update: as of iOS 13, Metal apps can run on the Simulator too.)
- The biggest benefit: Core ML can use the Neural Engine on devices with the A12 or A12X chip. The Neural Engine gives a great performance boost and right now Core ML is the only way to get your hands on this extraordinary computing power.

Prior to the release of the iPhone XS, I advised my clients to prefer Metal Performance Shaders over Core ML in order to get the best possible speed, especially for models that need to process real-time video. I still recommend that for older devices, but having access to this new Neural Engine really changes things and makes Core ML much more attractive.

Downsides of Core ML

There's also some bad news — the biggest downsides of Core ML are:

- Creating an `mlmodel` file is not always as easy as it looks. You may need to wrestle with `coremltools` and Python code. (That's why I called this the Core ML “survival” guide.)
- Core ML supports only a limited number of neural network layer types. It is possible to create [custom layers](#) but that requires implementing the machine learning algorithms yourself, and you lose the ability to use the Neural Engine. (Update: Core ML 3 introduces many new layer types, including low-level mathematics operations that can be used to implement new kinds of layers directly inside the `mlmodel`.)
- The speed can be unpredictable. Sometimes a model will suddenly run a lot slower than before. It appears as if Core ML is trying to be clever in how it schedules things, but doesn't always succeed.
- You have little control. Core ML is a black box and so it's hard to understand why and how it does certain things.
- Troubleshooting is difficult. For example, you cannot easily look at the output produced by intermediate layers.
- The `mlmodel` files are unprotected. Even though your app doesn't ship with the original `mlmodel` file but with a [compiled version](#), it's still possible for competitors to download your app and reverse engineer your model.
- Core ML is not open source and is only updated with new OS releases. Therefore, Core ML will always lag behind the state-of-the-art.

Some of these limitations are more serious than others, but if you want to use Core ML in your app, you'll have to learn to work with them or around them. That's what this guide is for!

Alternatives to Core ML

For performing inference on the device you have a few options besides Core ML:

- **Metal Performance Shaders (MPS).** This is a toolkit for running neural networks with Metal, Apple's language for programming the GPU. Core ML actually uses MPS

internally too, but you can often get a speed improvement by implementing the neural network yourself directly with MPS — especially on devices that do not have a Neural Engine. MPS even supports training on the device. (Update: Core ML 3 also supports a limited type of on-device training.)

- **TensorFlow Lite.** While it's possible to convert some TensorFlow models to Core ML, only a limited number of operations are supported. If you have a TensorFlow graph that Core ML can't handle, you can use [TF-Lite](https://www.tensorflow.org/lite/)⁶ instead. Note that TF-Lite currently only runs on the CPU, requires you to write C++, and also supports fewer operations than the full version of TensorFlow.
- **Roll your own.** Using the Accelerate framework, Metal or OpenGL, and third-party libraries like Eigen. Generally, I recommend against doing this as it's a lot of work and usually not worth the effort. (I've done this a few times with TensorFlow models to avoid the overhead of the TensorFlow API.)

Even if you use Core ML, you may need to write additional pre- or post-processing code using the Accelerate framework. And if your model needs a custom layer, you'll have to write some Metal code. So even with Core ML you may still end up using these other APIs.

⁶<https://www.tensorflow.org/lite/>

coremltools

Chances are that your model was trained with a Python-based tool such as TensorFlow, Keras, PyTorch, or MXNet. Because Python is the language of choice for machine learning, Apple has provided the **coremltools** package for working with Core ML models from Python.

The primary reason for using coremltools is to convert your model to Core ML format, but it does more than just conversions alone. You can also use it to tweak existing models and even build models from scratch.

What does it do?

coremltools performs three main tasks:

1. Converting a trained model to Core ML format.
2. Using a Core ML model to make predictions from Python.
3. Changing properties of an existing Core ML model.

The coremltools package also contains the official specification of the Core ML file format. You'll look at this in detail in the [mlmodel file format](#) chapter.

The built-in converters

The model converters are part of the `coremltools.converters` module. There currently are converters for:

- Caffe
- Keras
- TensorFlow
- LIBSVM

- `scikit-learn`
- `XGBoost`

The TensorFlow converter is available as of coremltools 3.0. To convert TensorFlow models to Core ML format you can also use the separate `tfcoreml` package.

For converting PyTorch, MXNet, or ONNX models you'll need separate converters in addition to coremltools.

Doing it by hand

If a converter does not exist for your model format, or the existing converter cannot handle your trained model for some reason, you can always write your own converter:

- `NeuralNetworkBuilder` from the `coremltools.models.neural_network.builder` module lets you construct a neural network layer-by-layer.
- `KNearestNeighborsClassifierBuilder` from `coremltools.models.nearest_neighbors` lets you construct a k-Nearest Neighbors classifier.
- `TreeEnsembleClassifier` and `TreeEnsembleRegressor` from `coremltools.models.tree_ensemble` let you construct tree ensemble models such as random forests.
- `Pipeline`, `PipelineClassifier`, and `PipelineRegressor` from `coremltools.models.pipeline` let you build pipelines, which consist of several models in sequence that act as if they were a single model.

Note: Because `mlmodel` files are in protobuf format, it's also possible to create them without using coremltools at all by [directly writing protobuf](#) messages.

Making predictions

To make predictions with a converted Core ML model, you use the `coremltools.models.MLModel` object. This is primarily useful for [verifying the model](#) was converted correctly.

Note: This functionality is only available on macOS 10.13 or better. Even though coremltools works fine on Linux for converting models, it can't make predictions from Linux.

Nurse, scalpel please!

The other functionality inside the `coremltools.models` module, as well as `coremltools.utils`, is used to perform surgery on existing Core ML models.

For example, you can use these functions to quantize the model's weights, to remove or rename layers, and anything else you might want to do to patch up your Core ML model. You'll see many examples in this guide.

Installing coremltools

The easiest way to install coremltools is using the Python package manager `pip`:

```
pip install -U coremltools
```

coremltools works with both Python versions 2 and 3. You can use it on macOS but also from Linux, although certain functionality that requires the Core ML framework, such as making predictions, only works on the Mac.

Tip: coremltools is an active open source project and the official [PyPI package](https://pypi.python.org/pypi/coremltools)⁷ is not always up-to-date with the latest changes. If you don't mind living on the edge, or if the official release does not yet have a bug fix or a new feature that you need, you can install coremltools directly from GitHub:

```
pip install -U git+https://github.com/apple/coremltools.git
```

For example, if loading a model using coremltools gives an error such as the following, then try installing the latest coremltools directly from the GitHub repo.

⁷<https://pypi.python.org/pypi/coremltools>

```
Error compiling model: "Error reading protobuf spec. validator error:  
The .mlmodel supplied is of version 3, intended for a newer version of  
Xcode. This version of Xcode supports model version 2 or earlier.
```

But be warned that using the GitHub version may cause problems of its own... It is under active development, which occasionally breaks things. I often find myself switching between the GitHub version and the last official release.

Using coremltools

To use coremltools in your Python script, write:

```
import coremltools
```

When Python loads the coremltools package, you may get warning messages such as the following:

```
WARNING:root:Keras version 2.3.0 detected. Last version known to be fully  
compatible of Keras is 2.2.4 .  
WARNING:root:TensorFlow version 1.14.0 detected. Last version known to be  
fully compatible is 1.13.1 .
```

Usually you can safely ignore these messages, but be aware that you may run into errors when converting your Keras or TensorFlow models. Especially the Keras API tends to change a lot between releases, and if coremltools depends on some functionality that got moved or renamed, the conversion will fail.

Tip: If coremltools throws an exception while converting your model, create a new virtualenv and install the last known compatible versions of Keras and TensorFlow into that environment — Keras 2.2.4 and TensorFlow 1.13.1 according to the warning, although by the time you read this the version numbers may have changed — and try the conversion from inside this virtualenv.

In the next chapters you'll see how to use coremltools to convert models. In parts 3 and 4, you'll learn how coremltools can be used to take apart models, fix issues, and put them back together again.

Part 2: Converting Models

To use a machine learning model with Core ML, you first need to convert it to the `mlmodel` format. This part of the book explains how to avoid potential pitfalls.

Image Preprocessing

The most common reason for getting wrong predictions out of Core ML is using incorrect image preprocessing options.

A `CVPixelBuffer` usually contains pixels in ARGB or BGRA format where each color channel is 8 bits. That means the pixel values in the image are between 0 and 255.

But your machine learning model may not actually expect pixels between 0 and 255. Other common choices are:

- between 0 and 1
- between -1 and +1
- between -2 and +2
- between -128 and +128 with the average values of R, G, and B subtracted
- the color channels in BGR order instead of RGB

If your model expects pixel values in a different range than 0 – 255, you need to let Core ML know about this. Core ML can then convert the `CVPixelBuffer` into the format that your model understands, using a special *preprocessing* stage in the neural network.

This is very important! With the incorrect preprocessing settings, `coremltools` will create an `mlmodel` file that will interpret your input images wrongly. The model will be working on data it does not understand — and that produces results that don't make sense. [Garbage in, garbage out](https://en.wikipedia.org/wiki/Garbage_in,_garbage_out).⁸

Note: For grayscale images, it's also important to know what value is considered black and what value is considered white. In some models 0 is black and 1 is white, but in others 1 is black and 0 is white.

⁸https://en.wikipedia.org/wiki/Garbage_in,_garbage_out

What is the correct preprocessing for my model?

Usually the Python training code for the model has a `preprocess_input()` or `normalize()` function that converts the pixels into the correct format. Keras, for example, uses the following arithmetic for models trained on ImageNet:

```
def preprocess_input(x):  
    x /= 127.5  
    x -= 1.0  
    return x
```

You need to tell Core ML to perform this exact same preprocessing step.

