Chapter 10. Machine Learning Interoperability

By Alfredo Deza

> *Mammalian brains have considerable power for generalized computation but special functions (e.g., subjectivity) commonly require specialized structures. Such a hypothesized structure has been facetiously termed a "subjectivity pump" by Marcel Kinsbourne. Well, that is exactly what some of us are looking for. And the mechanism for subjectivity is double, as shown by the duality of the anatomy, the success of hemispherectomy, and the split-brain results (in cats and monkeys as well as humans).*
>
> —Dr. Joseph Bogen

Peru has *several thousand* potato varieties. As someone who grew up in Peru, I find this surprising to hear. It is easy to assume that most potatoes taste somewhat similar, but this is not the case at all. Different dishes call for different potato varieties. You won't want to argue with a Peruvian cook if the recipe called for *Huayro* potatoes and you want to use the common *baking* potato found in most of the US. If you ever have the chance to be in South America (and certainly in Peru), try the experience of walking around a street market. The number of fresh vegetables, including several dozen potato varieties, can make you dizzy.

I no longer live in Peru, and I miss that variety of potatoes. I can't really cook some dishes and make them taste the same with a common baking potato bought from the local supermarket. It is not the same. I consider that the variety and different tastes provided by such a humble vegetable is critical for Peru's cuisine identity. You still might find it OK to cook with the common baking potato, but at the end of the day it is about choices and selections.

There is empowerment in variety and the ability to pick and choose what fits better. This empowerment is also true in machine learning when dealing with the end product: the trained model.

Trained machine learning models also come with distinct constraints, and most of them aren't going to work in an environment that isn't specifically tailored to support it. The main concept of model interoperability is to be able to *transform* a model from one platform to another, which creates choices. This chapter makes the case that, although there are sound arguments to use out-of-the-box models from a cloud provider and not worry much about vendor lock-in, it is essential to understand how to export models into other formats that can work in other platforms that have different constraints. Much like how containers can work seamlessly in most any system as long as there is an underlying container runtime (as I explain in "Containers"), model interoperability or having a model exported to different formats is key to flexibility and empowerment. Figure 10-1 demonstrates a rough overview of what this interoperability means by training in any ML framework, transforming the resulting model *once*, to deploy it almost anywhere: from edge devices to mobile phones and other operating systems.

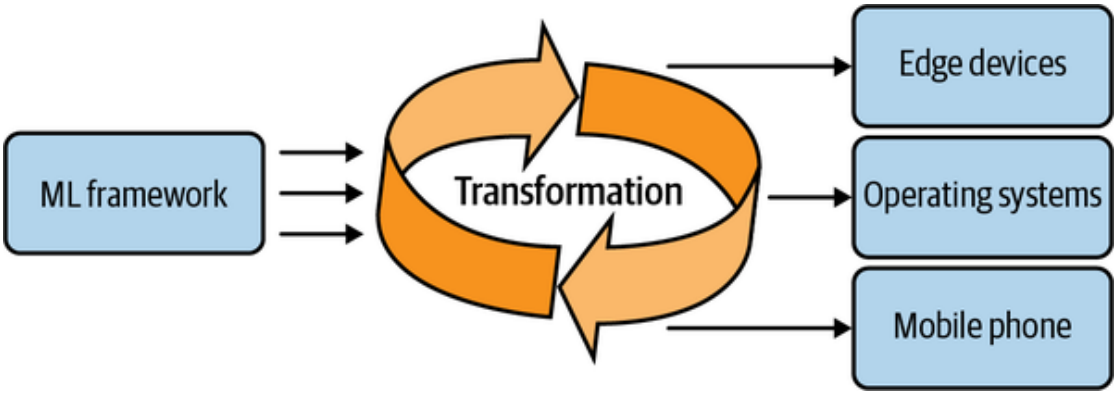

Figure 10-1. Interoperability overview

This situation is like producing screws that can work with any screwdriver, instead of producing screws that can only work with screwdrivers from a single home improvement store. In "Edge Devices", we already ran into problems when a model couldn't work with the Edge TPU, and conversion was ultimately necessary. Currently, there are some exciting options, and I'm particularly surprised by ONNX, a community-driven project with open standards that wants to make it easier to interact with

models by reducing complexity from toolchains. This chapter will go into some of the details that make ONNX a compelling machine learning choice and how most clouds are already supporting this format.

Why Interoperability Is Critical

Abstracting complex processes and interactions is a typical pattern in software engineering. Sometimes the abstractions can get entirely out of hand, causing the abstraction to be as complex as the underlying software it was trying to abstract. An excellent example of this is Openstack (an open source infrastructure-as-service platform) and its installer. Installing and configuring an infrastructure platform can be very complicated. Different machine types and network topologies create a tricky combination to solve with a one-size-fits-all installer.

A new installer was created to make it easier to install TripleO (Openstack On Openstack). TripleO produced a temporary instance that, in turn, would install Openstack. The project solved many problems associated with installation and configuration, but some thought it was still complicated, and there was a need to abstract further. This is how QuintupleO got created (Openstack On Openstack On Openstack). Without being actively involved in Openstack, I can tell you that it is difficult to deploy and that engineering teams are trying to solve those problems generically. But I'm doubtful that adding another layer is the solution.

It is easy to get persuaded that another layer will make things easier, but in reality, this is very hard to do well while pleasing everyone. I have an open question I often use to design systems: *the system can be very simple and opinionated, or flexible and complex. Which one do you choose?* Nobody likes these options, and everyone pushes for *simple and flexible*. It is possible to create such a system, but it is challenging to attain.

In machine learning, multiple platforms and cloud providers train models in different and particular ways. This doesn't matter much if staying within the platform and how it interacts with the model, but it is a recipe for frustration if you ever need to get that trained model running elsewhere.

While training a dataset with AutoML on Azure recently, I encountered several problems while trying to get local inferencing working. Azure has a "no-code" deployment with AutoML, and several types of trained models support this type of deployment. This means there is no need to write a single line of code to create the inferencing and serve the responses. Azure handles the API documentation that explains what the inputs and expected outputs are. I couldn't find the *scoring script* for the trained model, and I couldn't find any hints about the scoring script that would help me run it locally. There is no obvious way to understand how to load and interact with the model.

The model's suffix implied it is using Python's `pickle` module, so after trying a few different things, I managed to get it loaded but couldn't do any inferencing on it. I had to deal with the dependencies next. AutoML in Azure doesn't advertise the exact versions and libraries used to train a model. I'm currently using Python 3.8 but wasn't able to install the Azure SDK in my system because the SDK only supports 3.7. I had to install Python 3.7, then create a virtual environment and install the SDK there.

