Unity Sentis (Inference Engine) - Comprehensive Setup Guide & Sample Projects Tutorial

Unity Sentis (recently renamed to **Inference Engine**) is Unity's powerful neural network inference library that allows you to run AI models directly in Unity Runtime. Sentis enables AI models in the Unity Runtime to enhance your game or app on user devices without requiring cloud infrastructure.

What You Can Build with Unity Sentis

Unity Sentis allows for standardized AI model implementations such as style transfer and speech recognition. Models can be customized through model weights, layers, and chaining multiple inferences. Here are some practical applications:

- **Object Detection & Classification**: Detect, classify, and segment objects with an in-game or on-device camera
- **Smart NPCs**: Power a board game opponent with specific rules and custom difficulty curves
- **Handwriting Recognition**: Identify handwritten numbers, letters, and symbols for unique gameplay interactions
- **Depth Estimation**: Estimate the depth of real-world objects in an augmented reality view to occlude objects in a game scene
- **Speech Recognition**: Convert live speech to in-game text using a machine learning language model
- **Scientific Visualization**: Protein folding, physics simulations, and complex mathematical modeling

Key Advantages

- **Local Processing**: AI models are run locally with Unity Sentis, with no data stored or transferred to the cloud
- **Cross-Platform**: Sentis works with all Unity-supported platforms, from mobile to PC to popular game consoles
- **No Cloud Infrastructure**: Eliminates complex cloud infrastructure, network latency, and recurring inference costs
- **Easy Integration**: An intuitive "plug-and-play" API makes AI model integration accessible via the ONNX model file standard

---

System Requirements

Unity Version Requirements

- **Unity 2023.1 or higher** (recommended)
- Unity 2022.3 LTS (minimum supported)

Hardware Requirements

- **CPU**: Any modern multi-core processor
- **GPU**:
  - **Desktop**: DirectX 11/12 compatible GPU (NVIDIA/AMD)
  - **Mobile**: OpenGL ES 3.0+ or Vulkan compatible
- **RAM**: 4GB minimum, 8GB+ recommended
- **Storage**: 2GB+ available space

Platform Support

Sentis supports all the platforms Unity supports:

- Windows, macOS, Linux
- iOS, Android
- WebGL
- Console platforms (PlayStation, Xbox, Nintendo Switch)

Unity Sentis Installation & Setup

Step 1: Install Unity Sentis Package

1. **Open Unity Package Manager**:
   - Go to **Window > Package Manager**
   - In the dropdown, select **"Unity Registry"**
2. **Search and Install**:
   - Search for **"Sentis"** or package name: `com.unity.sentis`
   - Click **"Install"**
   - Current stable version: **2.1.3**

3. **Import Sample Projects** (Optional):

    - After installation, you'll see **"Import"** buttons for sample projects
    - These samples are invaluable for learning different aspects of Sentis

Step 2: Verify Installation

Create a simple test script to verify Sentis is working:

```
using UnityEngine;
using Unity.Sentis;

public class SentisTest : MonoBehaviour
{
    void Start()
    {
        Debug.Log("Unity Sentis is installed and ready!");

        // Check available backend types
        Debug.Log("Available backends:");
        Debug.Log("- CPU Backend: Available");
        if (SystemInfo.supportsComputeShaders)
        {
            Debug.Log("- GPU Backend: Available");
        }
        else
        {
            Debug.Log("- GPU Backend: Not available (no compute shader support)");
        }
    }
}
```

Step 3: Project Settings Configuration

1. **Graphics API Settings**:

    - Go to **Edit > Project Settings > Player > Other Settings**
    - Ensure **Compute Shaders** are supported for GPU inference
    - For mobile: Use **Vulkan** or **OpenGL ES 3.1+**
2. **Quality Settings**:

    - For better performance, consider lowering graphics quality for AI-heavy applications

- Balance visual quality with AI processing power

Project Structure & Basic Configuration

Recommended Folder Structure

```
Assets/
├── Models/              # Store your ONNX model files here
├── Scripts/
│   ├── AI/              # AI-related scripts
│   └── Core/            # Core game scripts
├── Textures/
│   └── TestImages/      # Sample images for testing
├── Scenes/
│   ├── MainScene.unity
│   └── TestScenes/      # Individual test scenes for each sample
└── StreamingAssets/     # For runtime model loading (optional)
```

Basic Sentis Workflow

Every Sentis application follows this pattern:

```
using Unity.Sentis;

public class BasicSentisTemplate : MonoBehaviour
{
    [Header("Model Configuration")]
    public ModelAsset modelAsset;          // Drag your .onnx model here
    public BackendType backendType = BackendType.GPUCompute;

    private Model runtimeModel;
    private IWorker worker;

    void Start()
    {
        // 1. Load the model
        runtimeModel = ModelLoader.Load(modelAsset);

        // 2. Create a worker (CPU or GPU)
        worker = WorkerFactory.CreateWorker(backendType, runtimeModel);
    }
```

```
    void Update()
    {
        // 3. Prepare input data
        // 4. Execute the model
        // 5. Get the output
        // 6. Process results
    }

    void OnDestroy()
    {
        // Always clean up resources
        worker?.Dispose();
    }
}
```

Sample Projects Overview

The inference-engine-samples repository contains the following sample projects: Digit Recognition, Board-game AI, Depth Estimation, Star Simulation, Protein Folding (only compatible with Sentis 1.X).