If you did not train the model yourself, but you're using a pretrained model that you downloaded from the web, you should try to find out what sort of preprocessing is done on the images before they go into the first neural network layer.

RGB or BGR?

Most training tools will load images in the RGB pixel order. Caffe, however, loads images as BGR. If your training tool uses OpenCV to load images, it probably uses the BGR pixel order too.

This is no problem, as Core ML will automatically swap the red and blue color channels if necessary. But you do need to tell it whether your model expects RGB or BGR.

If you get the pixel order wrong, your model may still appear to make reasonable predictions — but the results won't be as good as they could be! This can be a tricky bug to find.

See chapter [Making Sure the Input is Correct](#) for a way to make 100% sure your images are in the right pixel order.

The available preprocessing options

If you're using `coremltools` or `tfcoreml` to convert the model, you can specify the preprocessing options in the Python script that converts the model.

The following options are available:

- `image_scale`: the pixel values will be multiplied by this number
- `red_bias`, `blue_bias`, `green_bias`: these will be added to the RGB values of each pixel
- `gray_bias`: like the RGB biases but for grayscale images
- `is_bgr`: you typically need to set this to `True` for Caffe models, `False` otherwise

Instead of the channel bias values, the Caffe converter can also use a mean image from a “binaryproto” file. This is literally the average image over all the images in the training set.

If the model has multiple input images, you can supply a dictionary instead of a single value, to give each input its own preprocessing options. For example:

```
red_bias = { "image1": -90, "image2": -110 }
```

The trick is to choose appropriate values for these options!

How does it work?

The standard formula for normalization of data is:

```
normalized = (data - mean) / std
```

First you subtract the mean value, then you divide by the standard deviation of the data. What Core ML does is similar but not quite the same:

```
normalized = data*image_scale + bias
```

The `red_bias`, `blue_bias` and `green_bias` values act as the normalization mean for the three color channels (or just the `gray_bias` when the image only has one channel). The difference is that the mean is subtracted while the bias is added, so if you’re given the mean you need to flip its sign to get the bias:

```
red_bias   = -red_mean  
green_bias = -green_mean  
blue_bias  = -blue_mean
```

The `image_scale` acts like the standard deviation (`std` in the first formula), except that it is applied *before* the bias is added, while the standard deviation would be applied after the biases are added.

So if your training script supplies a standard deviation, you can use the `image_scale` option for that, but you have to take the reciprocal:

```
image_scale = 1 / std
```

and you also have to apply this to the bias values to make the math work out:

```
red_bias    /= std
green_bias  /= std
blue_bias   /= std
```

Note: If you need to scale each color channel individually, you cannot use `image_scale`. See the chapter [Using a Different Scale for Each Color Channel](#).

Usage examples

0 - 255

If your model expects pixel values in the range 0 – 255, you don't have to set preprocessing options. Simply use the default values:

```
image_scale = 1
red_bias = 0
green_bias = 0
blue_bias = 0
```

0 - 1

If your model expects pixel values in the range 0 – 1, you should set:

```
image_scale = 1/255.0
red_bias = 0
green_bias = 0
blue_bias = 0
```

The bias values use the default value of 0 and have no effect.

-1 ... +1

If your model expects pixel values in the range -1 to +1, you should set:

```
image_scale = 2/255.0
red_bias = -1
green_bias = -1
blue_bias = -1
```

This first multiplies the pixels by 2/255 , which is the same as dividing by 127.5. Now they are between 0.0 and 2.0. Then it subtracts 1.0 from each color channel to put the pixel values between -1.0 and 1.0. This is the normalization used by Keras for ImageNet-based models.

Caffe models

Models trained with Caffe on the ImageNet dataset will usually subtract the mean RGB values:

```
red_bias   = -123.68
green_bias = -116.779
blue_bias  = -103.939
is_bgr     = True
```

Now the pixels are roughly in the range -128 to +128.

Note that Caffe loads images in the BGR pixel order. Your training script may have the mean color values listed as [103.939, 116.779, 123.68]. The first element in this list is for the blue channel color, not red. This can be confusing, so pay attention when you copy-paste these values from your training scripts!

This is also why `is_bgr` should be set to `True`.

Sometimes Caffe models also have a standard deviation applied during preprocessing. The options then become:

```
image_scale = 0.017
red_bias    = -123.68 * 0.017
green_bias  = -116.779 * 0.017
blue_bias   = -103.939 * 0.017
is_bgr      = True
```

Here, the standard deviation is 58.8, which gives an image scale of $1/58.8 = 0.017$. Since scaling happens before the bias is added, you will need to multiply the red/green/blue_bias by this scaling factor as well.

For Caffe models you can also specify the path to your **mean.binaryproto** file, if you have one of those, that contains the average RGB values for the entire image. You would use this instead of red/green/blue_bias.

Mean & standard deviation

If you are given a mean for the RGB channels and a standard deviation, you can set the preprocessing options as follows:

```
image_scale = 1.0 / std
red_bias    = -red_mean / std
green_bias  = -green_mean / std
blue_bias   = -blue_mean / std
```

Here, the standard deviation is assumed to be the same for each color channel.

Note: The coremltools converters require that the input pixels are in the range 0 - 255. But some training tools assume that pixels are between 0 and 1. If the mean and standard deviation are given for the 0 - 1 range, you will need to divide the image_scale by 255.

Per-image mean and standard deviation

If your model does the following, you can't use the preprocessing options for this:

```
normalized = (img - np.mean(img)) / np.std(img)
```

The image_scale and bias options are assumed to be fixed constants that are the same for every image. If you want to normalize each image with its own mean and standard deviation values, look into adding a MeanVarianceNormalizeLayer to the model instead.

Different scaling for each color channel

Torch models often do their image preprocessing as follows:

```
def preprocess_input(x):  
    x /= 255.0  
    mean = [0.485, 0.456, 0.406]  
    std = [0.229, 0.224, 0.225]  
    return (x - mean) / std
```

This uses a separate standard deviation for each color channel. You can approximate this with the following options:

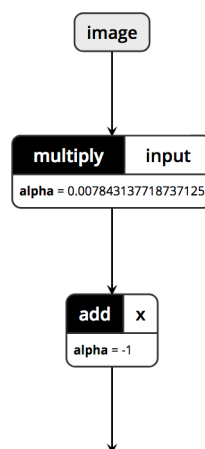
```
image_scale = 1.0 / (255.0 * 0.226)  
red_bias = -0.485 / 0.226  
green_bias = -0.456 / 0.226  
blue_bias = -0.406 / 0.226
```

However, this isn't exactly the same thing: it uses an average standard deviation of 0.226 but the original model has a slightly different standard deviation for each color channel.

The proper solution is to insert a “scale layer” at the beginning of the model that can scale each channel by its own standard deviation. However, this requires model surgery. You can read more about it in the chapter [Using a Different Scale for Each Color Channel](#).

The model already has its own preprocessing layers

Sometimes the preprocessing calculations are part of the model already. For example, at the start of the following graph there is a “multiply” layer followed by an “add” layer:



The multiply layer multiplies the pixels by the constant value 0.0078431 which is the same as $1/127.5$. The add layer subtracts the constant 1. Now the pixels are in the range -1 to +1.

If your model does something similar, then you should not supply any preprocessing options to Core ML, otherwise the preprocessing happens twice.

Changing the preprocessing for an existing model

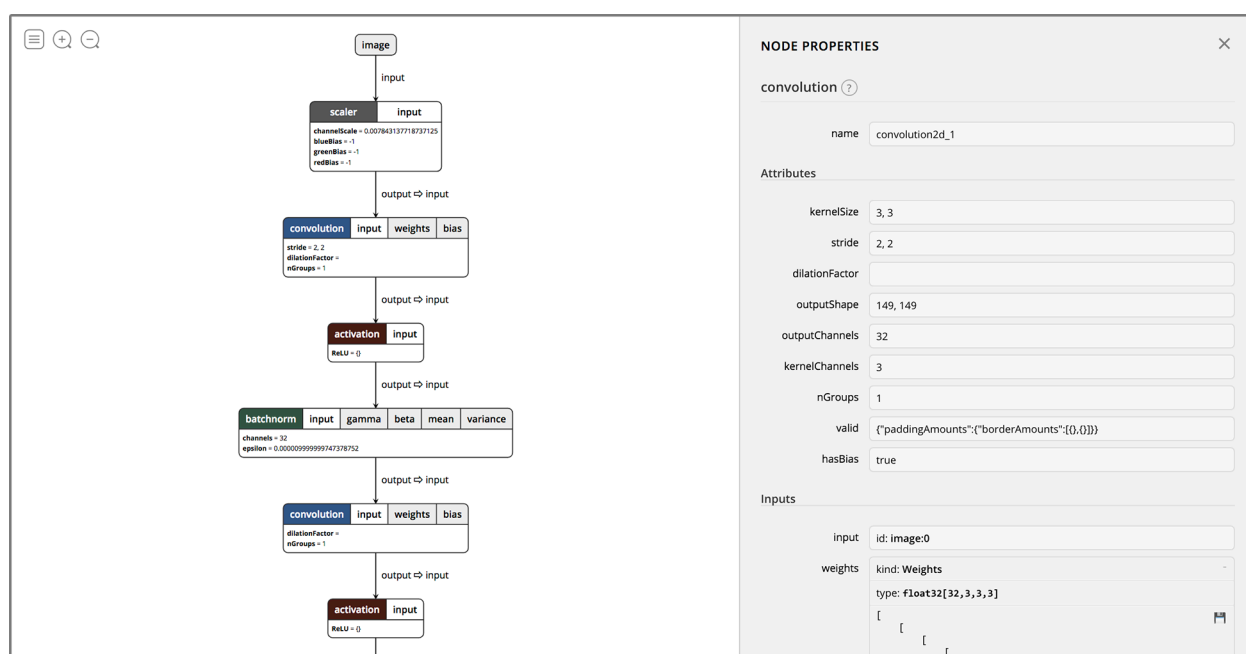
To examine and possibly change the preprocessing options from an existing mlmodel, see the chapter [Changing the Image Preprocessing Options](#).