One of the libraries (*xgboost*) had a non–backward compatibility for its latest versions (modules were moved or renamed), so I had to guess one that would allow a specific import. That turned out to be 0.90 from 2019. Finally, when training a model with AutoML in Azure, it seems to use the latest version of the Azure SDK *at the time the model gets trained*. But this isn't advertised either. That is, if a model is trained in January, and you are trying to use it a month later with a few SDK releases in between, you can't use the latest SDK release. You have to go back and find the latest release of the SDK when Azure trained the model.

By no means is this situation meant to be an overly critical description of Azure's AutoML. The platform is excellent to use and could improve by better advertising what versions are used and how to interact with a model locally. The prevalent problem is that you lose control granularity in the process: low code or no code is excellent for speed but can get complicated for portability. I faced all these problems by trying to do local inferencing, but the same could happen if you are leveraging Azure for machine learning in general, but your company is using AWS for hosting.

Another problematic situation that happens often is that scientists that produce a model in one platform have to make assumptions, such as the underlying environment, which includes compute power, storage, and memory. What happens when a model that performs well in AWS needs to deploy in an edge TPU device that is entirely incompatible? Let's assume for a second that your company has this situation covered already and produces the same model targeting different platforms. Still, the resulting model for the edge device is five gigabytes, which is beyond the accelerator's maximum storage capacity.

Model interoperability solves these problems by openly describing the constraints, making it easier to *transform models* from one format to the other while enjoying support from all the prominent cloud providers. In the next section, I'll go into the details of what makes ONNX a solid project for greater interoperability and how you can build automation to transform models effortlessly.

ONNX: Open Neural Network Exchange

As I've mentioned before, ONNX is not only a great choice for model interoperability but also the first initiative toward a system that would allow switching frameworks with ease. The project started in 2017 when both Facebook and Microsoft presented ONNX as an open ecosystem for AI model interoperability and jointly developed the project and tooling to push forward the adoption. Since then, the project has grown and matured as a large open source project with an established structure that includes SIGs (Special Interest Groups) and working groups for different areas like releases and training.

Beyond interoperability, the commonality of the framework allows hardware vendors to target ONNX and impact multiple other frameworks at once. By leveraging the ONNX representation, the optimizations are no longer required to be integrated individually into each framework (a time-consuming process). Although ONNX is relatively recent, it is refreshing to see it well supported across cloud providers. It shouldn't be

surprising that Azure even offers native support for ONNX models in its machine learning SDK.

The main idea is to train once in your preferred framework and run anywhere: from the cloud to the edge. Once the model is in the ONNX format, you can deploy it to various devices and platforms. This includes different operating systems as well. The effort to make this happen is substantial. Not many software examples come to mind that can run in several different operating systems, edge devices, and the cloud, all using the same format.

Although there are several machine learning frameworks supported (more being added all the time), Figure 10-2 shows the most common transformation pattern.

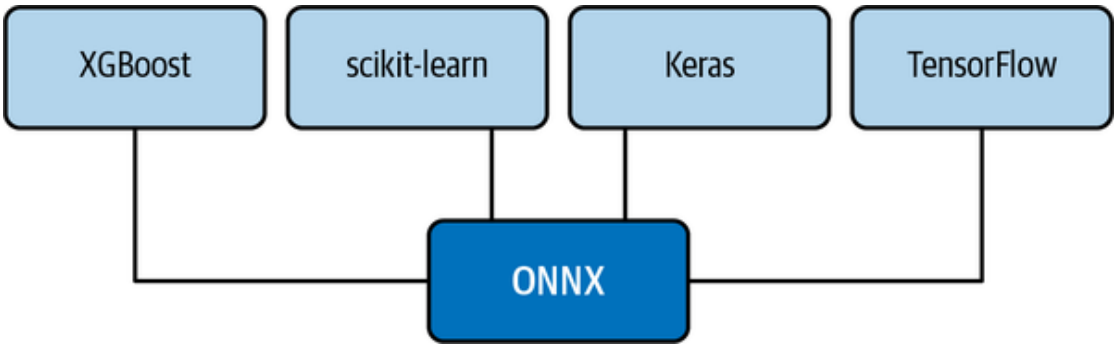

Figure 10-2. Convert into ONNX

You leverage the knowledge and features of the framework you like best and then transform into ONNX. However, as I demonstrate in "Apple Core ML", it is also possible (although not that common) to convert an ONNX model into a different runtime as well. These transformations are not "free" either: you can get into problems when new features aren't supported yet (ONNX converters are always catching up), or older models aren't supported with newer versions. I still believe that the idea of commonality and "run anywhere" is a solid one, and it is helpful to take advantage of it when possible.

Next, let's see where you can find pretrained ONNX models, to try some of them out.

ONNX Model Zoo

The *Model Zoo* is often referenced when discussing ONNX models. Although it is usually described as a registry for ready-to-use ONNX models, it is primarily an informational repository in GitHub that has links to several pretrained models contributed by the community and curated in the repository. The models are separated into three categories: vision, language, and everything else. If you are looking to get started with ONNX and do some inferencing, the Model Zoo is the place to go.

In "Packaging for ML Models" I used the *RoBERTa-SequenceClassification* model from the Model Zoo. Since I wanted to register in Azure, I was required to add some information like the ONNX runtime version. Figure 10-3 shows how this is all available at the Model Zoo for that particular model.

Model

Model	Download	Download (with sample test data)	ONNX version	Opset version	Accuracy
RoBERTa-BASE	499 MB	295 MB	1.6	11	88.5
RoBERTa-SequenceClassification	499 MB	432 MB	1.6	9	MCC of 0.85

Figure 10-3. Model Zoo

Aside from version and size information, the page will usually have some examples on how to interact with the model, which is crucial if you want to create a proof of concept to try it out quickly. There is one other thing that I believe is worth noting about these documentation pages: getting a *provenance* (a source of truth for the model). In the case of the *RoBERTa-SequenceClassification* model, it originates from *PyTorch RoBERTa*, then to ONNX, and finally made available in the Model Zoo.

It isn't immediately clear why knowing the origin of models and sources of work is essential. Whenever changes are needed or problems are detected that need to be solved, you had better be ready to pinpoint the source of truth so that anything that needs to be modified can be done with confidence. When I was a release manager for a large open source project, I was responsible for building RPM and other package types for different Linux distributions. One day the production repository got corrupted, and I was asked to rebuild these packages. While rebuilding them,

I couldn't find what script, pipeline, or CI platform was producing one of these packages included in several dozen of these repositories.

