# Tutorial: Create a Windows Machine Learning Desktop application (C++)

Windows ML APIs can be leveraged to easily interact with machine learning models within C++ desktop (Win32) applications. Using the three steps of loading, binding, and evaluating, your application can benefit from the power of machine learning.



Load -> Bind -> Evaluate

We will be creating a somewhat simplified version of the SqueezeNet Object Detection sample, which is available on [GitHub](https://github.com/Microsoft/Windows-Machine-Learning/tree/master/Samples/SqueezeNetObjectDetection/Desktop/cpp). You can download the complete sample if you want to see what it will be like when you finish.

We'll be using C++/WinRT to access the WinML APIs. See [C++/WinRT](https://docs.microsoft.com/windows/uwp/cpp-and-winrt-apis/) for more information.

In this tutorial, you'll learn how to:

* Load a machine learning model
* Load an image as a [VideoFrame](https://docs.microsoft.com/uwp/api/windows.media.videoframe)
* Bind the model's inputs and outputs
* Evaluate the model and print meaningful results

## Prerequisites

* [Visual Studio 2017, version 15.7.4 or later](https://developer.microsoft.com/windows/downloads)
* [Windows 10, version 1809 or later](https://www.microsoft.com/software-download/windows10)
* [Windows SDK, build 17763 or later](https://www.microsoft.com/software-download/windowsinsiderpreviewSDK)
* [Visual Studio extension for C++/WinRT](https://marketplace.visualstudio.com/items?itemName=CppWinRTTeam.cppwinrt101804264)
* [Windows-Machine-Learning Github repo](https://github.com/Microsoft/Windows-Machine-Learning) (you can either download it as a ZIP file or clone to your machine; alternatively, it should be installed already at **<insert path here>**)

## Create the project

First, we will create the project in Visual Studio:

1. Select **File > New > Project** to open the **New Project** window.
2. In the left pane, select **Installed > Visual C++ > Windows Desktop**, and in the middle, select **Windows Console Application (C++/WinRT)**.
3. Give your project a **Name** and **Location**, then click **OK**.
4. In the **New Universal Windows Platform Project** window, set the **Target** and **Minimum Versions** both to build 17763 or later, and click **OK**.
5. Make sure the dropdown menus in the top toolbar are set to **Debug** and either **x64** or **x86** depending on your computer's architecture.
6. Press **Ctrl+F5** to run the program without debugging. A terminal should open with some "Hello world" text. Press any key to close it.

## Load the model

Next, we'll load the ONNX model into our program using [LearningModel.LoadFromFilePath](https://docs.microsoft.com/uwp/api/windows.ai.machinelearning.learningmodel.loadfromfilepath):

1. In **pch.h** (in the **Header Files** folder), add the following include statements (these give us access to all the APIs that we'll need):

* #include <winrt/Windows.AI.MachineLearning.h>  
  #include <winrt/Windows.Foundation.Collections.h>  
  #include <winrt/Windows.Graphics.Imaging.h>  
  #include <winrt/Windows.Media.h>  
  #include <winrt/Windows.Storage.h>  
    
  #include <string>  
  #include <fstream>  
    
  #include <Windows.h>

1. In **main.cpp** (in the **Source Files** folder), add the following using statements:

* using namespace Windows::AI::MachineLearning;  
  using namespace Windows::Foundation::Collections;  
  using namespace Windows::Graphics::Imaging;  
  using namespace Windows::Media;  
  using namespace Windows::Storage;  
    
  using namespace std;

1. Add the following variable declarations after the using statements:

* // Global variables  
  hstring modelPath;  
  string deviceName = "default";  
  hstring imagePath;  
  LearningModel model = nullptr;  
  LearningModelDeviceKind deviceKind = LearningModelDeviceKind::Default;  
  LearningModelSession session = nullptr;  
  LearningModelBinding binding = nullptr;  
  VideoFrame imageFrame = nullptr;  
  string labelsFilePath;  
  vector<string> labels;

1. Add the following forward declarations after your global variables:

* // Forward declarations  
  void LoadModel();  
  VideoFrame LoadImageFile(hstring filePath);  
  void BindModel();  
  void EvaluateModel();  
  void PrintResults(IVectorView<float> results);  
  void LoadLabels();

1. In **main.cpp**, remove the "Hello world" code (everything in the main function after init\_apartment).
2. Find the **SqueezeNet.onnx** file in your local clone of the **Windows-Machine-Learning** repo. It should be located in \*\*-Machine-Learning\*.
3. Copy the file path and assign it to your modelPath variable where we defined it at the top. Remember to prefix the string with an L to make it a wide character string so that it works properly with hstring, and to escape any backslashes (\) with an extra backslash. For example:

