

When to use TensorFlowSharp

TensorFlowSharp is a good runtime to run your existing models, and is mostly a straight binding to the underlying TensorFlow runtime. Most people will want to use a higher-level library for interfacing with TensorFlow.

The library was designed to blend in the .NET ecosystem and use the .NET naming conventions.

I strongly recommend that you use TensorFlow.NET which takes a different approach than TensorFlowSharp, it uses the Python naming convention and has a much broader support for the higher level operations that you are likely to need - and is also actively maintained.

TensorFlowSharp

TensorFlowSharp are .NET bindings to the TensorFlow library published here:

<https://github.com/tensorflow/tensorflow>

This surfaces the C API as a strongly-typed .NET API for use from C# and F#.

The API surfaces the entire low-level TensorFlow API, it is on par with other language bindings. But currently does not include a high-level API like the Python binding does, so it is more cumbersome to use for those high level operations.

You can prototype using TensorFlow or Keras in Python, then save your graphs or trained models and then load the result in .NET with TensorFlowSharp and feed your own data to train or run.

The current API documentation is here.

Using TensorFlowSharp

Installation

The easiest way to get started is to use the NuGet package for TensorFlowSharp which contains both the .NET API as well as the native libraries for 64-bit Linux, Mac and Windows using the CPU backend.

You can install using NuGet like this:

```
1 nuget install TensorFlowSharp
```

Or select it from the NuGet packages UI on Visual Studio.

On Visual Studio, make sure that you are targeting .NET 4.6.1 or later, as this package uses some features of newer .NETs. Otherwise, the package will not be added. Once you do this, you can just use the TensorFlowSharp nuget

Alternatively, you can download it directly.

Using TensorFlowSharp

Your best source of information right now are the SampleTest that exercises various APIs of TensorFlowSharp, or the stand-alone samples located in “Examples”.

This API binding is closer design-wise to the Java and Go bindings which use explicit TensorFlow graphs and sessions. Your application will typically create a graph (TFGraph) and setup the operations there, then create a session from it (TFSession), then use the session runner to setup inputs and outputs and execute the pipeline.

Something like this:

```
1 using (var graph = new TFGraph ())
2 {
3     // Load the model
4     graph.Import (File.ReadAllBytes ("MySavedModel"));
5     using (var session = new TFSession (graph))
6     {
7         // Setup the runner
8         var runner = session.GetRunner ();
9         runner.AddInput (graph ["input"] [0], tensor);
10        runner.Fetch (graph ["output"] [0]);
11
12        // Run the model
13        var output = runner.Run ();
14
15        // Fetch the results from output:
16        TFTensor result = output [0];
17    }
18 }
```

If your application is sensitive to GC cycles, you can run your model as follows. The `Run` method will then allocate managed memory only at the first call and reuse it later on. Note that this requires you to reuse the `Runner` instance and not to change the shape of the input data:

```
1 // Some input matrices
2 var inputs = new float[,] {
3     new float[,] { { 1, 2 }, { 3, 4 } },
4     new float[,] { { 2, 4 }, { 6, 8 } }
```

```

5 };
6
7 // Assumes all input matrices have identical shape
8 var shape = new long[] { inputs[0].GetLongLength(0), inputs[0].
    GetLongLength(1) };
9 var size = inputs[0].Length * sizeof(float);
10
11 // Empty input and output tensors
12 var input = new TFTensor(TFDataType.Float, shape, size);
13 var output = new TFTensor[1];
14
15 // Result array for a single run
16 var result = new float[1, 1];
17
18 using (var graph = new TFGraph())
19 {
20     // Load the model
21     graph.Import(File.ReadAllBytes("MySavedModel"));
22     using (var session = new TFSession(graph))
23     {
24         // Setup the runner
25         var runner = session.GetRunner();
26         runner.AddInput(graph["input"][0], input);
27         runner.Fetch(graph["output"][0]);
28
29         // Run the model on each input matrix
30         for (int i = 0; i < inputs.Length; i++)
31         {
32             // Mutate the input tensor
33             input.SetValue(inputs[i]);
34
35             // Run the model
36             runner.Run(output);
37
38             // Fetch the result from output into `result`
39             output[0].GetValue(result);
40         }
41     }
42 }

```

In scenarios where you do not need to setup the graph independently, the session will create one for you. The following example shows how to abuse TensorFlow to compute the addition of two numbers:

```

1 using (var session = new TFSession())
2 {
3     var graph = session.Graph;
4
5     var a = graph.Const(2);
6     var b = graph.Const(3);

```

```
7 Console.WriteLine("a=2 b=3");
8
9 // Add two constants
10 var addingResults = session.GetRunner().Run(graph.Add(a, b));
11 var addingResultValue = addingResults.GetValue();
12 Console.WriteLine("a+b={0}", addingResultValue);
13
14 // Multiply two constants
15 var multiplyResults = session.GetRunner().Run(graph.Mul(a, b));
16 var multiplyResultValue = multiplyResults.GetValue();
17 Console.WriteLine("a*b={0}", multiplyResultValue);
18 }
```

Here is an F# scripting version of the same example, you can use this in F# Interactive:

```
1 #r @"packages\TensorFlowSharp.1.4.0\lib\net471\TensorFlowSharp.dll"
2
3 open System
4 open System.IO
5 open TensorFlow
6
7 // set the path to find the native DLL
8 Environment.SetEnvironmentVariable("Path",
9     Environment.GetEnvironmentVariable("Path") + ";" +
10     __SOURCE_DIRECTORY__ + @"/packages/TensorFlowSharp.1.2.2/native")
11
12 module AddTwoNumbers =
13     let session = new TFSession()
14     let graph = session.Graph
15
16     let a = graph.Const(new TFTensor(2))
17     let b = graph.Const(new TFTensor(3))
18     Console.WriteLine("a=2 b=3")
19
20     // Add two constants
21     let addingResults = session.GetRunner().Run(graph.Add(a, b))
22     let addingResultValue = addingResults.GetValue()
23     Console.WriteLine("a+b={0}", addingResultValue)
24
25     // Multiply two constants
26     let multiplyResults = session.GetRunner().Run(graph.Mul(a, b))
27     let multiplyResultValue = multiplyResults.GetValue()
28     Console.WriteLine("a*b={0}", multiplyResultValue)
```

Working on TensorFlowSharp

If you want to work on extending TensorFlowSharp or contribute to its development read the CONTRIBUTING.md file.

Please keep in mind that this requires a modern version of C# as this uses some new capabilities there. So you will want to use Visual Studio 2017.

Possible Contributions

Build More Tests

Would love to have more tests to ensure the proper operation of the framework.

Samples

The binding is pretty much complete, and at this point, I want to improve the API to be easier and more pleasant to use from both C# and F#. Creating samples that use Tensorflow is a good way of finding easy wins on the usability of the API, there are some here:

<https://github.com/tensorflow/models>

Packaging

Mobile: we need to package the library for consumption on Android and iOS.

Documentation Styling

The API documentation has not been styled, I am using the barebones template for documentation, and it can use some work.

Issues

I have logged some usability problems and bugs in Issues, feel free to take on one of those tasks.

Documentation

Much of the online documentation comes from TensorFlow and is licensed under the terms of Apache 2 License, in particular all the generated documentation for the various operations that is generated by using the tensorflow reflection APIs.

Last API update: Release 1.9