After tracing the various steps involved in finding the origin of that one package, I found that a script was downloading it from a developer's home directory (long gone from the company) in a server that had nothing to do with building packages. A single file sitting in a home directory in a server that had no responsibility for the build system is a ticking bomb. I couldn't tell what the origin of the package was, how to make any updates to it, or the reason it needed to be included in this way. These situations are not uncommon. You must be prepared to have everything in order and have a solid answer when determining the source of truth of all the elements in your production pipeline.

When you are sourcing models from places like the Model Zoo, make sure you capture as much information as possible and include it wherever the destination is for these models. Azure has several fields you can use for this purpose when registering models. As you will see in the following sections, some of the model transformers allow adding metadata. Taking advantage of this seemingly unimportant task can be critical for debugging production problems. Two beneficial practices that have reduced time debugging and accelerated onboarding and maintenance ease is using meaningful names and as much metadata as possible. Using a meaningful name is crucial for identification and providing clarity. A model registered as "production-model-1" does not tell me what it is or what it is about. If you pair this with no additional metadata or information, this will cause frustration and delays when figuring out a production problem.

Convert PyTorch into ONNX

Getting started with a different framework is always daunting, even when the underlying task is to train a model from a dataset. PyTorch does an excellent job at including pretrained models that can quickly help you get started since you can try different aspects of the framework without dealing with curating a dataset and then figuring out how to train it. Many other frameworks (like TensorFlow and scikit-learn) are doing the

same, and it is an excellent way to jumpstart learning. In this section, I use a pretrained vision model from PyTorch and then export it to ONNX.

Create a new virtual environment and a *requirements.txt* file that looks like the following:

```
numpy==1.20.1
onnx==1.8.1
Pillow==8.1.2
protobuf==3.15.6
six==1.15.0
torch==1.8.0
torchvision==0.9.0
typing-extensions==3.7.4.3
```

Install the dependencies and then create a *convert.py* file to produce the PyTorch model first:

```
import torch
import torchvision

dummy_tensor = torch.randn(8, 3, 200, 200)

model = torchvision.models.resnet18(pretrained=True)

input_names = [ "input_%d" % i for i in range(12) ]
output_names = [ "output_1" ]

torch.onnx.export(
    model,
    dummy_tensor,
    "resnet18.onnx",
    input_names=input_names,
    output_names=output_names,
    opset_version=7,
    verbose=True,
)
```

Let's go through some of the steps that the Python script goes through to produce an ONNX model. It creates a tensor filled with random numbers

using three channels (crucial for the pretrained model). Next, we retrieve the *resnet18* pretrained model available through the *torchvision* library. I define a few inputs and outputs and finally export the model with all that information.

The example is overly simplistic to prove a point. The exported model is not robust at all and full of dummy values that aren't meaningful. The idea is to demonstrate how PyTorch enables you to export the model to ONNX in a straightforward way. The fact that the converter is part of the framework is reassuring because it is responsible for ensuring that this works flawlessly. Although separate converter libraries and projects exist, I prefer frameworks that offer the conversion, like PyTorch.

NOTE

The `opset_version` argument in the `export()` function is critical. PyTorch's tensor indexing can get you into problems with an unsupported ONNX opset version. Some indexer types do not support anything other than version 12 (the latest version). Always double-check that the versions match the supported features you need.

Run the *convert.py* script, which creates a *resnet18.onnx* file. You should see output similar to this:

```
$ python convert.py
graph(%learned_0 : Float(8, 3, 200, 200, strides=[120000, 40000, 200, 1],
  requires_grad=0, device=cpu),
  %fc.weight : Float(1000, 512, strides=[512, 1], requires_grad=1, device=cpu)
  %fc.bias : Float(1000, strides=[1], requires_grad=1, device=cpu),
  %193 : Float(64, 3, 7, 7, strides=[147, 49, 7, 1], requires_grad=0,
```

Now that the ONNX model is available and produced by the script using PyTorch, let's use the ONNX framework to verify that the model produced is compatible. Create a new script called *check.py*:

```
import onnx

# Load the previously created ONNX model
```

```
model = onnx.load("resnet18.onnx")

onnx.checker.check_model(model)

print(onnx.helper.printable_graph(model.graph))
```

Run the *check.py* script from the same directory where *resnet18.onnx* is, and verify that the output is similar to this:

```
$ python check.py
graph torch-jit-export (
  %learned_0[FLOAT, 8x3x200x200]
) optional inputs with matching initializers (
  %fc.weight[FLOAT, 1000x512]
[...]
  %189 = GlobalAveragePool(%188)
  %190 = Flatten[axis = 1](%189)
  %output_1 = Gemm[alpha = 1, beta = 1, transB = 1](%190, %fc.weight, %fc.bias
  return %output_1
}
```

The verification from the call to the `check_model()` function should not produce any errors, proving that the conversion has some level of correctness. To fully ensure that the converted model is correct, inferencing needs to be evaluated, capturing any possible drift. If you are unsure about what metrics to use or how to create a solid comparison strategy, check the "Basics of Model Monitoring". Next, let's see how we can use the same checking pattern in a command line tool.

Create a Generic ONNX Checker

Now that I've gone through the details of exporting a model from PyTorch to ONNX and having a step to verify, let's create a simple and generic tool that can verify any ONNX model, not just one in particular. Although we dedicate a large section about building powerful command line tools in the next chapter (see "Command Line Tools" in particular), we can still try to build something that works well for this use case. Another concept that comes from DevOps and my experience as a Systems Administrator is to attempt automation whenever possible and start with the most

straightforward problem first. For this example, I will not use any command line tool framework or any advanced parsing.

First, create a new file called *onnx-checker.py* with a single `main()` function:

```
def main():
    help_menu = """
    A command line tool to quickly verify ONNX models using
    check_model()
    """
    print(help_menu)


if __name__ == '__main__':
    main()
```

Run the script, and the output should show the help menu:

```
$ python onnx-checker.py

    A command line tool to quickly verify ONNX models using
    check_model()
```

The script is not doing anything special yet. It uses the `main()` function to produce the help menu and a widely used crutch in Python to call a specific function when Python executes the script in the terminal. Next, we need to handle arbitrary input. Command line tool frameworks help with this, no doubt, but we can still have something valuable with minimal effort. To check the script's arguments (we will need those to know what model to check), we need to use the `sys.argv` module. Update the script, so it imports the module and passes it into the function:

```
import sys

def main(arguments):
    help_menu = """
    A command line tool to quickly verify ONNX models using
    check_model()
```

```
        """

        if "--help" in arguments:
            print(help_menu)

if __name__ == '__main__':
    main(sys.argv)
```

