ONNX and the JVM

Adam Pocock

Machine Learning Researcher

Oracle Labs, Machine Learning Research Group

June 24, 2022

Why do we want to support ONNX on the JVM?

- Machine Learning models are an increasingly important component in applications

- Most applications (especially large business applications) are developed in non-Python languages
  - Either we need to persuade all those developers to move to Python (which seems unlikely)
  - Or we bring Machine Learning to them in the languages they work in like Java (or C#, JS)

- Java is one of the largest platforms for software development in the world, with millions of Java developers building software which runs companies

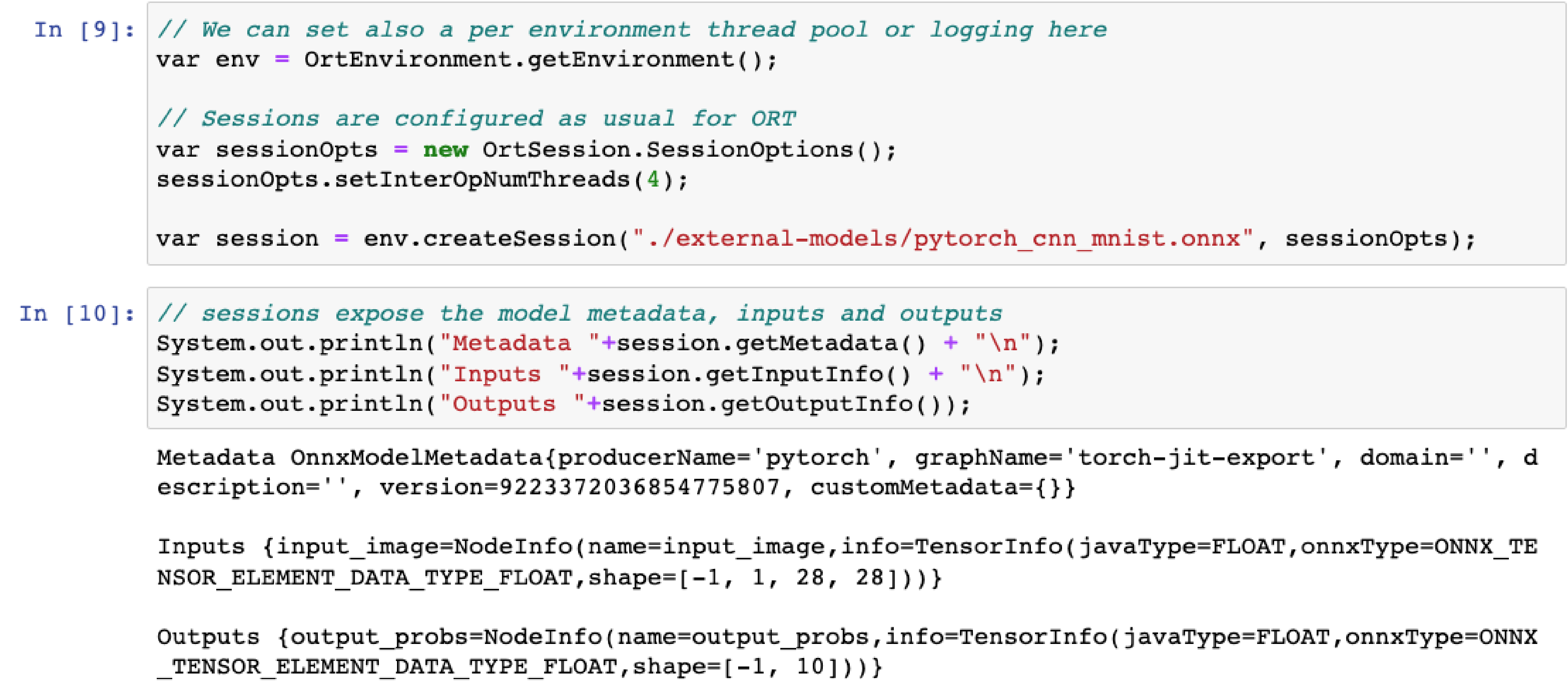
- We think that the ONNX community (and the wider ML community) could be building tools to help Java developers integrate ML into their applications