Part 3: Examining Models

The `mlmodel` file format is the cornerstone of Core ML. This part of the book dives deep into how this file format works. Knowing this is important to understand how Core ML does things, but also how to fix issues with your `mlmodel` files.

Viewing Models With Netron

An indispensable tool for examining machine learning models is **Netron** by Lutz Roeder. It supports many different model formats, including Core ML `mlmodels`. Netron is open source, completely free, and runs on macOS, Linux, Windows and even in your web browser. Download and installation instructions are on the [Netron GitHub page](#)⁹.



Netron shows the model on the left and details about the selected layer on the right. You can also examine the weights and biases for your layers, and even save them to disk as NumPy arrays. Click on the model's input or output to see the properties of the `mlmodel` file itself, including the Core ML specification version it uses (under “format”).

Using this tool to inspect your models is highly recommended!

Tip: After converting a model to Core ML, open both the original model and the converted model in Netron. Put them side-by-side to compare how the different layers were translated to Core ML.

⁹<https://github.com/lutzroeder/netron>

The mlmodel File Format

To use a machine learning model with Core ML it needs to be in the **mlmodel** format.

The mlmodel file describes the layers in your model, the input and outputs, the class labels, and any preprocessing that needs to happen on the data. It also contains all the learned parameters: the weights and biases.

For neural network models that are *updatable* — models that can be trained on device — the mlmodel file also contains the loss function, the optimizer, and the training hyperparameters.

Everything you need to use the model with Core ML is inside this one file.

You can use `coremltools` to convert your trained models into an mlmodel file. Hopefully all goes well and you're good to go! But if your converted model doesn't work as expected with Core ML, you'll need to examine the mlmodel file and possibly do some surgery on it. Therefore, it makes sense to learn how mlmodel files work.

Knowing how to read the mlmodel specification also gives you a better idea of what Core ML can do and what it cannot.

Protobuf files

The mlmodel file format is an open standard that is based on Protocol Buffers, or **protobuf**, a method of serializing structured data that was invented by Google. You can compare protobuf to XML or JSON, but it produces smaller files and is faster. Models from TensorFlow and Caffe are saved using protobuf formats too.

The mlmodel specification is defined in a number of `.proto` files. These are text files containing the schema that is used by protobuf to read and write mlmodel files. (Core ML uses the proto3 version of the protocol buffers language.)

You can find these proto files inside the `coremltools` GitHub repo, in the folder [mlmodel/format](#)¹⁰. There is also a copy in the [Turi Create repo](#)¹¹.

¹⁰<https://github.com/apple/coremltools/tree/master/mlmodel/format>

¹¹https://github.com/apple/turicreate/tree/master/src/external/coremltools_wrap/coremltools/mlmodel/format

Branch: master


coremltools / mlmodel / format /

Create new file

Upload files


Find file

History

 Sohaib Qureshi Consolidate changes for coremltools 2.0 public release


Latest commit f11c43e on Aug 30

..

 [ArrayFeatureExtractor.proto](#)


Add Linux build support

a year ago

 [BayesianProbitRegressor.proto](#)


Consolidated changes from 2.0b1 release

4 months ago

 [CategoricalMapping.proto](#)


Add Linux build support

a year ago

 [CustomModel.proto](#)


Consolidated changes from 2.0b1 release

4 months ago

 [DataStructures.proto](#)


Add Linux build support

a year ago

 [DictVectorizer.proto](#)

minor proto sp fixes

9 months ago

 [FeatureTypes.proto](#)

Consolidated changes from 2.0b1 release

4 months ago

Everything that Core ML can do is described by these proto files. If it's not in here, Core ML does not support it.

Note: Whenever a new capability is added to Core ML, such as a new neural network layer type, it is added to these proto files. One way to find out what functionality is supported by the [different versions of Core ML](#) is to look at the commit history for this folder.

Model.proto

The main file in the mlmodel specification is **Model.proto**. The most important definition in this file is at the bottom and is also named `Model`. Here is an excerpt:

```
message Model {
  int32 specificationVersion = 1;
  ModelDescription description = 2;
  bool isUpdatable = 10;

  oneof Type {
    PipelineClassifier pipelineClassifier = 200;
    PipelineRegressor pipelineRegressor = 201;
    Pipeline pipeline = 202;

    GLMRegressor glmRegressor = 300;
    SupportVectorRegressor supportVectorRegressor = 301;
    TreeEnsembleRegressor treeEnsembleRegressor = 302;
    ...
  }
}
```

The protobuf syntax may take a little getting used to, but it's not really that difficult once you understand the terminology.

As with all data serialization formats, the different data structures that can occur in the serialized files need to be defined somewhere. In protobuf parlance, a data structure is called a “message”.

The code snippet above defines the `Model` data structure, which is the top-level message in the `mlmodel` format. `Model` consists of four fields:

- `specificationVersion`, an integer
- `description`, a so-called `ModelDescription` object
- `isUpdatable`, a boolean
- `andType`, a special `oneof` structure that defines the actual type of the model, such as a `SupportVectorRegressor` OR a `TreeEnsembleRegressor`.

Notice how each field in the message definition consists of a data type, a name, and a number. This number is called the “tag”. When the `mlmodel` is written to disk as a binary file, the message objects are stored as dictionaries. Each field in the serialized message is a key-value pair where the key is the numeric tag. This is more efficient than storing the actual field name.

Note: Occasionally when I read through these proto files, I get confused about the tag values. When it says `int32 specificationVersion = 1;` that makes it seem you're assigning the value 1 to the specification version of the model. But that's not the case: it just says that the dictionary key for this field is 1, just like how a dictionary in Swift would use a hash value for the key. It's best to simply ignore these tags as you read through the proto files — they're just an implementation detail of how protobuf files work and not important for understanding Core ML.

Digging through the proto files

Most of the fields in the `Model` message refer to other messages, such as `ModelDescription`. You can find this message definition in `Model.proto` too:

```
/**
 * A description of a model,
 * consisting of descriptions of its input and output features.
 * Both regressor and classifier models require the name of the
 * primary predicted output feature (`predictedFeatureName`).
 * Classifier models can specify the output feature containing
 * probabilities for the predicted classes
 * (`predictedProbabilitiesName`).
 */
message ModelDescription {
  repeated FeatureDescription input = 1;
  repeated FeatureDescription output = 10;

  string predictedFeatureName = 11;
  string predictedProbabilitiesName = 12;

  repeated FeatureDescription trainingInput = 50;

  Metadata metadata = 100;
}
```

This message has six fields. There is also a comment at the top that describes what these fields mean. Not everything in the proto files is documented but there are plenty of comments that help you figure out what's going on.

The ModelDescription structure contains the data that you see when you open the mlmodel in Xcode:

- One or more FeatureDescriptions for the model's inputs. This is a repeated field, which is an array or list of such objects.
- Likewise, a list of FeatureDescription objects for the model's outputs.
- There is also a list of FeatureDescription objects for *training inputs*. These are used instead of the regular inputs when you're training the model.
- A predictedFeatureName and a predictedProbabilitiesName field, both strings.
- A Metadata object with additional information about the model.

If you want to know more about what a FeatureDescription or a Metadata is, look for its message definition elsewhere in the proto file or in one of the other proto files.

Classifiers and regressors

As the comment at the top of message `ModelDescription` says, `predictedFeatureName` must be the name of the model's main output for regressor and classifier models. And if the model is a classifier, `predictedProbabilitiesName` should be the name of the output that predicts the probabilities.

Core ML makes a distinction between regressors, classifiers, and “other” models.

For example, there is a `NeuralNetworkRegressor` model type, a `NeuralNetworkClassifier` model type, and a plain `NeuralNetwork` model type. For classical machine learning models such as Support Vector Machines or Tree Ensembles, there are also separate regressor and classifier variants.

This allows Core ML to perform some extra magic that makes it easier to use these models. If a model is a classifier, and thus fills in the `predictedProbabilitiesName` field, Core ML can automatically turn the numbers from that output into a dictionary of class names and probabilities.

Of course, for this to work the mlmodel must also contain the class names. You can see this in the message describing the classifier model, for example `GLMClassifier` looks like this:

```
message GLMClassifier {  
    ...  
    oneof ClassLabels {  
        StringVector stringClassLabels = 100;  
        Int64Vector  int64ClassLabels = 101;  
    }  
}
```

Note: The fields inside the protobuf messages are always optional. If a model is not a classifier or regressor, simply don't fill in the `predictedFeatureName` and `predictedProbabilitiesName` fields and they will not appear in the serialized mlmodel. If a field is required by Core ML, the comments say so.

Supported model types

Core ML can handle several different types of models, such as:

- linear regression and logistic regression
- support vector machines (SVM)
- tree ensembles such as random forests and boosted trees
- neural networks: feed-forward, convolutional, recurrent
- k-Nearest Neighbors

Most of these can be used for regression as well as classification. In addition your model can contain typical ML preprocessing steps like one-hot encoding, feature scaling, imputation of missing values, and so on.

This kind of high-level overview is all well and good, but to know exactly what Core ML can do, you have to look inside the proto files. The list of supported model types is in `message Model` from `Model.proto`, which you saw earlier.