The change will cause the script to output the help menu only when using the `--help` flag. The script is still not doing anything useful, so let's update the `main()` function once more to include the ONNX check functionality:

```
import sys
import onnx

def main(arguments):
    help_menu = """
    A command line tool to quickly verify ONNX models using
    check_model()
    """

    if "--help" in arguments:
        print(help_menu)
        sys.exit(0)

    model = onnx.load(arguments[-1])
    onnx.checker.check_model(model)
    print(onnx.helper.printable_graph(model.graph))
```

There are two crucial changes to the function. First, it is now calling `sys.exit(0)` after the help menu check to prevent the next block of code from executing. Next, if the help condition is not met, it uses the last argument (whatever that is) as the model's path to check. Finally, it uses the same functions from the ONNX framework to the model check. Note that there is no sanitation or verification of inputs at all. This is a very brittle script, but it still proves helpful if you run it:

```
$ python onnx-checker.py ~/Downloads/roberta-base-11.onnx
graph torch-jit-export (
```

```
    %input_ids[INT64, batch_sizexseq_len]
  ) initializers (
    %1621[FLOAT, 768x768]
    %1622[FLOAT, 768x768]
    %1623[FLOAT, 768x768]
  [...]
    %output_2 = Tanh(%1619)
    return %output_1, %output_2
  }
```

The path I used is to the RoBERTa base model that is in a separate path in my *Downloads* directory. This type of automation is a building brick: the simplest approach possible to do a quick check that can be leveraged later in other automation like a CI/CD system or a pipeline in a cloud provider workflow. Now that we've tried some models, let's see how to convert models created in other popular frameworks into ONNX.

Convert TensorFlow into ONNX

There is a project dedicated to doing model conversions from TensorFlow in the ONNX GitHub repository. It offers a wide range of supported versions from both ONNX and TensorFlow. Again, it is critical to ensure that whatever tooling you choose has the versions your model requires to achieve a successful conversion to ONNX.

Finding the right project, library, or tool for doing conversions can get tricky. For TensorFlow specifically, you can use the onnxmltools, which has a `onnxmltools.convert_tensorflow()` function, or the tensorflow-onnx project, which has two ways to do a conversion: with a command line tool, or using the library.

This section uses the *tensorflow-onnx* project with a Python module that you can use as a command line tool. The project allows you to convert models from both TensorFlow major versions (1 and 2) as well as *tflite*, and *tf.keras*. The broad ONNX opset support is excellent (from version 7 to 13) because it allows greater flexibility when planning a conversion strategy.

Before getting into the actual conversion, it is worth exploring how to invoke the converter. The *tf2onnx* project uses a Python shortcut to expose a command line tool from a file instead of packaging a command line tool with the project. This means that the invocation requires you to use the Python executable with a special flag. Start by installing the library in a new virtual environment. Create a *requirements.txt* file to ensure that all the suitable versions for this example will work:

```
certifi==2020.12.5
chardet==4.0.0
flatbuffers==1.12
idna==2.10
numpy==1.20.1
onnx==1.8.1
protobuf==3.15.6
requests==2.25.1
six==1.15.0
tf2onnx==1.8.4
typing-extensions==3.7.4.3
urllib3==1.26.4
tensorflow==2.4.1
```

Now use `pip` to install all the dependencies at their pinned versions:

```
$ pip install -r requirements.txt
Collecting tf2onnx
[...]
Installing collected packages: six, protobuf, numpy, typing-extensions,
onnx, certifi, chardet, idna, urllib3, requests, flatbuffers, tf2onnx
Successfully installed numpy-1.20.1 onnx-1.8.1 tf2onnx-1.8.4 ...
```

NOTE

If you install the *tf2onnx* project without the *requirements.txt* file, the tool will not work because it doesn't list `tensorflow` as a requirement. For the examples in this section, I'm using `tensorflow` at version 2.4.1. Make sure you install it to prevent dependency problems.

Run the help menu to check what is available. Remember, the invocation looks somewhat unconventional because it needs the Python executable to use it:

```
$ python -m tf2onnx.convert --help
usage: convert.py [...]

Convert tensorflow graphs to ONNX.

[...]

Usage Examples:

python -m tf2onnx.convert --saved-model saved_model_dir --output model.onnx
python -m tf2onnx.convert --input frozen_graph.pb  --inputs X:0 \
  --outputs output:0 --output model.onnx
python -m tf2onnx.convert --checkpoint checkpoint.meta  --inputs X:0 \
  --outputs output:0 --output model.onnx
```

I'm omitting a few sections of the help menu for brevity. Calling the help menu is a reliable way to ensure that the library can load after installation. This wouldn't be possible if `tensorflow` is not installed, for example. I left the three examples from the help menu because those are the ones you will need, depending on the type of conversion you are performing. None of these conversions are straightforward unless you have a good understanding of the internals of the model you are trying to convert. Let's start with a conversion that requires no knowledge of the model, allowing the conversion to work out-of-the-box.

First, download the ssd_mobilenet_v2 model (compressed in a *tar.gz* file) from *tfhub*. Then create a directory and uncompress it there:

```
$ mkdir ssd
$ cd ssd
$ mv ~/Downloads/ssd_mobilenet_v2_2.tar.gz .
$ tar xzvf ssd_mobilenet_v2_2.tar.gz
x ./
x ./saved_model.pb
x ./variables/
```

```
x ./variables/variables.data-00000-of-00001
x ./variables/variables.index
```

Now that the decompressed model is in a directory use the *tf2onnx* conversion tool to port *ssd_mobilenet* over to ONNX. Make sure you are using an opset of 13 to prevent incompatible features of the model. This is a shortened exception traceback you can experience when specifying an unsupported opset:

```
File "/.../.../tf2onnx/tfonnx.py", line 294, in tensorflow_onnx_mapping
  func(g, node, **kwargs, initialized_tables=initialized_tables, ...)
File "/.../.../tf2onnx/onnx_opset/tensor.py", line 1130, in version_1
  k = node.inputs[1].get_tensor_value()
File "/.../.../tf2onnx/graph.py", line 317, in get_tensor_value
  raise ValueError("get tensor value: '{}' must be Const".format(self.name))
ValueError: get tensor value:
  'StatefulPartitionedCall/.../SortByField/strided_slice__1738' must be Const
```