1. Digit Recognition Sample

- **Purpose**: Runs a handwritten digit (number) detection AI model called MNIST which can read written numbers and identify the most probable number drawn
- **Model**: MNIST neural network
- **Use Case**: Number recognition in games, educational apps
- **Difficulty**: ⭐ Beginner

2. Board Game AI Sample

- **Purpose**: Uses Sentis to build a bot opponent for a board game called Othello, where the game has configurable difficulty. It runs a neural network trained on the game rules and determines game win probabilities after each move
- **Model**: Game strategy neural network
- **Use Case**: AI opponents, strategy games
- **Difficulty**: ⭐⭐ Intermediate

3. Depth Estimation Sample

- **Purpose**: Showcases how to integrate Sentis into an augmented reality (AR) experience. It uses a depth estimation neural network to allow real-world objects to occlude objects in the game scene
- **Model**: Depth estimation CNN
- **Use Case**: AR applications, camera-based games
- **Difficulty**: ⭐⭐⭐ Advanced

4. Star Simulation Sample

- **Purpose**: Uses Inference Engine as a tensor-based linear algebra engine to simulate a physical system in real time. Tackles the N-body problem with each star represented by an emissive sphere
- **Model**: Physics simulation using tensors
- **Use Case**: Scientific visualization, physics simulations
- **Difficulty**: ⭐⭐⭐ Advanced

5. Protein Folding Sample

- **Purpose**: A simplified recreation of Alphafold (version 1) which won the CASP (2018) protein folding competition. Demonstrates the Unity Sentis API and how it can be useful for scientific visualisation
- **Model**: AlphaFold-inspired protein folding
- **Use Case**: Scientific visualization, educational applications
- **Difficulty**: ⭐⭐⭐⭐ Expert

Detailed Sample Project Tutorials

1. Digit Recognition Sample - Complete Setup

This sample demonstrates the fundamentals of Unity Sentis using the classic MNIST dataset.

Step 1: Download and Setup

1. **Get the MNIST Model**:
    - Download from Hugging Face Unity Models
    - Place the `.onnx` file in your `Assets/Models/` folder
2. **Create the Scene**:
    - Create a new scene: **File > New Scene**
    - Add a Canvas: **GameObject > UI > Canvas**
    - Add a Raw Image for displaying drawings