* hstring modelPath = L"C:\\Repos\\Windows-Machine-Learning\\SharedContent\\models\\SqueezeNet.onnx";

1. First, we'll implement the LoadModel method. Add the following method after the main method. This method loads the model and outputs how long it took:

* void LoadModel()  
  {  
   // load the model  
   printf("Loading modelfile '%ws' on the '%s' device\n", modelPath.c\_str(), deviceName.c\_str());  
   DWORD ticks = GetTickCount();  
   model = LearningModel::LoadFromFilePath(modelPath);  
   ticks = GetTickCount() - ticks;  
   printf("model file loaded in %d ticks\n", ticks);  
  }

1. Finally, call this method from the main method:

* LoadModel();

1. Run your program without debugging. You should see that your model loads successfully!

## Load the image

Next, we'll load the image file into our program:

1. Add the following method. This method will load the image from the given path and create a [VideoFrame](https://docs.microsoft.com/uwp/api/windows.media.videoframe) from it:

* VideoFrame LoadImageFile(hstring filePath)  
  {  
   printf("Loading the image...\n");  
   DWORD ticks = GetTickCount();  
   VideoFrame inputImage = nullptr;  
    
   try  
   {  
   // open the file  
   StorageFile file = StorageFile::GetFileFromPathAsync(filePath).get();  
   // get a stream on it  
   auto stream = file.OpenAsync(FileAccessMode::Read).get();  
   // Create the decoder from the stream  
   BitmapDecoder decoder = BitmapDecoder::CreateAsync(stream).get();  
   // get the bitmap  
   SoftwareBitmap softwareBitmap = decoder.GetSoftwareBitmapAsync().get();  
   // load a video frame from it  
   inputImage = VideoFrame::CreateWithSoftwareBitmap(softwareBitmap);  
   }  
   catch (...)  
   {  
   printf("failed to load the image file, make sure you are using fully qualified paths\r\n");  
   exit(EXIT\_FAILURE);  
   }  
    
   ticks = GetTickCount() - ticks;  
   printf("image file loaded in %d ticks\n", ticks);  
   // all done  
   return inputImage;  
  }

1. Add a call to this method in the main method:

* imageFrame = LoadImageFile(imagePath);

1. Find the **media** folder in your local clone of the **Windows-Machine-Learning** repo. It should be located at \*\*-Machine-Learning\*.
2. Choose one of the images in that folder, and assign its file path to the imagePath variable where we defined it at the top. Remember to prefix it with an L to make it a wide character string, and to escape any backslashes with another backslash. For example:

* hstring imagePath = L"C:\\Repos\\Windows-Machine-Learning\\SharedContent\\media\\kitten\_224.png";

1. Run the program without debugging. You should see the image loaded successfully!

## Bind the input and output

Next, we'll create a session based on the model and bind the input and output from the session using [LearningModelBinding.Bind](https://docs.microsoft.com/uwp/api/windows.ai.machinelearning.learningmodelbinding.bind). For more information on binding, see [Bind a model](bind-a-model.md).

1. Implement the BindModel method. This creates a session based on the model and device, and a binding based on that session. We then bind the inputs and outputs to variables we've created using their names. We happen to know ahead of time that the input feature is named "data\_0" and the output feature is named "softmaxout\_1". You can see these properties for any model by opening them in [Netron](https://lutzroeder.github.io/Netron/), an online model visualization tool.

* void BindModel()  
  {  
   printf("Binding the model...\n");  
   DWORD ticks = GetTickCount();  
    
   // now create a session and binding  
   session = LearningModelSession{ model, LearningModelDevice(deviceKind) };  
   binding = LearningModelBinding{ session };  
   // bind the input image  
   binding.Bind(L"data\_0", ImageFeatureValue::CreateFromVideoFrame(imageFrame));  
   // bind the output  
   vector<int64\_t> shape({ 1, 1000, 1, 1 });  
   binding.Bind(L"softmaxout\_1", TensorFloat::Create(shape));  
    
   ticks = GetTickCount() - ticks;  
   printf("Model bound in %d ticks\n", ticks);  
  }

1. Add a call to BindModel from the main method:

* BindModel();

1. Run the program without debugging. The model's inputs and outputs should be bound successfully. We're almost there!

## Evaluate the model

We're now on the last step in the diagram at the beginning of this tutorial, **Evaluate**. We'll evaluate the model using [LearningModelSession.Evaluate](https://docs.microsoft.com/uwp/api/windows.ai.machinelearning.learningmodelsession.evaluate):

1. Implement the EvaluateModel method. This method takes our session and evaluates it using our binding and a correlation ID. The correlation ID is something we could possibly use later to match a particular evaluation call to the output results. Again, we know ahead of time that the output's name is "softmaxout\_1".