Use the `--saved-model` flag with the path to where the model got extracted to get a conversion working finally. In this case, I'm using opset 13:

```
$ python -m tf2onnx.convert --opset 13 \
  --saved-model /Users/alfredo/models/ssd --output ssd.onnx
2021-03-24 - WARNING - '--tag' not specified for saved_model. Using --tag serve
2021-03-24 - INFO - Signatures found in model: [serving_default].
2021-03-24 - INFO - Using tensorflow=2.4.1, onnx=1.8.1, tf2onnx=1.8.4/cd55bf
2021-03-24 - INFO - Using opset <onnx, 13>
2021-03-24 - INFO - Computed 2 values for constant folding
2021-03-24 - INFO - folding node using tf type=Select,
  name=StatefulPartitionedCall/Postprocessor/.../Select_1
2021-03-24 - INFO - folding node using tf type=Select,
  name=StatefulPartitionedCall/Postprocessor/.../Select_8
2021-03-24 - INFO - Optimizing ONNX model
2021-03-24 - INFO - After optimization: BatchNormalization -53 (60->7), ...
  Successfully converted TensorFlow model /Users/alfredo/models/ssd to ONNX
2021-03-24 - INFO - Model inputs: ['input_tensor:0']
2021-03-24 - INFO - Model outputs: ['detection_anchor_indices', ...]
2021-03-24 - INFO - ONNX model is saved at ssd.onnx
```

These examples might look overly simplistic, but the idea here is that these are building blocks so that you can explore further automation by knowing what is possible in conversions. Now that I have demonstrated what is needed to convert a TensorFlow model, let's see what is required to convert a *tflite* one, another of the supported types from *tf2onnx*.

Download a *tflite* quantized version of the *mobilenet* model from tfhub. The *tflite* support in *tf2onnx* makes the invocation slightly different. This is one of those cases where a tool gets created following one criterion (convert TensorFlow models to ONNX) and then has to pivot support for something else that doesn't quite fit the same pattern. In this case, you must use the `--tflite` flag, which should point to the downloaded file:

```
$ python -m tf2onnx.convert \
  --tflite ~/Downloads/mobilenet_v2_1.0_224_quant.tflite \
  --output mobilenet_v2_1.0_224_quant.onnx
```

I quickly get into trouble once again when running the command because the supported opset doesn't match the default. Further, this model is quantized, which is another layer that the converter has to resolve. Here is another short traceback excerpt from trying that out:

```
    File "/.../.../tf2onnx/tfonnx.py", line 294, in tensorflow_onnx_mapping
      func(g, node, **kwargs, initialized_tables=initialized_tables, dequantize)
    File "/.../.../tf2onnx/tflite_handlers/tfl_math.py", line 96, in version_1
      raise ValueError
ValueError: \
  Opset 10 is required for quantization.
  Consider using the --dequantize flag or --opset 10.
```

At least this time the error is hinting that the model is quantized and that I should consider using a different opset (versus the default opset, which clearly doesn't work).

TIP

Different TensorFlow ops have varying support for ONNX and can sometimes create problems if the incorrect version is used. The tf2onnx Support Status page can be useful when trying to determine what is the correct version to use.

I'm usually very suspicious when a book or demo shows perfection all the time. There is tremendous value in tracebacks, errors, and getting into trouble—which is happening for me when trying to get *tf2onnx* to work properly. If the examples in this chapter show you how it all "just works," you will undoubtedly think that there is a significant knowledge gap or that the tooling is failing, giving no opportunity to understand why things aren't quite working out. I add these tracebacks and errors because *tf2onnx* has a higher degree of complexity that allows me to get into a broken state without much effort.

Let's fix the invocation and give it an opset of 13 (the highest supported offset at the moment), and try again:

```
$ python -m tf2onnx.convert --opset 13 \
  --tflite ~/Downloads/mobilenet_v2_1.0_224_quant.tflite \
  --output mobilenet_v2_1.0_224_quant.onnx

2021-03-23 INFO - Using tensorflow=2.4.1, onnx=1.8.1, tf2onnx=1.8.4/cd55bf
2021-03-23 INFO - Using opset <onnx, 13>
2021-03-23 INFO - Optimizing ONNX model
2021-03-23 INFO - After optimization: Cast -1 (1->0), Const -307 (596->289)...
2021-03-23 INFO - Successfully converted TensorFlow model \
                  ~/Downloads/mobilenet_v2_1.0_224_quant.tflite to ONNX
2021-03-23 INFO - Model inputs: ['input']
2021-03-23 INFO - Model outputs: ['output']
2021-03-23 INFO - ONNX model is saved at mobilenet_v2_1.0_224_quant.onnx
```

At last, the quantized *tflite* model gets converted to ONNX. There is room for improvement, and like we've seen in the previous steps throughout this section, it is invaluable to have a good grasp of the model inputs and outputs and how the model was created. This know-how is crucial at the time of conversion, where you can provide the tooling as much informa-

tion as possible to secure a successful outcome. Now that I've converted some models to ONNX, let's see how to deploy them with Azure.

Deploy ONNX to Azure

Azure has very good ONNX integration in its platform with direct support in its Python SDK. You can create an experiment to train a model using PyTorch that can then export to ONNX, and as I'll show you in this section, you can deploy that ONNX model to a cluster for live inferencing. This section will not cover how to perform the actual training of the model; however, I'll explain how straightforward it is to use a trained ONNX model that is registered in Azure and deploy it to a cluster.

In "Packaging for ML Models" I covered all the details you need to get a model into Azure and then package it in a container. Let's reuse some of the container code to create the *scoring file*, which Azure needs to deploy it as a web service. In essence, it is the same thing: the script receives a request, which knows how to translate the inputs to make a prediction with the loaded model and then return the values.

NOTE

These examples uses Azure's *workspace*, defined as a `ws` object. It is necessary to configure it before starting. This is covered in detail in "Azure CLI and Python SDK".

Create the scoring file, call it *score.py*, and add an `init()` function to load the model:

```
import torch
import os
import numpy as np
from transformers import RobertaTokenizer
import onnxruntime


def init():
    global session
```

```
    model = os.path.join(
       os.getenv("AZUREML_MODEL_DIR"), "roberta-sequence-classification-9.onnx
    )
    session = onnxruntime.InferenceSession(model)
```

Now that the scoring script's basics are out of the way, a `run()` function is required when running in Azure. Update the *score.py* script by creating the `run()` function that understands how to interact with the RoBERTa classification model:

```
def run(input_data_json):
    try:
        tokenizer = RobertaTokenizer.from_pretrained("roberta-base")
        input_ids = torch.tensor(
            tokenizer.encode(input_data_json[0], add_special_tokens=True)
        ).unsqueeze(0)

        if input_ids.requires_grad:
            numpy_func = input_ids.detach().cpu().numpy()
        else:
            numpy_func = input_ids.cpu().numpy()

        inputs = {session.get_inputs()[0].name: numpy_func(input_ids)}
        out = session.run(None, inputs)

        return {"result": np.argmax(out)}
    except Exception as err:
        result = str(err)
        return {"error": result}
```