`Model` has a `oneof` field that lists all the available model types. Here's the entire list as of Core ML 3:

```
oneof Type {
  PipelineClassifier pipelineClassifier = 200;
  PipelineRegressor pipelineRegressor = 201;
  Pipeline pipeline = 202;

  GLMRegressor glmRegressor = 300;
  SupportVectorRegressor supportVectorRegressor = 301;
  TreeEnsembleRegressor treeEnsembleRegressor = 302;
  NeuralNetworkRegressor neuralNetworkRegressor = 303;
  BayesianProbitRegressor bayesianProbitRegressor = 304;

  GLMClassifier glmClassifier = 400;
  SupportVectorClassifier supportVectorClassifier = 401;
  TreeEnsembleClassifier treeEnsembleClassifier = 402;
  NeuralNetworkClassifier neuralNetworkClassifier = 403;
  KNearestNeighborsClassifier kNearestNeighborsClassifier = 404;

  NeuralNetwork neuralNetwork = 500;
  ItemSimilarityRecommender itemSimilarityRecommender = 501;

  CustomModel customModel = 555;
  LinkedModel linkedModel = 556;

  OneHotEncoder oneHotEncoder = 600;
  Imputer imputer = 601;
  FeatureVectorizer featureVectorizer = 602;
  DictVectorizer dictVectorizer = 603;
```

```

Scaler scaler = 604;
CategoricalMapping categoricalMapping = 606;
Normalizer normalizer = 607;
ArrayFeatureExtractor arrayFeatureExtractor = 609;
NonMaximumSuppression nonMaximumSuppression = 610;

Identity identity = 900;

CoreMLModels.TextClassifier textClassifier = 2000;
CoreMLModels.WordTagger wordTagger = 2001;
CoreMLModels.VisionFeaturePrint visionFeaturePrint = 2002;
CoreMLModels.SoundAnalysisPreprocessing soundAnalysisPreprocessing = 2003;
CoreMLModels.Gazetteer gazetteer = 2004;
CoreMLModels.WordEmbedding wordEmbedding = 2005;
}

```

A `oneof` declaration means that the message can only include one of these fields. This is like a switch statement in Swift, where each entry is one of the cases.

I want to highlight some of the model types:

- Identity is just for testing. It simply returns the inputs as the outputs.
- GLMRegressor and GLMClassifier are for linear regression and logistic regression, respectively. These models are created by the scikit-learn converter.
- SupportVectorRegressor and SupportVectorClassifier are — what else? — for support vector machines. These models are from LIBSVM or scikit-learn.
- TreeEnsembleRegressor and TreeEnsembleClassifier are for decision trees, random forests, and boosted trees. These models are from XGBoost or scikit-learn.
- There are three types of neural nets: NeuralNetworkRegressor, NeuralNetworkClassifier, and a generic NeuralNetwork.
- KNearestNeighborsClassifier is especially useful for building models that can be trained on-device on data provided by the user.
- CustomModel is for wrapping your own algorithms into an mlmodel. When using a custom model, the mlmodel file is only a container for your model's learned parameters and any metadata. It doesn't know anything about the actual model algorithm. See also the chapter on [custom models](#).
- The CoreMLModels are the model types that you can train with Create ML or Turi Create.

There are also a number of model types for feature engineering tasks (most of these are used with scikit-learn):

- `OneHotEncoder` and `CategoricalMapping`: convert categorical features into numbers
- `FeatureVectorizer` and `DictVectorizer`: for converting features to arrays
- `Imputer`: fills in missing values in the data
- `Scaler` and `Normalizer`: for normalizing the input data

Also worth mentioning is a model type that is specifically meant for post-processing the output data:

- `NonMaximumSuppression`: filters bounding box predictions from object detection models

Usually you'd do something like one-hot encoding or normalizing before passing the data to a classifier model, and non-maximum suppression is done after running an object detection model. So it may seem a little weird that these things are considered to be separate models by Core ML — especially because there is only a single `Model` object in the mlmodel file.

Fear not! This is what the `Pipeline` model type is for. A pipeline lets you combine multiple models as if they are one model. The `Pipeline` message is defined as follows:

```
message Pipeline {  
    repeated Model models = 1;  
    repeated string names = 2;  
}
```

A pipeline is nothing more than a list of `Model` objects, letting you nest multiple `Model` objects inside a single mlmodel file. The output of one model is the input to the next.

For example, when you train an object detection model with Turi Create, it produces an mlmodel file whose model is of type `Pipeline`. The first model in this pipeline is a `NeuralNetwork` that implements YOLO and predicts a fixed number of bounding boxes. The second model in this pipeline is `NonMaximumSuppression`, which filters the predicted bounding boxes and only keeps the best ones.

Core ML 3 also introduced the concept of a `LinkedModel`, a placeholder inside a pipeline that refers to another mlmodel file. This allows you to re-use the same model in multiple pipelines, which is more efficient than giving each pipeline an identical copy.

Read more about pipelines in the chapter [Building Pipeline Models](#).

Note: For a full description of what each of these model types does, look at their corresponding proto files to see what kind of functionality they support. The documentation isn't always perfect, but you should be able to get a pretty good idea from the proto file. Bonus tip: Also look at the source code for the different converters from the `coremltools` repo to see how they use these model types.

Neural networks

If you're interested in using Core ML for deep learning, check out **NeuralNetwork.proto** to see all the neural network layers you can use. This is by far the largest proto file from the mlmodel specification. Fortunately, it has extensive documentation.

The main data structure for a neural network model is:

```
message NeuralNetwork {
    repeated NeuralNetworkLayer layers = 1;
    repeated NeuralNetworkPreprocessing preprocessing = 2;

    NeuralNetworkMultiArrayShapeMapping arrayInputShapeMapping = 5;
    NeuralNetworkImageShapeMapping imageInputShapeMapping = 6;
    NetworkUpdateParameters updateParams = 10;
}
```

This contains a list (repeated) of the layers, as well as a list of `NeuralNetworkPreprocessing` definitions, one for each input in the model. Specification version 4 added the other three fields, which allow for [dynamic tensor shapes](#) and on-device training, respectively.

Want to know what sort of preprocessing is available in Core ML? Look at the message definition:

```
message NeuralNetworkPreprocessing {
    string featureName = 1;
    oneof preprocessor {
        NeuralNetworkImageScaler scaler = 10;
        NeuralNetworkMeanImage meanImage = 11;
    }
}
```

The `featureName` is the name of the model's input to which this preprocessing gets applied. You have the choice between a `NeuralNetworkImageScaler` or a `NeuralNetworkMeanImage`. Let's look at the scaler to see what that does:

```
message NeuralNetworkImageScaler {
  float channelScale = 10;
  float blueBias = 20;
  float greenBias = 21;
  float redBias = 22;
  float grayBias = 30;
}
```

These are the same [preprocessing options](#) that you supply to `coremltools` when you convert the model. Now you know that these are stored inside a `NeuralNetworkImageScaler` object, inside a `NeuralNetworkPreprocessing` object, inside a `NeuralNetwork` object, inside a `Model` object, inside the `mlmodel` file. Phew!

The definition for `NeuralNetworkLayer` is pretty big, because it encompasses all the possible layer types:

```
message NeuralNetworkLayer {
  string name = 1;
  repeated string input = 2;
  repeated string output = 3;

  repeated Tensor inputTensor = 4;
  repeated Tensor outputTensor = 5;
  bool isUpdatable = 10;

  oneof layer {
    ConvolutionLayerParams convolution = 100;
    PoolingLayerParams pooling = 120;
    ActivationParams activation = 130;
    InnerProductLayerParams innerProduct = 140;
    EmbeddingLayerParams embedding = 150;
    /* and many more... */
  }
}
```

As you can see, in Core ML a neural network layer has a name, one or more inputs, one or more outputs, and a layer type (from the `oneof`). As of Core ML 3, layers can also be marked as being updatable for on-device training.

The inputs and outputs are strings with the output and input names of the layers that are connected to this one. Most layers will have just one input and one output. A “concat” layer can have multiple inputs; a “split” layer will produce multiple outputs.

Using these three fields — `name`, `input` and `output` — you can describe the complete graph of the layers in the neural network.

To know more about a particular layer type, check out its message definition.

Note that in the mlmodel specification, an activation function is considered to be a separate layer. To see the supported activation functions, look at ActivationParams:

```
message ActivationParams {  
  oneof NonlinearityType {  
    ActivationLinear linear = 5;  
    ActivationReLU ReLU = 10;  
    ActivationLeakyReLU leakyReLU = 15;  
    ActivationThresholdedReLU thresholdedReLU = 20;  
    ...  
  }  
}
```

And so on... everything you want to know about Core ML's neural network layers is inside this proto file. I suggest reading through it, there are a lot of comments explaining how everything works.

Tip: I wrote a [blog post](https://machinethink.net/blog/new-in-coreml3/)¹² about everything that's new in Core ML 3. This also describes the available model types and neural network layer types in more detail, so be sure to check that out.

¹²<https://machinethink.net/blog/new-in-coreml3/>

Using the Spec to Edit Models

After you convert a model to Core ML format using `coremltools`, you get an `MLModel` object from the `coremltools.models` module.

You can also create an `MLModel` object by loading an `mlmodel` file:

```
model = coremltools.models.MLModel("YourModel.mlmodel")
```

The `coremltools.models.MLModel` class provides a very basic interface to using Core ML from Python. You can use it to change the model's metadata and make predictions. (Note that this is not the same as the `MLModel` class from `CoreML.framework`. We're in Python land here.)

That's nice, but to make more extensive changes to the model definition you need the **spec**, which contains the **protobuf objects** that describe the model's properties in detail. With the spec you can add new layers, remove layers, rename inputs and outputs, and change the model in pretty much any way you please.

The Python `MLModel` object is really just a very simple wrapper around such a spec object. For most model surgery tasks, you'll need to use the spec directly.