* void EvaluateModel()  
  {  
   // now run the model  
   printf("Running the model...\n");  
   DWORD ticks = GetTickCount();  
    
   auto results = session.Evaluate(binding, L"RunId");  
    
   ticks = GetTickCount() - ticks;  
   printf("model run took %d ticks\n", ticks);  
    
   // get the output  
   auto resultTensor = results.Outputs().Lookup(L"softmaxout\_1").as<TensorFloat>();  
   auto resultVector = resultTensor.GetAsVectorView();  
   PrintResults(resultVector);  
  }

1. Now let's implement PrintResults. This method gets the top three probabilities for what object could be in the image, and prints them:

* void PrintResults(IVectorView<float> results)  
  {  
   // load the labels  
   LoadLabels();  
   // Find the top 3 probabilities  
   vector<float> topProbabilities(3);  
   vector<int> topProbabilityLabelIndexes(3);  
   // SqueezeNet returns a list of 1000 options, with probabilities for each, loop through all  
   for (uint32\_t i = 0; i < results.Size(); i++)  
   {  
   // is it one of the top 3?  
   for (int j = 0; j < 3; j++)  
   {  
   if (results.GetAt(i) > topProbabilities[j])  
   {  
   topProbabilityLabelIndexes[j] = i;  
   topProbabilities[j] = results.GetAt(i);  
   break;  
   }  
   }  
   }  
   // Display the result  
   for (int i = 0; i < 3; i++)  
   {  
   printf("%s with confidence of %f\n", labels[topProbabilityLabelIndexes[i]].c\_str(), topProbabilities[i]);  
   }  
  }

1. We also need to implement LoadLabels. This method opens the labels file that contains all of the different objects that the model can recognize, and parses it:

* void LoadLabels()  
  {  
   // Parse labels from labels file. We know the file's entries are already sorted in order.  
   ifstream labelFile{ labelsFilePath, ifstream::in };  
   if (labelFile.fail())  
   {  
   printf("failed to load the %s file. Make sure it exists in the same folder as the app\r\n", labelsFilePath.c\_str());  
   exit(EXIT\_FAILURE);  
   }  
    
   std::string s;  
   while (std::getline(labelFile, s, ','))  
   {  
   int labelValue = atoi(s.c\_str());  
   if (labelValue >= labels.size())  
   {  
   labels.resize(labelValue + 1);  
   }  
   std::getline(labelFile, s);  
   labels[labelValue] = s;  
   }  
  }

1. Locate the **Labels.txt** file in your local clone of the **Windows-Machine-Learning** repo. It should be in \*\*-Machine-Learning\*.
2. Assign this file path to the labelsFilePath variable where we defined it at the top. Make sure to escape any backslashes with another backslash. For example:

* string labelsFilePath = "C:\\Repos\\Windows-Machine-Learning\\Samples\\SqueezeNetObjectDetection\\Desktop\\cpp\\Labels.txt";

1. Add a call to EvaluateModel in the main method:

* EvaluateModel();

1. Run the program without debugging. It should now correctly recognize what's in the image! Here is an example of what it might output:

* Loading modelfile 'C:\Repos\Windows-Machine-Learning\SharedContent\models\SqueezeNet.onnx' on the 'default' device  
  model file loaded in 250 ticks  
  Loading the image...  
  image file loaded in 78 ticks  
  Binding the model...Model bound in 15 ticks  
  Running the model...  
  model run took 16 ticks  
  tabby, tabby cat with confidence of 0.931461  
  Egyptian cat with confidence of 0.065307  
  Persian cat with confidence of 0.000193

## Next steps

Hooray, you've got object detection working in a C++ desktop application! Next, you can try using command line arguments to input the model and image files rather than hard-coding them, similar to what the sample on GitHub does. You could also try running the evaluation on a different device, like the GPU, to see how the performance differs.

Play around with the other samples on GitHub and extend them however you like!

## See also

* [Windows ML samples (GitHub)](https://github.com/Microsoft/Windows-Machine-Learning/tree/master)
* [Windows.AI.MachineLearning Namespace](https://docs.microsoft.com/uwp/api/windows.ai.machinelearning)
* [Windows ML](index.md)