Next, create the configuration to perform the inferencing. Since these examples are using the Python SDK, you can use them in a Jupyter Notebook or using the Python shell directly. Start by creating a YAML file that describes your environment:

```
from azureml.core.conda_dependencies import CondaDependencies

environ = CondaDependencies.create(
  pip_packages=[
    "numpy","onnxruntime","azureml-core", "azureml-defaults",
```

```
        "torch", "transformers"]
    )

    with open("environ.yml","w") as f:
        f.write(environ.serialize_to_string())
```

Now that the YAML file is done, set up the configuration:

```
    from azureml.core.model import InferenceConfig
    from azureml.core.environment import Environment


    environ = Environment.from_conda_specification(
        name="environ", file_path="environ.yml"
    )
    inference_config = InferenceConfig(
        entry_script="score.py", environment=environ
    )
```

Finally, deploy the model using the SDK. Create the Azure Container Instance (ACI) configuration first:

```
    from azureml.core.webservice import AciWebservice

    aci_config = AciWebservice.deploy_configuration(
        cpu_cores=1,
        memory_gb=1,
        tags={"model": "onnx", "type": "language"},
        description="Container service for the RoBERTa ONNX model",
    )
```

With the `aci_config`, deploy the service:

```
    from azureml.core.model import Model

    # retrieve the model
    model = Model(ws, 'roberta-sequence', version=1)

    aci_service_name = "onnx-roberta-demo"
    aci_service = Model.deploy(
```

```
        ws, aci_service_name,
        [model],
        inference_config,
        aci_config
    )

    aci_service.wait_for_deployment(True)
```

Several things have happened to get this deployment working. First, you defined the environment with the dependencies required for the inferencing to work. Then you configured an Azure Container Instance, and finally, you retrieved version 1 of the *roberta_sequence* ONNX model and used the `Model.deploy()` method from the SDK to deploy the model. Again, the training specifics of the model aren't covered here. You could very well train any model in Azure, export it to ONNX, register it, and pick up right in this section to continue the deployment process. Few modifications are needed in these examples to make progress. Perhaps different libraries and certainly a different way to interact with the model are required. Still, this workflow empowers you to add another layer of automation to deploy models programmatically from PyTorch to ONNX (or straight from previously registered ONNX models) in a container instance in Azure.

You will want to use ONNX to deploy to other noncloud environments like mobile devices in some other situations. I cover some of the details involved in the next section with Apple's machine learning framework.

Apple Core ML

Apple's machine learning framework is somewhat unique in that there is support to convert Core ML models into ONNX *as well as* convert them from ONNX to Core ML. As I've said already in this chapter, you have to be very careful and ensure that there is support for the model conversion and getting the versions right. Currently, *coremltools* supports ONNX opset versions 10 and newer. It isn't that hard to get into a situation where a model lacks support and breakage occurs. Aside from the ONNX

support, you must be aware of the target conversion environment and if that environment supports iOS and macOS releases.

TIP

See the minimum target supported for different environments in the *Core ML* documentation.

Aside from support in a target environment like iOS, there is also a good list of tested models from ONNX known to work well. It is from that list that I'll pick up the MNIST model to try out a conversion. Go to the Model Zoo and find the MNIST section. Download the latest version (1.3 in my case). Now create a new virtual environment and a *requirements.txt* with the following libraries, which include *coremltools*:

```
attr==0.3.1
attrs==20.3.0
coremltools==4.1
mpmath==1.2.1
numpy==1.19.5
onnx==1.8.1
packaging==20.9
protobuf==3.15.6
pyparsing==2.4.7
scipy==1.6.1
six==1.15.0
sympy==1.7.1
tqdm==4.59.0
typing-extensions==3.7.4.3
```

Install the dependencies so that we can create tooling to do the conversion. Let's create the most straightforward tool possible, just like in "Create a Generic ONNX Checker", without argument parsers or any fancy help menus. Start by creating the `main()` function and the special Python magic needed at the end of the file so that you can call it in the terminal as a script:

```
import sys
from coremltools.converters.onnx import convert

def main(arguments):
    pass

if __name__ == '__main__':
    main(sys.argv)
```

In this case, the script isn't doing anything useful yet, and I'm also skipping implementing a help menu. You should always include a help menu in your scripts so that others can have some idea of the inputs and outputs of the program when they need to interact with it. Update the `main()` function to try out the conversion. I'll assume that the last argument received will represent a path to the ONNX model that needs converting:

```
def main(arguments):
    model_path = arguments[-1]
    basename = model_path.split('.onnx')[0]
    model = convert(model_path,  minimum_ios_deployment_target='13')
    model.short_description = "ONNX Model converted with coremltools"
    model.save(f"{basename}.mlmodel")
```

First, the function captures the last argument as the path to the ONNX model, and then it computes the base name by stripping out the *.onnx* suffix. Finally, it goes through the conversion (using a minimum iOS version target of 13), includes a description, and saves the output. Try the updated script with the previously downloaded MNIST model:

```
$ python converter.py mnist-8.onnx
1/11: Converting Node Type Conv
2/11: Converting Node Type Add
3/11: Converting Node Type Relu
4/11: Converting Node Type MaxPool
5/11: Converting Node Type Conv
6/11: Converting Node Type Add
7/11: Converting Node Type Relu
8/11: Converting Node Type MaxPool
```

```
9/11: Converting Node Type Reshape
10/11: Converting Node Type MatMul
11/11: Converting Node Type Add
Translation to CoreML spec completed. Now compiling the CoreML model.
Model Compilation done.
```

The resulting operation should've produced a *mnist-8.mlmodel* file, which is a Core ML model that you can now load in a macOS computer that has XCode installed. Use the Finder in your Apple computer, double-click the newly generated *coreml* model, and double-click it. The MNIST model should load right away, with the description included in the converter script as shown in Figure 10-4.