- I’ve spent the past few years building Java ML tooling, both for ONNX and other ML libraries

ONNX Runtime Java API

- Developed in Oracle Labs in Spring 2019, contributed to the upstream project in December 2019
  - Binaries available on Maven Central since June 2020 as part of ORT 1.3.1
  - Used in production in Oracle and other companies
- Goal is to provide the whole ORT C API in Java, keep feature parity with the other APIs
  - Currently missing custom allocators & IOBinding due to complexities in exposing pointers
  - If there's something else missing, or you need those features, open an issue
- The Java API is a thin layer over the C API with minimal performance impact
  - Input tensors have a zero copy path from Java -> ORT
  - Output tensors currently require a copy (but for many tasks are much smaller than inputs)
- Targets Java 8 (and runs on all versions > 8), has no dependencies other than the ORT native library which is packaged with it

ONNX Runtime in Java code example (I)



```
In [9]:  // We can set also a per environment thread pool or logging here
         var env = OrtEnvironment.getEnvironment();

         // Sessions are configured as usual for ORT
         var sessionOpts = new OrtSession.SessionOptions();
         sessionOpts.setInterOpNumThreads(4);

         var session = env.createSession("./external-models/pytorch_cnn_mnist.onnx", sessionOpts);
```

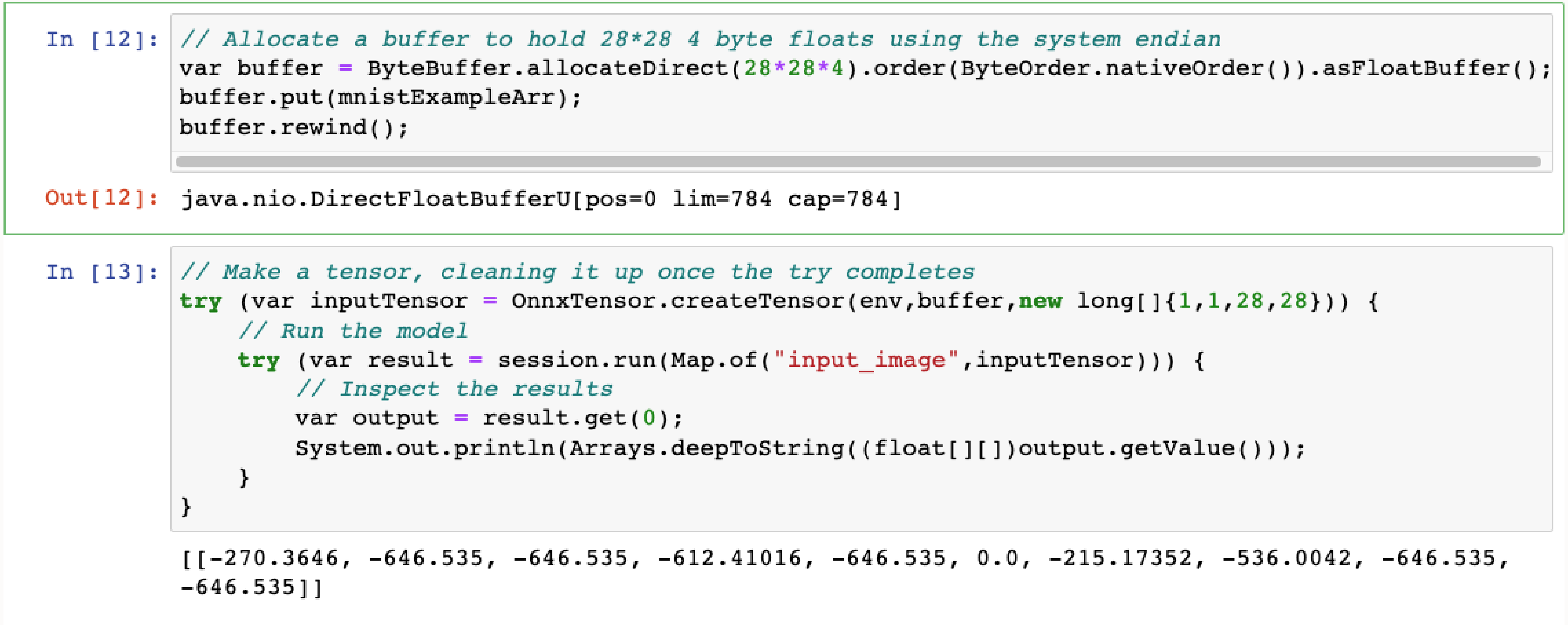
```
In [10]: // sessions expose the model metadata, inputs and outputs
         System.out.println("Metadata "+session.getMetadata() + "\n");
         System.out.println("Inputs "+session.getInputInfo() + "\n");
         System.out.println("Outputs "+session.getOutputInfo());
```

```
Metadata OnnxModelMetadata{producerName='pytorch', graphName='torch-jit-export', domain='', d
escription='', version=9223372036854775807, customMetadata={}}

Inputs {input_image=NodeInfo(name=input_image,info=TensorInfo(javaType=FLOAT,onnxType=ONNX_TE
NSOR_ELEMENT_DATA_TYPE_FLOAT,shape=[-1, 1, 28, 28]))}

Outputs {output_probs=NodeInfo(name=output_probs,info=TensorInfo(javaType=FLOAT,onnxType=ONNX
_TENSOR_ELEMENT_DATA_TYPE_FLOAT,shape=[-1, 10]))}
```