Tip: I prefer to work interactively from the Python REPL when dealing with spec objects. I recommend using [IPython](https://ipython.readthedocs.io/en/stable/)¹³ or even a [Jupyter notebook](https://jupyter.org)¹⁴.

Getting the spec

There are a few ways to get the spec object. You can get it from the `MLModel`:

```
spec = model.get_spec()
```

This creates a deep copy of the spec, so changing anything in this copy does not affect the existing `MLModel` object.

If you don't want to use a copy but directly make changes to the `MLModel` object, you can do:

¹³<https://ipython.readthedocs.io/en/stable/>

¹⁴<https://jupyter.org>

```
spec = model._spec
```

In Python, symbols starting with an underscore are private. You're not really supposed to use `_spec` directly, but there's also no one stopping you. Any changes you make to `spec` are now immediately reflected in the `MLModel`. (Although if you want to make predictions with the changed spec, you'll need to create a new `MLModel` instance, which runs the Core ML compiler again.)

The third way to get a spec is to skip `MLModel` altogether and load it directly:

```
spec = coremltools.utils.load_spec("YourModel.mlmodel")
```

A quick way to make sure the spec was loaded correctly is to print the model description:

```
print(spec.description)
```

This prints out the model's metadata as well as the descriptions of its inputs and outputs. (This is the contents of the `ModelDescription` message from `Model.proto`.)

Tip: don't do `print(spec)`. That will print out the entire model, including all of its learned parameters. You'll regret it!

Making changes using the protobuf API

The spec consists of protobuf objects. To see this, run the following command from the Python REPL:

```
type(spec)
```

This prints `Model_pb2.Model`. The `_pb2` suffix means this is an object that is described by the `Model.proto` file.

All the protobuf “message” objects in the spec have similar types. For example, writing `type(spec.description)` prints out `Model_pb2.ModelDescription`, and so on. These types literally correspond to the definitions in the proto files. Therefore, understanding the `mlmodel` specification is important in order to work with the spec (see the previous chapter).

Because these are all protobuf objects, you will need to use the Python protobuf API to manipulate them. This API is actually quite convenient to use, but not always obvious if you're new to it.

To change a field that is an integer or string, you can simply assign it a new value. For example, the human-readable description of the model that is shown in Xcode:

```
spec.description.metadata.shortDescription = "This is my awesome model!"
```

If you don't want to include such a description, simply set it to the empty string "". You can also reset the field to its default value:

```
spec.description.metadata.ClearField("shortDescription")
```

All the fields in the spec objects are optional. Numeric fields and string fields will always have a default value (0 or the empty string) but fields that are messages may not be present at all. To inspect if the model has a certain message, you can write:

```
spec.description.HasField("metadata")
```

If a field is a oneof, you can use `WhichOneof()` to figure out which one it is (notice the spelling of `Oneof`!). Recall that `spec` is an instance of the `Model` message that has a field `oneof Type` that determines what sort of machine learning model is in the `mlmodel` file.

```
print(spec.WhichOneof("Type"))
```

This may print something like `neuralNetworkClassifier`. You can look at the properties of this object by writing:

```
spec.neuralNetworkClassifier.someProperty
```

Where `someProperty` is a field from the message type `NeuralNetworkClassifier`. (Don't try to print out all of `spec.neuralNetworkClassifier` as it dumps the contents of all the layers and their learned parameters.)

If you write `spec.neuralNetwork` or `spec.treeEnsembleClassifier` or any of the other model type names, you'll get `None` as the answer (or just nothing) because those are not the correct type for this particular `mlmodel`. The name you use, in this case `spec.neuralNetworkClassifier`, has to match the result from `WhichOneof("Type")`.

Note: `type(spec.neuralNetworkClassifier)` is `NeuralNetwork_pb2.NeuralNetworkClassifier` because this message definition is located in the file `NeuralNetwork.proto`.

The `NeuralNetworkClassifier` has a repeated field called `layers` that consists of zero or more `NeuralNetworkLayer` objects. This works like a list and so you can iterate through it:

```
for layer in spec.neuralNetworkClassifier.layers:
    print(layer.name)
```

Or if you just want to look at the convolution layers:

```
for layer in spec.neuralNetworkClassifier.layers:
    if layer.HasField("convolution"):
        print(layer.name)
```

Or equivalently:

```
for layer in spec.neuralNetworkClassifier.layers:
    if layer.WhichOneof("layer") == "convolution":
        print(layer.name)
```

I hope you're following along with `NeuralNetwork.proto` to check that this makes sense!

To look at a specific layer you can write:

```
layer = spec.neuralNetworkClassifier.layers[0]
print("name", layer.name)
print("inputs", layer.input)
print("outputs", layer.output)
```

Because `input` and `output` are repeated fields, you can treat them as Python lists. For example, to add a second input to the layer:

```
layer.input.append("another_input")
print(len(layer.input))
```

Of course, if you now save the spec to an `mlmodel` file, Xcode won't be able to load it because this new input isn't defined anywhere. So let's delete it again:

```
del layer.input[-1]
```

You can also add new layers. But because layers are message objects, you cannot write `append()` like you just did. That only works for simple types like strings and numbers. You first have to add a new layer object:

```
new_layer = spec.neuralNetworkClassifier.layers.add()
```

This `new_layer` object is of type `NeuralNetwork_pb2.NeuralNetworkLayer` but it doesn't have any contents yet. To give it something to do, simply assign values to its fields:

```
new_layer.name = "My new layer"  
new_layer.input.append("old_output")  
new_layer.output.append("new_output")
```

This inserts the new layer after the layer named "old_output", which now becomes the input to this new layer.

Note: You can't write `new_layer.input = ["old_output"]`. That is an error with the Python protobuf API. Since these are not regular Python lists but protobuf objects, the API is sometimes a bit different than what you may be used to.

The new layer doesn't have a type yet. This is where the protobuf API is a little weird. Let's say you want to make this a linear activation layer. According to the proto file, activation layers are described by the `ActivationParams` message.

Unlike what you might perhaps expect, you don't create a new `ActivationParams` object and then assign it to the layer. This doesn't work:

```
new_layer.activation = coremltools.proto.NeuralNetwork_pb2.ActivationParams()
```

Instead, you simply write something like:

```
new_layer.activation.linear.alpha = 1
```

By accessing the `.activation` field you're telling protobuf you want to this layer to have an `ActivationParams` object. By writing `.linear` you're saying it should be an `ActivationLinear`. And by accessing the `.alpha` field you actually make all of this happen.

If you now print out the properties of `new_layer`, you should see the following:

```
name: "My new layer"
input: "old_output"
output: "new_output"
activation {
  linear {
    alpha: 1
  }
}
```

Sometimes messages don't have any fields, such as `ActivationReLU`. To change the activation function of this layer to `ReLU`, you have to write the following instead:

```
new_layer.activation.ReLU.SetInParent()
```

And if you wanted to change this layer to a completely different type, write something like this:

```
new_layer.innerProduct.inputChannels = 100
```

Now this layer magically becomes an inner product layer — what Core ML calls a fully-connected layer — of type `NeuralNetwork_pb2.InnerProductLayerParams`.

Note: In the proto file for `NeuralNetworkLayer` the layer types are defined inside the `oneof` layer declaration. But note that you cannot use the `new_layer.layer` field to get its layer type. You always access the thing from the `oneof` directly by its name, such as `activation` or `innerProduct`. The name of the `oneof` itself is only used with `WhichOneof()`.

It should be obvious by now that you really need to have the proto files handy to know what fields and messages you can use here! By the way, to explore what these protobuf objects from the proto files can do, you can use the `dir()` command in the Python REPL:

```
dir(coremltools.proto.Model_pb2.Model)
```

This is not as easy to understand as the actual proto files, but it can be useful to quickly look up what a certain field is called.

To remove a layer, simply delete it. For example, to remove the layer you just added:

```
del spec.neuralNetworkClassifier.layers[-1]
```

This was only a short introduction to the most common things you can do with the Python protobuf API. For the full story, [check out the documentation](#)¹⁵. In the next chapters you'll be using these techniques for all kinds of advanced model surgery.

Saving the new model

Once you're done making your changes to the spec, you can write everything to an mlmodel file again:

```
coremltools.utils.save_spec(spec, "YourNewModel.mlmodel")
```

Tip: Use a different filename. It's really easy to mess something up when you edit the spec by hand and you don't want to overwrite your original mlmodel file!

If you used `model._spec` to get the spec, you can simply save the `MLModel` object:

```
model.save("YourNewModel.mlmodel")
```

Note: this doesn't work when you used `model.get_spec()` since that created a copy.

You can also make a new `MLModel` object from the spec and then save the model:

```
new_model = coremltools.models.MLModel(spec)
new_model.save("YourNewModel.mlmodel")
```

Personally, I prefer to load and save specs directly without using `MLModel` unless I also want to make predictions with the model from Python. Everything else is just as easy with just the spec.

Note: Because the mlmodel format is based on protobuf, you can load, save, and change mlmodel files with any language that supports protobuf. That means you don't necessarily need to use Python to create new mlmodel files — you can even use Swift! See the chapter [Using Protobuf Without coremltools](#) for more details.

¹⁵<https://developers.google.com/protocol-buffers/docs/reference/python-generated>

Part 4: Model Surgery

Model surgery is the act of changing an `mlmodel` after it has been converted. This is useful for cleaning up a model after conversion or to add missing functionality.

Saving the Weights as 16-bit Floats

Neural networks often have millions of learned parameters. When you convert a model to Core ML, these parameters are saved as 32-bit floats or 4 bytes per parameter.

That means a model with 10 million parameters will take up 40 MB in your app bundle. That's 40 additional megabytes that users of your app will have to download. And 10 million parameters isn't really that much yet — there are models with 50 or even 100 million parameters.