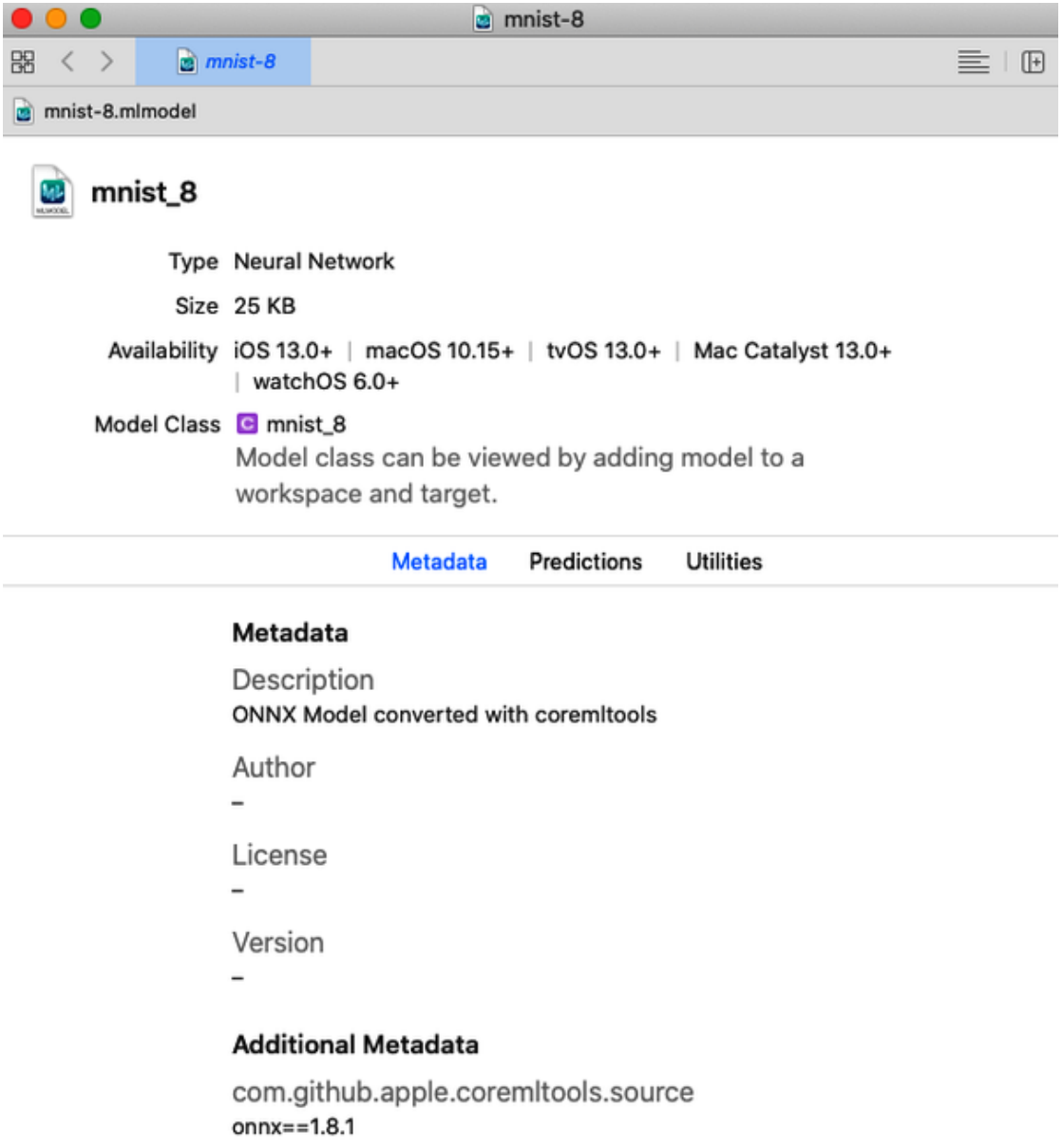

Figure 10-4. ONNX to Core ML

Verify that the Availability section shows the minimum iOS targets of 13, just like the converter script set them. The Predictions section has useful information about the inputs and outputs that the model accepts, as shown in Figure 10-5.

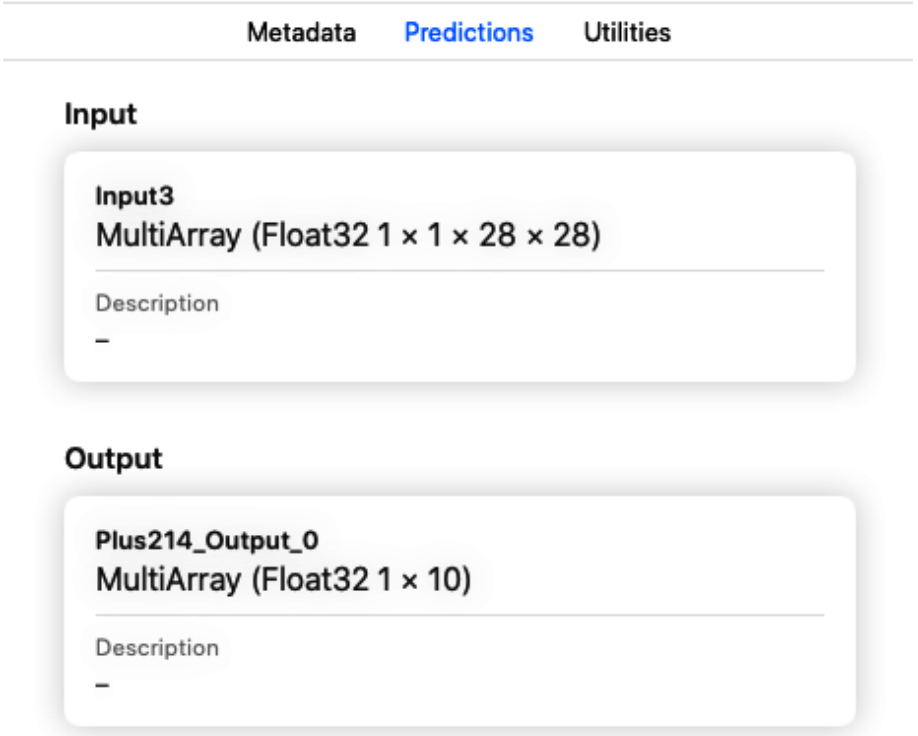

Figure 10-5. ONNX to CoreML Predictions

Finally, the Utilities section provides helpers to deploy that model using a model archive that integrates with CloudKit (Apple's environment for iOS application resources) as shown in Figure 10-6.