- Add a Text component for showing results

Step 2: Core Script Implementation

```
using UnityEngine;
using Unity.Sentis;
using Unity.Sentis.Layers;

public class DigitClassifier : MonoBehaviour
{
    [Header("Model Configuration")]
    public ModelAsset modelAsset;
    public BackendType backendType = BackendType.GPUCompute;

    [Header("UI References")]
    public UnityEngine.UI.RawImage drawingImage;
    public UnityEngine.UI.Text resultText;

    [Header("Drawing Settings")]
    public int textureSize = 28; // MNIST expects 28x28 images
    public Color drawColor = Color.white;
    public float brushSize = 2f;

    private Model runtimeModel;
    private IWorker worker;
    private Texture2D drawingTexture;
    private bool isDrawing = false;

    void Start()
    {
        SetupModel();
        SetupDrawingTexture();
    }

    void SetupModel()
    {
        // Load the MNIST model
        runtimeModel = ModelLoader.Load(modelAsset);

        // Add softmax layer for probability output
        string softmaxOutputName = "Softmax_Output";
        runtimeModel.AddLayer(new Softmax(softmaxOutputName, runtimeModel.outputs[0]));
        runtimeModel.outputs[0] = softmaxOutputName;

        // Create worker
```

```
    worker = WorkerFactory.CreateWorker(backendType, runtimeModel);

    Debug.Log("MNIST model loaded successfully!");
}

void SetupDrawingTexture()
{
    // Create a texture for drawing
    drawingTexture = new Texture2D(textureSize, textureSize, TextureFormat.RGB24, false);
    ClearTexture();
    drawingImage.texture = drawingTexture;
}

void ClearTexture()
{
    // Fill with black (background)
    Color[] pixels = new Color[textureSize * textureSize];
    for (int i = 0; i < pixels.Length; i++)
    {
        pixels[i] = Color.black;
    }
    drawingTexture.SetPixels(pixels);
    drawingTexture.Apply();
}

void Update()
{
    HandleDrawingInput();
}

void HandleDrawingInput()
{
    // Simple mouse/touch drawing
    if (Input.GetMouseButton(0))
    {
        Vector2 mousePos = Input.mousePosition;
        RectTransform rectTransform = drawingImage.rectTransform;

        if (RectTransformUtility.ScreenPointToLocalPointInRectangle(
            rectTransform, mousePos, null, out Vector2 localPoint))
        {
            // Convert to texture coordinates
            Vector2 normalizedPoint = new Vector2(
                (localPoint.x + rectTransform.rect.width * 0.5f) / rectTransform.rect.width,
```

```
            (localPoint.y + rectTransform.rect.height * 0.5f) / rectTransform.rect.height
        );

        if (normalizedPoint.x >= 0 && normalizedPoint.x <= 1 &&
            normalizedPoint.y >= 0 && normalizedPoint.y <= 1)
        {
            DrawOnTexture(normalizedPoint);
        }
    }
  }
  else if (Input.GetMouseButtonUp(0) && isDrawing)
  {
    isDrawing = false;
    ClassifyDrawing();
  }
}

void DrawOnTexture(Vector2 normalizedPosition)
{
  isDrawing = true;

  int x = Mathf.FloorToInt(normalizedPosition.x * textureSize);
  int y = Mathf.FloorToInt(normalizedPosition.y * textureSize);

  // Draw with brush size
  for (int dx = -Mathf.FloorToInt(brushSize); dx <= brushSize; dx++)
  {
    for (int dy = -Mathf.FloorToInt(brushSize); dy <= brushSize; dy++)
    {
      int pixelX = Mathf.Clamp(x + dx, 0, textureSize - 1);
      int pixelY = Mathf.Clamp(y + dy, 0, textureSize - 1);

      if (dx * dx + dy * dy <= brushSize * brushSize)
      {
        drawingTexture.SetPixel(pixelX, pixelY, drawColor);
      }
    }
  }

  drawingTexture.Apply();
}

public void ClassifyDrawing()
{
```

```
    // Convert texture to tensor
    using var inputTensor = TextureConverter.ToTensor(
        drawingTexture,
        width: textureSize,
        height: textureSize,
        channels: 1
    );

    // Execute the model
    worker.Execute(inputTensor);

    // Get output
    using var outputTensor = worker.PeekOutput() as TensorFloat;
    outputTensor.MakeReadable();

    float[] probabilities = outputTensor.ToReadOnlyArray();

    // Find the digit with highest probability
    int predictedDigit = 0;
    float maxProbability = probabilities[0];

    for (int i = 1; i < probabilities.Length; i++)
    {
        if (probabilities[i] > maxProbability)
        {
            maxProbability = probabilities[i];
            predictedDigit = i;
        }
    }

    // Update UI
    resultText.text = $"Predicted Digit: {predictedDigit}\\nConfidence: {maxProbability:P1}";

    Debug.Log($"Classified digit: {predictedDigit} with confidence: {maxProbability:P2}");
}

public void ClearDrawing()
{
    ClearTexture();
    resultText.text = "Draw a digit (0-9)";
}

void OnDestroy()
{
```

```
        worker?.Dispose();
    }
}
```

Step 3: UI Setup

1. **Drawing Canvas**:
    - Set Canvas **Render Mode** to **"Screen Space - Overlay"**
    - Scale appropriately for your target device
2. **Drawing Area**:
    - Create a **Raw Image** component
    - Set size to at least 200x200 pixels
    - Add the **DigitClassifier** script to a GameObject
    - Assign the Raw Image to the **drawingImage** field
3. **Result Display**:
    - Add a **Text** component for showing classification results
    - Assign to the **resultText** field
4. **Clear Button**:
    - Add a **Button** that calls `ClearDrawing()` method

2. Depth Estimation Sample - AR Integration

This sample demonstrates how to use Sentis for real-time depth estimation in AR applications.

Core Concept

This sample showcases how to integrate Sentis into an augmented reality (AR) experience. It uses a depth estimation neural network to allow real-world objects to occlude objects in the game scene. The depth is determined by processing video frames from the camera.