ONNX Runtime in Java code example (II)



```
In [12]: // Allocate a buffer to hold 28*28 4 byte floats using the system endian
         var buffer = ByteBuffer.allocateDirect(28*28*4).order(ByteOrder.nativeOrder()).asFloatBuffer();
         buffer.put(mnistExampleArr);
         buffer.rewind();

Out[12]: java.nio.DirectFloatBufferU[pos=0 lim=784 cap=784]
```

```
In [13]: // Make a tensor, cleaning it up once the try completes
         try (var inputTensor = OnnxTensor.createTensor(env,buffer,new long[]{1,1,28,28})) {
             // Run the model
             try (var result = session.run(Map.of("input_image",inputTensor))) {
                 // Inspect the results
                 var output = result.get(0);
                 System.out.println(Arrays.deepToString((float[][])output.getValue()));
             }
         }

         [[-270.3646, -646.535, -646.535, -612.41016, -646.535, 0.0, -215.17352, -536.0042, -646.535,
         -646.535]]
```

Memory management

- Much of the work in this API is shuffling memory between the Java heap and the native heap
  - This needs to be as efficient as possible to maximise throughput & minimize latency

- All Java objects which hold native objects must be closed by users otherwise they leak memory
  - This is typically done with a try-with-resources statement, and in the future we will add a safety net to ensure memory is freed as the Java objects are GC'd

- We recommend users use NIO direct byte buffers, which allow a zero copy pass through from Java to native code
  - The buffer lifetime needs to be managed so it's longer than a single call
  - Buffers can be reused for same size inputs reducing allocation
  - Java's existing multidimensional arrays are a poor abstraction for ML as they are not flat and require pointer chasing for 2D or higher structures

Future work on ONNX Runtime's Java API

- Moving to a modern version of Java as Java 8 is 8 years old
  - Features like the JEP 424 Foreign Function & Memory interface make things faster and safer by allowing easier cleanup of native memory and autogenerating the native interface
  - There have been many language & runtime improvements which should improve the code

- We're interested in adding support for single op execution and training as these provide functionality hard to access elsewhere on the JVM

- Continued build out to match the ORT C API
  - New EPs, new methods, better support for memory pinning with IOBinding