With a bit of model surgery it's easy to change the weights and biases to be 16-bit floats instead. This cuts the storage size of your model in half.

Note: This is a feature of `mlmodel` version 2, supported by iOS 11.2 and macOS 10.13.2.

To tell whether a model is using 16-bit weights or the full 32-bit weights, you can look at the `weights` field from any of your layers. The weight values are stored either in `floatValue`, `float16Value`, or `rawValue`. For example, for a convolution layer:

```
def examine_weights(weights):
    if len(weights.floatValue) > 0:
        print("Weights are 32-bit")
    elif len(weights.float16Value) > 0:
        print("Weights are 16-bit")
    elif len(weights.rawValue) > 0:
        print("Weights are quantized or custom layer")
    else:
        print("This layer has no weights")
```

```
nn = get_nn(spec)
examine_weights(nn.layers[i].convolution.weights)
```

To convert the weights for the entire model to 16-bit floats, you can use the function `coremltools.utils.convert_neural_network_spec_weights_to_fp16()`. This works on all neural network-type models and also on pipeline models.

Here is a handy utility script that you can use to convert an existing model to 16-bit:

```
import sys, coremltools

if len(sys.argv) != 3:
    print("USAGE: %s <input_mlmodel> <output_mlmodel>" % sys.argv[0])
    sys.exit(1)

input_model_path = sys.argv[1]
output_model_path = sys.argv[2]

spec = coremltools.utils.load_spec(input_model_path)
spec_fp16 = coremltools.utils.convert_neural_network_spec_weights_to_fp16(spec)
coremltools.utils.save_spec(spec_fp16, output_model_path)
```

Save this script as **convert_to_float16.py** and then invoke it as follows:

```
$ python3 convert_to_float16.py input.mlmodel output.mlmodel
```

I tried this script on a few models from [Apple's developer site](#)¹⁶. Here are the file sizes of the original models and saved with 16-bit floats:

Model name	32-bit weights	16-bit weights
SqueezeNet.mlmodel	4971516	2500504
MobileNet.mlmodel	17136858	8585292
Inceptionv3.mlmodel	94704130	47382082
ResNet50.mlmodel	102586628	51313154

As you can tell from these results, the 16-bit version is indeed 50% of the original size.

Keep in mind that 32-bit or 16-bit is only for the way your weights are stored inside the mlmodel file. What happens during runtime is independent of the storage method! The runtime data type for the weights depends on the hardware that is running the model:

- CPU: uses 32-bit weights, even if the model is saved as 16-bit.
- GPU: uses 16-bit weights, even if the model is saved as 32-bit.
- Neural Engine: who knows?

I think it's a good idea to always save your models using 16-bit floats. You do lose a bit of precision when compared to the full 32-bit numbers the model was originally trained with, but on average these errors cancel out and it shouldn't make the model's predictions any worse.

¹⁶<https://developer.apple.com/machine-learning/models/>

Part 5: Inside the App

Learn how to make the most out of the APIs from the Core ML and Vision frameworks.

The Neural Engine

The first Neural Engine was introduced with the A11 Bionic processor in the iPhone X, but Core ML does not appear to make use of it on those devices (apparently it's just for Face ID).

The A12 and A12X processors have a new, more powerful Neural Engine that can do 5 trillion operations per second. Apple claims that the A12's Neural Engine makes Core ML up to 9 times faster at 1/10th the power. I have indeed seen massive speed boosts on the iPhone XS but only with certain types of models.

Not a lot is known about the Neural Engine, or ANE as it's also called, but here is what we do know:

- Core ML automatically runs your models on the Neural Engine when it can. This frees up the GPU to do other work.
- There is no public API for using the Neural Engine. You can only use it through Core ML. (There are private frameworks but these are obviously not documented and you can't use them in App Store apps.)
- If you use custom layers anywhere in your model, even if only at the beginning or at the very end, the model will not run on the Neural Engine. As of yet, there is no method in `MLCustomLayer` that you can implement to make your custom layer work with the Neural Engine.
- The Neural Engine seems to be optimized for large matrix multiplications. A model such as TinyYOLO, which consists of a small number of heavy convolution layers, is quite slow on the GPU but extremely fast on the Neural Engine. On the other hand, a model such as MobileNetV2 + SSDLite, which has more layers but that do less work, is very fast on the GPU but slower on the Neural Engine.
- Not all layer types are supported. For example, if your model has RNN layers such as LSTM or GRU, it won't run on the Neural Engine. The same is true for dilated convolutions, and possibly other layer types as well.
- Using an `mlmodel` file that stores its weights as 16-bit floats appears to be slower than a model that uses 32-bit floats. On the GPU this makes no difference, as it always converts the weights to 16-bit floats anyway, but the Neural Engine seems to struggle with this. (I am not 100% sure what is going on here.)

It's hard to exactly understand and explain some of these results since there are no profiling tools for the Neural Engine, nor any documentation about how this chip works. (Also, Core ML can be a bit temperamental, making it hard to get consistent measurements.)

In any case, it appears that the design tradeoffs for efficient models are different for the Neural Engine than for the GPU, but figuring them out so far has been mostly trial-and-error.

Tip: To prevent a model from running on the Neural Engine, you can set the `computeUnits` field from `MLModelConfiguration` to `.cpu` or `.cpuAndGPU`. To include the Neural Engine, use `computeUnits = .all`.

Transposing an MLMultiArray

Suppose you have an `MLMultiArray` with the shape (100, 200, 4) but you need to pass this into a function that expects the shape (4, 100, 200). In other words, the last dimension must become the first.

This is a common occurrence during training. NumPy and TensorFlow offer a `transpose()` function for this, while PyTorch calls it `permute()`.

Unlike [reshaping](#), which only changes how you view the dimensions but doesn't change the data, transposing or permuting will literally move around the bytes in the array.

For example, to convert the (100, 200, 4) array into (4, 100, 200) you would call:

```
let transposedArray = multiArray.transpose(to: [2, 0, 1])
```

The notation `[2, 0, 1]` means that dimension 2 moves to the front and the other two dimensions will shift one position towards the back.

Unfortunately, `MLMultiArray` has no API for doing this, so you'll have to write it yourself.

Tip: Rather than doing this in Swift, you can also [add a new layer](#) at the end of the `mlmodel` that rearranges the data. That way Core ML will automatically handle the permutation and you don't have to worry about any of it. The type of layer to add is `PermuteLayerParams`. See `NeuralNetwork.proto` for more details.

To move the last dimension to the front as in the above example, the transpose routine would look like this in pseudocode:

```

for h in 0..

```

You read the value using the source strides and write it using the destination strides. Pretty straightforward. However, this code has a couple of limitations:

- It assumes the array will always have three dimensions.
- It always swaps (h, w, c) to (c, h, w), so it can only transpose one way.

The code for a transpose function that can handle any number of dimensions and any possible permutation order is more complex. Here is one way to write it:

```

extension MLMultiArray {
  @nonobjc public func transposed(to order: [Int]) throws -> MLMultiArray {
    let ndim = order.count
    precondition(ndim == strides.count)

    let newShape = shape.indices.map { shape[order[$0]] }
    let newArray = try MLMultiArray(shape: newShape, dataType: self.dataType)

    let srcPtr = UnsafeMutablePointer<Double>(OpaquePointer(dataPointer))
    let dstPtr = UnsafeMutablePointer<Double>(OpaquePointer(newArray.dataPointer))
  }
}

```

This first swaps around the dimensions so they are in the new order and then creates a MLMultiArray object with that new shape. You also grab the pointers to the memory of both arrays.

Note: This function only works with multi-arrays that have the datatype `.double`. To support `.float32` or `.int32` you need to change the type of the `UnsafeMutablePointer`.

Inside the loop that will copy the values from the original to the new array, you need to compute the source index for reading from and the destination index for writing to. The problem is that you don't know beforehand how many dimensions the array will have, and so you can't use multiple nested loops.

Instead, you'll be using a single loop that treats the multi-arrays as being one-dimensional. You will need to map that scalar index to the different dimensions in both the source and destination array. For that, you first define some variables:

```
let srcShape = shape.map { $0.intValue }
let dstStride = newArray.strides.map { $0.intValue }
var idx = [Int](repeating: 0, count: ndim)
```

The shape and strides are `NSNumber`s, so to gain a bit of speed you turn these into plain `Int` values. The `idx` array is a list of integer indices, one for every dimension. This is how you'll keep track of where you're reading inside the source array. Initially, these values are all 0.

Next up is the actual loop:

```
for j in 0..
```

It's a straight `for` loop that goes from 0 to the number of elements in the array. The source index that you use for reading is simply `j`.

The destination index must be composed of the destination strides and the permuted indices from the `idx` array. This is how the reading position in the source array is mapped to the writing position in the destination array. And then you simply copy the value from the source into the destination.

Finally, you need to update the values of `idx` to move them ahead to the next reading position. There are two ways to do this: 1) divide `j` by the source strides using division and the remainder, 2) simply increment the values in `idx`. Because dividing is relatively slow, you'll choose option number two here:

```

    var i = ndim - 1
    idx[i] += 1
    while i > 0 && idx[i] >= srcShape[i] {
        idx[i] = 0
        idx[i - 1] += 1
        i -= 1
    }
}
return newArray
}
}

```

Let's say the source array has three dimensions of sizes (100, 200, 4). The value of *j* will go from 0 to 80000 because that's how many elements there are (i.e. $100 \times 200 \times 4$).

Initially, *idx* is [0, 0, 0]. Every iteration of the loop you increment the last element of *idx*, so after the first iteration it is [0, 0, 1].