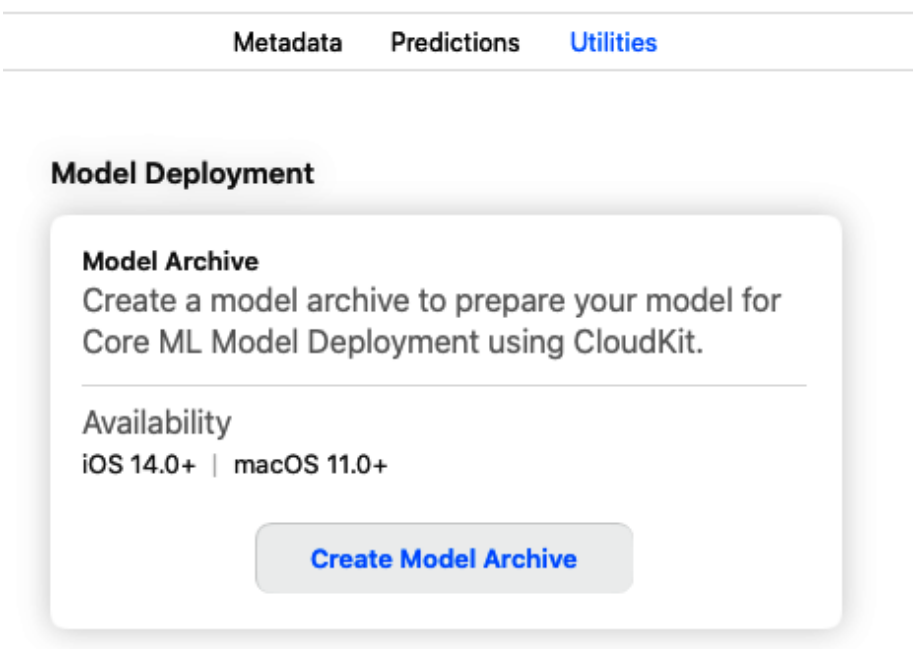

Figure 10-6. ONNX to Core ML model archive

It is exciting to see broad support for ONNX and growing support in other frameworks and operating systems like OSX in this case. If you are interested in iOS development and deploying models, you can still use the frameworks you are used to. Then, you can do the conversion to the targeted iOS environment for deployment. This process makes a compelling reason to use ONNX because it allows you to leverage well-established frameworks that you can transform into targeted environments. Next, let's see some other edge integrations that further emphasize the usefulness of the ONNX framework and tooling.

Edge Integration

Recently, ONNX announced a new internal model format known as ORT, which minimizes the build size of the model for optimal deployment on embedded or edge devices. The smaller footprint of a model has several aspects that help on edge devices. Edge devices have limited storage capacity, and most times, the storage isn't fast at all. Reading and writing to small storage devices can quickly become problematic. Further, ONNX, in general, keeps trying to support more and different hardware configurations with varying CPUs and GPUs; not an easy problem to solve, but indeed a welcomed effort. The better and broader the support, the easier it is to get machine learning models deployed to help environments where

this was not possible before. As I've covered most of Edge's benefits and crucial aspects in "Edge Devices", this section will concentrate on getting a conversion done from an ONNX model down to the ORT format.

Start by creating a new virtual environment, activating it, and then installing the dependencies as shown in this *requirements.txt* file:

```
flatbuffers==1.12
numpy==1.20.1
onnxruntime==1.7.0
protobuf==3.15.6
six==1.15.0
```

There is currently no separate tooling available to install, so you are required to clone the entire *onnxruntime* repository to try a conversion to ORT. After cloning, you will use the *convert_onnx_models_to_ort.py* file in the *tools/python* directory:

```
$ git clone https://github.com/microsoft/onnxruntime.git
$ python onnxruntime/tools/python/convert_onnx_models_to_ort.py --help
usage: convert_onnx_models_to_ort.py [-h]
[...]
Convert the ONNX format model/s in the provided directory to ORT format models
All files with a `.onnx` extension will be processed. For each one, an ORT
format model will be created in the same directory. A configuration file will
also be created called `required_operators.config`, and will contain the list
of required operators for all converted models. This configuration file should
be used as input to the minimal build via the `--include_ops_by_config`
parameter.
[...]
```

The ORT converter produces a configuration file as well as an ORT model file from the ONNX model. You can deploy the optimized model along with a unique ONNX runtime build. First, try a conversion with an ONNX model. In this example, I downloaded the mobilenet_v2-1.0 ONNX model into the *models/* directory and used it as an argument to the converter script:

```
$ python onnxruntime/tools/python/convert_onnx_models_to_ort.py \
        models/mobilenetv2-7.onnx
Converting optimized ONNX model to ORT format model models/mobilenetv2-7.ort
Processed models/mobilenetv2-7.ort
Created config in models/mobilenetv2-7.required_operators.config
```

The configuration is crucial here because it lists the operators needed by the model. This allows you to create an ONNX runtime with only those operators. For the converted model, that file looks like this:

```
ai.onnx;1;Conv,GlobalAveragePool
ai.onnx;5;Reshape
ai.onnx;7;Add
com.microsoft;1;FusedConv
```

You can accomplish a binary size reduction for the runtime by specifying only the needs of the model. I will not be covering all the specifics on how to build the ONNX runtime from source, but you can use the build guide as a reference for the next steps as we explore what flags and options are helpful in the binary reduction.

First, you must use the `--include_ops_by_config` flag. In this case, the value for this flag is the path to the generated config from the previous step. In my case, that path is *models/mobilenetv2-7.required_operators.config*. I also suggest you try out the `--minimal_build`, which supports loading and executing ORT models only (dropping support for normal ONNX formats). Finally, if you are targeting an Android device, using the `--android_cpp_shared` flag will produce a smaller binary by using the shared *libc++* library instead of the static one that comes by default in the runtime.

Conclusion

The commonality that a framework (and set of tools) like ONNX provides helps the whole machine learning ecosystem. One of the ideals of this book is to provide *practical* examples to get models into production in a reproducible and reliable way. The more comprehensive the support for

machine learning models, the easier it will be for engineers to try it out and take advantage of everything machine learning offers. I'm particularly excited about edge applications, especially for remote environments where it is impossible to connect to the internet or have any network connectivity. ONNX is lowering the friction that exists to deploy to those environments. I hope that the effort to continue to make this easier with more tooling and better support will continue to benefit from collective knowledge and contributions. Although we briefly tried command line tools, I go into much more detail on how to make robust command line tools with error handling and well-defined flags in the next chapter. In addition, I cover Python packaging and using microservices, which will give you the flexibility to try different approaches when solving machine learning challenges.

Exercises

- Update all scripts to verify the produced ONNX model with the script from "Create a Generic ONNX Checker".
- Modify the Core ML converter script to use the Click framework for better parsing of options and a help menu.
- Group three converters into a single command line tool so that it is easy to make conversions with different inputs.
- Improve the *tf2onnx* converter so that it is wrapped in a new script, which can catch common errors and report them with a more user-friendly message.
- Use a different ONNX model for an Azure deployment.

Critical Thinking Discussion Questions

- Why is ONNX important? Give at least three reasons.
- What is something useful about creating a script without a command line tool framework? What are the advantages of using a framework?
- How is the ORT format useful? In what situations can you use it?
- What are some problems that you may encounter if portability doesn't exist? Give three reasons why improving those problems will improve machine learning in general.