- Contributions are welcome – https://github.com/microsoft/onnxruntime

Writing ONNX models from Java

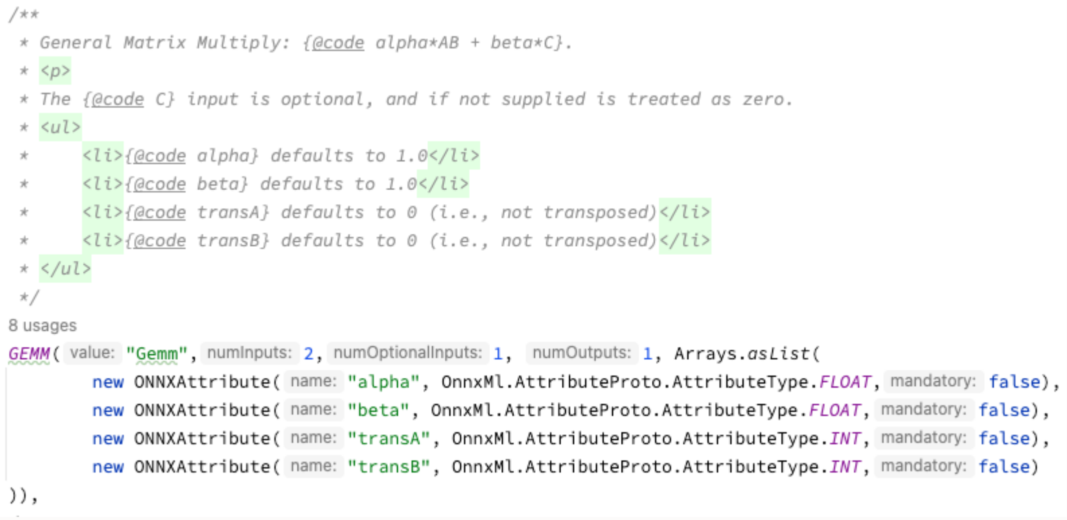
- Inference is an important workload but we'd also like to convert models trained in Java to ONNX

- Writing the protobuf directly is possible, but it's tricky to write well formed ONNX models
  - Bare protobufs have no graph validation for cycles or checks that node inputs and outputs line up

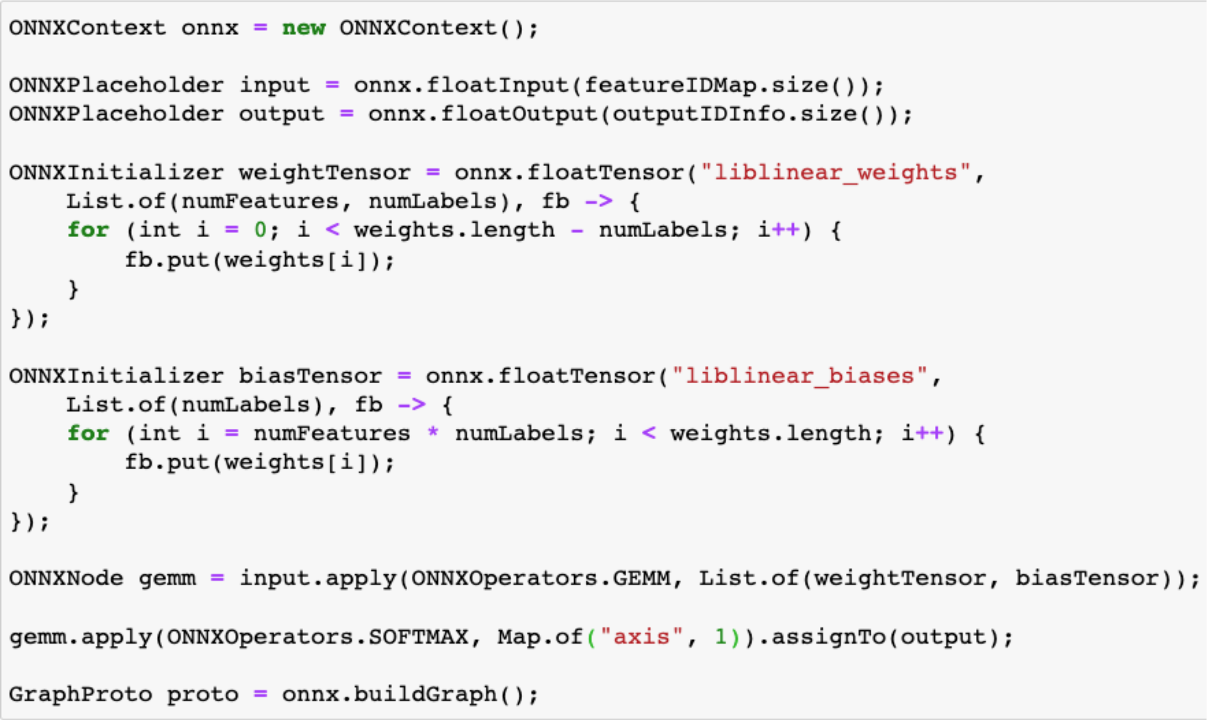
- We developed and open sourced a small library for generating ONNX models in Java
  - It provides some type safety, graph correctness checking, attribute validation, export of Java arrays as initializers or attributes, and a fluent interface
  - It's Apache 2.0 licensed, and lives inside the Tribuo repository – https://github.com/oracle/tribuo/tree/main/Util/ONNXExport
  - Built to support converting Tribuo models to ONNX, but only depends on protobuf so can be used without Tribuo, and also targets Java 8 (but works on all versions > 8)