After 4 iterations, *idx* is [0, 0, 4]. But that is one position beyond the length of this last dimension in the source array. Now you roll over that last index and increment the second-to-last index instead, so that *idx* becomes [0, 1, 0].

After another 4 iterations, *idx* will be [0, 2, 0], and so on. It works like a clock, where the minute is incremented when seconds reach 60, and the hour is incremented when minutes reach 60.

And that's it for a basic transpose routine. This code isn't particularly fast, especially for large arrays, but it's useful for debugging and testing things.

Note: For certain special cases you can use the [vImage framework](#) to do very fast permutations of the dimensions. For example, `vImageConvert_PlanarFtoARGBFFFF()` will convert a multi-array of shape (4, height, width) to (height, width, 4). There is also `vImageConvert_PlanarFtoRGBFFFF()` for when you have 3 channels instead of 4. These functions will only work for MLMultiArrays of type `.float32` but they're much faster than writing your own code.

Part 6: Advanced Topics

Become a Core ML pro with these advanced chapters.

Working With Video

When working with `AVCaptureSession` to capture video from the live camera feed, your app receives `CMSampleBuffer` objects in the following delegate method:

```
public func captureOutput(_ output: AVCaptureOutput,
                          didOutput sampleBuffer: CMSampleBuffer,
                          from connection: AVCaptureConnection) {
    if let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer) {
        /* make a prediction using the pixel buffer */
    }
}
```

When working with ARKit to capture live video, your app receives `ARFrame` objects in the following callback:

```
func session(_ session: ARSession, didUpdate frame: ARFrame) {
    if case .normal = frame.camera.trackingState {
        let pixelBuffer = frame.capturedImage
        /* make a prediction using the pixel buffer */
    }
}
```

And when using an `AVAssetReader` to read video frames from a file, your app receives `CMSampleBuffer` objects in a loop:

```
while assetReader.status == .reading {
    autoreleasepool {
        if let sampleBuffer = readerTrackOutput.copyNextSampleBuffer(),
           let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer) {
            /* make a prediction using the pixel buffer */
        }
    }
}
```

What all these methods have in common is that they will receive many `CVPixelBuffer` objects per second. Your app needs to decide how to handle all those pixel buffers.

- If the Core ML or Vision prediction for the previous frame is still running when a new frame arrives, you will need to ignore this new frame in a real-time setting. Your app simply isn't able to keep up with the FPS of the live camera feed.

- When processing frames offline — such as when reading from an mp4 file — you will need to temporarily block reading until the model is ready for the next frame.
- To get higher throughput, you can already schedule the next Core ML or Vision request while the previous one is still running. That's because some part of the model will run on the CPU and some part on the GPU. If the CPU is just sitting there waiting for the GPU to complete, it's more efficient to use that idle time to already prepare the next request. However, you want to limit this to two or three requests at most, or you'll eventually end up with a huge backlog of unprocessed pixel buffers.

Exactly how to handle this situation depends on the demands of your app. This chapter shows a few different approaches.

Note: When using live video, you don't want to hang on to any given pixel buffer for too long. The camera only has a small, fixed pool of buffers that it reuses over and over. If you're holding on to all these buffers, waiting for them to be processed by Core ML at some point in the future, the camera's buffer pool will run out of available buffers and it has nowhere left to store the next frame. That's asking for trouble. You want to recycle the buffers quickly.

Blocking the video thread

The easiest solution is to block the video thread while the model is busy making a prediction. In the delegate method, do the following:

```
public func captureOutput(_ output: AVCaptureOutput,
                        didOutput sampleBuffer: CMSampleBuffer,
                        from connection: AVCaptureConnection) {
    if let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer) {
        let resizedPixelBuffer = ...
        if let output = try? model.prediction(image: resizedPixelBuffer) {
            /* do something with the output */
        }
    }
}
```

Core ML's `model.prediction(...)` method will block the current thread until the result is available. Likewise for the `perform()` method from Vision's `VNImageRequestHandler`.

With this approach, you make the video capture thread wait until Core ML is done. That's fine when the model is really fast and finishes before the next frame arrives from the camera.

But what happens when the model is slower than the camera? In that case, `AVCaptureSession` will not be able to call `captureOutput(_:didOutput:from:)` again on the video capture thread, because that thread is currently occupied with running the Core ML model.

Fortunately, `AVCaptureSession` is smart: if the video capture thread is blocked for too long, `AVCaptureSession` will automatically drop the next frame. When that happens, instead of `captureOutput(_:didOutput:from:)`, the delegate method `captureOutput(_:didDrop:from:)` is called to inform you that your app was too slow.

Note: The setting for this is `alwaysDiscardsLateVideoFrames` from `AVCaptureVideoDataOutput`.

All the logic for dropping the frames happens in `AVCaptureSession` and so you don't have to worry about it. Simply run the model directly on the video capture thread, so that it blocks this thread while Core ML is working. (This appears to be true for ARKit too, which uses `AVCaptureSession` behind the scenes.)

In most apps the camera will run at 30 FPS. If the model takes longer than 1/30th of a second, or 33 ms, then not all video frames can be delivered. Let's say the model takes 50 ms instead. The approach taken here will drop every other frame and `model.prediction(...)` will only be called 15 times per second instead of 30.

This also highlights the drawback of this simple method: a model that takes 50 ms can actually run at 20 FPS instead of 15. This is also true for offline processing with `AVAssetReader` — if you can get this model to process 20 frames per second, it's done in 3/4th the time. To get the maximum amount of throughput, you'll need to be a bit more clever.

Running the model on a separate thread

One way to try and be more clever is to not run the prediction on the video capture thread, but on a separate thread or dispatch queue. You might be tempted to wrap it in a `DispatchQueue.async`:

```

public func captureOutput(_ output: AVCaptureOutput,
                          didOutput sampleBuffer: CMSampleBuffer,
                          from connection: AVCaptureConnection) {
    if let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer) {
        DispatchQueue.global(qos: .userInitiated).async {
            let resizedPixelBuffer = ...
            if let output = try? model.prediction(image: resizedPixelBuffer) {
                /* do something with the output */
            }
        }
    }
}

```

This is a bad idea! Now `model.prediction(...)`, or the `VNImageRequestHandler.perform()` method if you're using Vision, will no longer block the video capture thread. And when this thread is not blocked, `AVCaptureSession` will no longer automatically drop frames when your app is too slow.

For every new frame that arrives from the camera — 30 times per second — this code will schedule another Core ML prediction on the dispatch queue. If the model is slower than 33 ms, you'll quickly build up a huge backlog of unprocessed requests, as Core ML won't be able to keep up.

Eventually, `AVCaptureSession` will start dropping frames anyway. This happens because the global `DispatchQueue` only has so many free threads to run all the Core ML requests. At some point all your CPU cores will be in use and there is no thread that is free to handle the next incoming video frame.

Alternatively, because of the backlog of pending requests, `AVCaptureSession` can eventually run out of `CVPixelBuffer` objects and shut down. Not good. :-)

Another problem with having a backlog is that it increases the *latency* of the requests. Let's say the first 10 frames will all get scheduled on the `DispatchQueue`, then the next 60 frames are dropped because your CPU is completely maxed out and is slowly working its way through that backlog of the first ten frames. By the time you get the results of frame number 10, it's already two seconds later.

If your model isn't fast enough to keep up with the camera, it's better to drop more frames than to get way behind and deliver results for frames that are no longer relevant.

The solution here is to block the video capture thread again, but now manually. The simplest approach is to ignore any new frames — i.e. to drop them by hand — until Core ML is free for work again. First, you need to add a new instance variable:

```
var currentBuffer: CVPixelBuffer?
```

Then in your video capture delegate method do the following:

```
public func captureOutput(_ output: AVCaptureOutput,
                          didOutput sampleBuffer: CMSampleBuffer,
                          from connection: AVCaptureConnection) {
    if currentBuffer == nil,
        let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer) {
        currentBuffer = pixelBuffer
        DispatchQueue.global(qos: .userInitiated).async {
            let resizedPixelBuffer = ...
            if let output = try? model.prediction(image: resizedPixelBuffer) {
                /* do something with the output */
            }
            self.currentBuffer = nil
        }
    }
}
```

The new frame will only be processed by Core ML when `currentBuffer` is `nil`, i.e. when Core ML is not already doing anything. Otherwise, nothing happens.

With this method you still only do a single prediction at a time, but now you run it on a thread of your own choosing. Using a different thread in this manner actually improves the performance by quite a bit!

Using a semaphore

The above method already works quite well, but it's not 100% thread-safe. The `currentBuffer` variable is written to from different threads: the video capture thread but also any threads that belong to the `DispatchQueue`. There's a small but not unlikely chance that these writes might interfere with each other.

You could solve this by locking the `currentBuffer` variable before reading or writing it, but there's another solution that neatly avoids this issue, and that is to use a *semaphore*.

First, create the semaphore object:

```
let semaphore = DispatchSemaphore(value: 1)
```

Then the video capture delegate becomes:

```

public func captureOutput(_ output: AVCaptureOutput,
                          didOutput sampleBuffer: CMSampleBuffer,
                          from connection: AVCaptureConnection) {
    if let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer) {
        semaphore.wait()
        DispatchQueue.global(qos: .userInitiated).async {
            let resizedPixelBuffer = ...
            if let output = try? model.prediction(image: resizedPixelBuffer) {
                /* do something with the output */
            }
            self.semaphore.signal()
        }
    }
}

```

The first time `semaphore.wait()` is called, it doesn't actually wait. It only decrements the semaphore's value from 1 to 0 and the thread continues. You can safely run a Core ML prediction on that frame.

If the next frame arrives before Core ML is finished, `semaphore.wait()` really will wait. It blocks the video thread because now the semaphore's value is 0. `AVCaptureSession` will automatically drop any other frames that arrive in the mean time.