Writing ONNX models from Java

Code examples



```
/**
 * General Matrix Multiply: {@code alpha*AB + beta*C}.
 * <p>
 * The {@code C} input is optional, and if not supplied is treated as zero.
 * <ul>
 *     <li>{@code alpha} defaults to 1.0</li>
 *     <li>{@code beta} defaults to 1.0</li>
 *     <li>{@code transA} defaults to 0 (i.e., not transposed)</li>
 *     <li>{@code transB} defaults to 0 (i.e., not transposed)</li>
 * </ul>
 */
8 usages
GEMM( value: "Gemm", numInputs: 2, numOptionalInputs: 1,  numOutputs: 1, Arrays.asList(
        new ONNXAttribute( name: "alpha", OnnxMl.AttributeProto.AttributeType.FLOAT, mandatory: false),
        new ONNXAttribute( name: "beta", OnnxMl.AttributeProto.AttributeType.FLOAT, mandatory: false),
        new ONNXAttribute( name: "transA", OnnxMl.AttributeProto.AttributeType.INT, mandatory: false),
        new ONNXAttribute( name: "transB", OnnxMl.AttributeProto.AttributeType.INT, mandatory: false)
)),
```



```
ONNXContext onnx = new ONNXContext();

ONNXPlaceholder input = onnx.floatInput(featureIDMap.size());
ONNXPlaceholder output = onnx.floatOutput(outputIDInfo.size());

ONNXInitializer weightTensor = onnx.floatTensor("liblinear_weights",
    List.of(numFeatures, numLabels), fb -> {
    for (int i = 0; i < weights.length - numLabels; i++) {
        fb.put(weights[i]);
    }
});

ONNXInitializer biasTensor = onnx.floatTensor("liblinear_biases",
    List.of(numLabels), fb -> {
    for (int i = numFeatures * numLabels; i < weights.length; i++) {
        fb.put(weights[i]);
    }
});

ONNXNode gemm = input.apply(ONNXOperators.GEMM, List.of(weightTensor, biasTensor));

gemm.apply(ONNXOperators.SOFTMAX, Map.of("axis", 1)).assignTo(output);

GraphProto proto = onnx.buildGraph();
```

Future work on writing ONNX models in Java

- We currently support a subset of opset 13 and ONNX-ML v1, those used to export Tribuo models, we'd like to expand this to full coverage of ONNX ops
  - It's easy to expand the operator enum to fill out the set
  - In the future we may look at autogenerating the enum (or op classes) from the op definitions
- Abstract over opsets to allow users to export models targeting different opsets
  - This is straightforward to do, but we haven't needed it yet
  - Also enables users to integrate custom ops into their models
- Integrate provenance and metadata into converted models
  - Tribuo exports its detailed model provenance as a field in the ONNX metadata, but this isn't standardised, we're interested in collaborating with the ONNX community on better solutions
- Contributions are welcome – https://github.com/oracle/tribuo

Questions?