Once Core ML is done, you call `semaphore.signal()` to increment the value of the semaphore from 0 to 1. Now `semaphore.wait()` sees that the semaphore is no longer 0 and wakes up the video capture thread again.

And so on... Using a semaphore is completely thread-safe and a really neat solution, plus with a small modification it allows you to perform multiple Core ML requests at once.

Note: Besides this being a 100% thread-safe solution, there is another small difference with the method from the previous section. There you ignored any new frames while Core ML was still busy. After Core ML has finished, you may have to wait a short while until the next frame arrives (anywhere between 0 and 33 ms).

With the method from this section, you're letting `AVCaptureSession` take care of dropping the frames. When Core ML is finished and the video capture thread gets unblocked, it will immediately start a new prediction — no need to wait until the next frame arrives, because you already have a frame.

However, this is actually an older pixel buffer from a few frames ago, not the most recent one. So that's the trade-off: with the semaphore method you get higher throughput but what Core ML sees may be a little behind on what's happening right now. Naturally, the faster your model, the less of a concern this is.

Multiple requests at a time

The methods described so far were all used to make sure you only perform one Core ML or Vision request at a time, so that you don't build up a large backlog and the video capture session does not run out of buffers.

That's a good idea indeed. However, there is a speed advantage to running multiple requests at a time, as long as you limit this to 2 or 3 simultaneous requests at most.

When Core ML runs a model on the GPU, it also does a bunch of work on the CPU such as encoding all the Metal kernels into command buffers. This can take several milliseconds. But once the job has been handed over to the GPU, the CPU sits idle and waits until the GPU is done.

The CPU and GPU are two parallel processors that can work independently, so it's a waste of time to make the CPU wait on the GPU and vice versa. While the GPU is handling request n , the CPU can already do the preparation work for request $n+1$ and possibly even request $n+2$. Keep 'em busy!

To do this in your app, simply change the value of the semaphore when you initialize it:

```
let semaphore = DispatchSemaphore(value: 2)
```

Now `semaphore.wait()` needs to be called two times before it will block. That means your video capture thread will be able to schedule two separate Core ML requests at the same time. This is no problem for Core ML because internally it uses a serial queue to manage multiple requests.

As soon as the first request is done, `semaphore.signal()` unblocks the video thread again, making room for the next frame to be processed. This way you'll always have two Core ML predictions running concurrently. This scheme is also called *double-buffering*.

Double-buffering will appear to make your model run a little faster, because it overlaps some of the CPU work with the GPU work from a previous frame.

The timeline looks like this:

```
start prediction for frame 1
start prediction for frame 2
block thread for frame 3
drop frame 4
finish prediction for frame 1
start prediction for frame 3
block thread for frame 5
drop frame 6
drop frame 7
finish prediction for frame 2
start prediction for frame 5
...and so on...
```

After the predictions for the first two frames are underway, the video capture thread blocks on frame 3. Any new frames arriving after this will be automatically dropped by `AVCaptureSession`.

As soon as the prediction for frame 1 finishes, the video capture thread resumes. That thread got blocked on frame 3, so that's the frame it makes the prediction for. Note that this is not the most recent frame, as frame 4 got dropped in the mean time.

The next frame that arrives, 5, will block the video capture thread again. The other prediction that's still running at this point is the one for frame 2. Once that is done, frame 5 can have a go, and so on.

Note: If your model is fast enough, no frames will actually get dropped. The above timeline is for a model that is significantly slower than 33 ms per frame.

Why stop at two concurrent predictions, why not three or even more? In practice it turns out you gain the most from having two predictions in parallel. On faster devices it may be worth using a semaphore with value 3, known as *triple-buffering*. (The faster the device, the quicker the model runs but also the more idle time there is to take advantage of.)

There's an important trade-off here: the more requests you schedule at the same time, the longer it will take before you get the results. Look at the above timeline: you don't get the results for frame 2 until about five frames later. With triple-buffering, this time delay — known as the *latency* — only becomes longer.

You get a higher throughput, i.e. the device is able to squeeze as much performance out of the CPU and GPU as possible, but you lose responsiveness.

For offline work, such as when you're using `AVAssetReader` to read the frames from a video file, the latency isn't important. But the law of diminishing returns also applies here: you

won't gain much from doing more than two or three predictions at a time. If your CPU and GPU are both maxed out already, adding more work won't make them go any faster...

The in-flight index: whose turn is it?

When scheduling multiple Core ML requests at a time, there is an important detail you need to take care of. Because every request will run from its own thread, these threads should not be sharing resources.

Suppose you're using Core Image to resize the camera's `CVPixelBuffer` to the dimensions that are expected by the `mlmodel`. Instead of allocating a new destination `CVPixelBuffer` for every frame, it's more efficient to just allocate it once and put it in an instance variable — let's call it `resizedPixelBuffer` — and keep re-using that.

However, it's possible that two different prediction threads will now try to access the `resizedPixelBuffer` variable at the same time. That's a race condition waiting to happen...

The solution is to allocate one of these `resizedPixelBuffer` objects for each possible Core ML request. You also need to write code to cycle through these buffers. First, add some instance variables:

```
let maxInflightBuffers = 2      // or 3
let semaphore: DispatchSemaphore
var inflightBuffer = 0
var resizedPixelBuffers: [CVPixelBuffer] = []
```

Initialize them like so:

```
semaphore = DispatchSemaphore(value: maxInflightBuffers)

for _ in 0..
```

Then change the video capture delegate method to:

```

public func captureOutput(_ output: AVCaptureOutput,
                          didOutput sampleBuffer: CMSampleBuffer,
                          from connection: AVCaptureConnection) {
    if let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer) {
        semaphore.wait()

        let inflightIndex = inflightBuffer
        inflightBuffer += 1
        if inflightBuffer >= maxInflightBuffers {
            inflightBuffer = 0
        }

        DispatchQueue.global(qos: .userInitiated).async {
            let resizedPixelBuffer = self.resizedPixelBuffers[inflightIndex]
            /* actually resize the pixel buffer */

            if let output = try? model.prediction(image: resizedPixelBuffer) {
                /* do something with the output */
            }
            self.semaphore.signal()
        }
    }
}

```

Now `inflightIndex` will cycle between 0 and `maxInflightBuffers`. With `maxInflightBuffers` = 2, there will be at most two Core ML predictions happening in parallel, each in its own thread. One of these will always have `inflightIndex` = 0 and the other will have `inflightIndex` = 1. Each thread is therefore always reading from and writing to its own `resizedPixelBuffer` object.

Note: The code that increments `inflightIndex` must always be run from the same thread, in this case the video capture thread. Don't put this inside the `DispatchQueue` or it's no longer thread-safe.

Vision requests are not thread-safe!

In the previous examples I've only shown the code for using Core ML directly. When using double or triple-buffering, you'll need to treat the Vision request object as a resource that should not be shared between threads too.

It's OK to call Core ML's `model.prediction(...)` method from multiple threads, but a Vision request object should only be used by one thread at a time.

So if you're planning to use Vision in combination with the semaphore, you need to do allocate as many Vision request objects as your `maxInflightBuffers`:

```
var visionRequests = [VNCoreMLRequest]()

for _ in 0..maxInflightBuffers {
    let request = VNCoreMLRequest(model: ...) { request, error in
        /* handle the results */
    }
    requests.append(request)
}
```

It's OK for all these `VNCoreMLRequest` objects to have the same completion handler. (I usually put it in a separate function.)

Now the code in the camera capture delegate will be slightly different:

```
public func captureOutput(_ output: AVCaptureOutput,
                          didOutput sampleBuffer: CMSampleBuffer,
                          from connection: AVCaptureConnection) {
    if let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer) {
        semaphore.wait()

        let inflightIndex = inflightBuffer
        inflightBuffer += 1
        if inflightBuffer >= maxInflightBuffers {
            inflightBuffer = 0
        }

        DispatchQueue.global(qos: .userInitiated).async {
            let handler = VNImageRequestHandler(cvPixelBuffer: pixelBuffer)
            let request = self.requests[inflightIndex]
            try? handler.perform([request])
            self.semaphore.signal()
        }
    }
}
```

Note that you can also call `semaphore.signal()` from the `VNCoreMLRequest` completion handler.

Is this really a good idea?

Having multiple predictions in progress at once really improves the throughput of the model, because now the CPU and GPU are as busy as they can be. However, on an iPhone

that is not always the best idea. You may need to use the GPU for drawing as well, especially if you're doing things with AR.

Maxing out the GPU for a long stretch of time will also make it hot, and eventually the GPU will throttle down to run at a lower clock speed.

These high throughput techniques certainly have a place, but use them judiciously.

Even if your model is fast enough to run at 30 FPS or faster, I would encourage you to find ways to run the Core ML model as little as possible. Keep the camera at 30 FPS but add some logic to `captureOutput(_:didOutput:from:)` to skip every other frame, for example. This will seriously improve the energy usage of your app and keeps users' batteries charged for longer.

For the best user experience, my advice is to run the model at the lowest possible frame rate you can possibly get away with. For example, if the user is waving the camera around too much then pause the predictions and wait until they've stopped moving before running the model again.

How big is my preview?

If you're displaying the live video feed from the camera using `AVCaptureVideoPreviewLayer`, the video may be letterboxed on the sides or on the top and bottom. This depends on the screen size of the device and the preview layer's `videoGravity` setting.

Sometimes it's useful to know what the screen coordinates are of the actual video, for example to draw something on top, such as a detected bounding box or a segmentation mask. You can find these screen coordinates with the following code snippet:

```
let videoPreviewRect = previewLayer.layerRectConverted(fromMetadataOutputRect:
    CGRect(x: 0, y: 0, width: 1, height: 1))
```