Implementation Example

```
using UnityEngine;
using Unity.Sentis;
using UnityEngine.Rendering;

public class DepthEstimation : MonoBehaviour
{
    [Header("Model Configuration")]
    public ModelAsset depthModel;
    public BackendType backendType = BackendType.GPUCompute;
```

```
[Header("Camera Setup")]
public Camera mainCamera;
public Material depthMaterial;

private Model runtimeModel;
private IWorker worker;
private WebCamTexture webCamTexture;
private RenderTexture depthTexture;

void Start()
{
    SetupModel();
    SetupCamera();
}

void SetupModel()
{
    var model = ModelLoader.Load(depthModel);

    // Add post-processing for depth normalization
    var graph = new FunctionalGraph();
    var inputs = graph.AddInputs(model);
    var outputs = Functional.Forward(model, inputs);
    var output = outputs[0];

    // Normalize depth values between 0 and 1
    var max0 = Functional.ReduceMax(output, new[] { 1, 2 }, false);
    var min0 = Functional.ReduceMin(output, new[] { 1, 2 }, false);
    output = (output - min0) / (max0 - min0);

    runtimeModel = graph.Compile(output);
    worker = WorkerFactory.CreateWorker(backendType, runtimeModel);
}

void SetupCamera()
{
    // Initialize webcam
    webCamTexture = new WebCamTexture();
    webCamTexture.Play();

    // Create render texture for depth output
    depthTexture = new RenderTexture(256, 256, 0, RenderTextureFormat.R8);
}
```

```
    void Update()
    {
        if (webCamTexture.isPlaying)
        {
            ProcessDepthEstimation();
        }
    }

    void ProcessDepthEstimation()
    {
        // Convert webcam texture to tensor
        using var inputTensor = TextureConverter.ToTensor(
            webCamTexture,
            width: 256,
            height: 256,
            channels: 3
        );

        // Execute depth estimation
        worker.Execute(inputTensor);

        // Get depth output
        using var outputTensor = worker.PeekOutput() as TensorFloat;

        // Convert back to texture
        TextureConverter.RenderToTexture(outputTensor, depthTexture);

        // Apply depth to material
        depthMaterial.SetTexture("_DepthTex", depthTexture);
    }

    void OnDestroy()
    {
        worker?.Dispose();
        if (webCamTexture != null)
        {
            webCamTexture.Stop();
        }
    }
}
```

3. Board Game AI Sample - Strategic AI

This sample creates an AI opponent for the Othello board game.

Key Features

- Neural network trained on game rules
- Configurable difficulty levels
- Real-time move prediction
- Win probability calculation

Basic Implementation Structure

```
using Unity.Sentis;

public class OthelloAI : MonoBehaviour
{
    [Header("AI Configuration")]
    public ModelAsset othelloModel;
    public float difficultyLevel = 0.7f; // 0 = easy, 1 = hard

    private Model runtimeModel;
    private IWorker worker;
    private int[,] gameBoard = new int[8, 8];

    public Vector2Int GetBestMove(int[,] currentBoard, int playerTurn)
    {
        // Convert board state to tensor
        using var boardTensor = ConvertBoardToTensor(currentBoard, playerTurn);

        // Execute AI model
        worker.Execute(boardTensor);

        // Get move probabilities
        using var outputTensor = worker.PeekOutput() as TensorFloat;
        outputTensor.MakeReadable();

        var moveScores = outputTensor.ToReadOnlyArray();

        // Apply difficulty scaling
        return SelectMoveWithDifficulty(moveScores, difficultyLevel);
    }

    private TensorFloat ConvertBoardToTensor(int[,] board, int player)
    {
        var data = new float[8 * 8 * 3]; // Board + player info

        for (int x = 0; x < 8; x++)
```

```
        {
            for (int y = 0; y < 8; y++)
            {
                int index = x * 8 + y;
                data[index] = board[x, y]; // Current piece
                data[index + 64] = (board[x, y] == player) ? 1f : 0f; // Own pieces
                data[index + 128] = (board[x, y] == -player) ? 1f : 0f; // Enemy pieces
            }
        }

        return new TensorFloat(new TensorShape(1, 3, 8, 8), data);
    }

    private Vector2Int SelectMoveWithDifficulty(float[] moveScores, float difficulty)
    {
        // Implementation depends on your difficulty algorithm
        // Higher difficulty = always pick best move
        // Lower difficulty = sometimes pick suboptimal moves

        if (Random.value < difficulty)
        {
            // Pick best move
            return GetBestMoveFromScores(moveScores);
        }
        else
        {
            // Pick random valid move
            return GetRandomValidMove(moveScores);
        }
    }
}
```

Creating Your Own AI Models

Supported Model Formats

To make sure the exported model is compatible with Sentis, set the ONNX opset version to 15. Unity Sentis supports:

- **ONNX format** (primary)
- **Converted from**: PyTorch, TensorFlow, Keras, scikit-learn

- **Hugging Face models**: Pre-validated models optimized for Unity Sentis available on Hugging Face Hub

Converting PyTorch Models to ONNX

```
import torch
import torch.onnx

# Your trained PyTorch model
model = YourTrainedModel()
model.eval()

# Create dummy input with the expected shape
dummy_input = torch.randn(1, 3, 224, 224)  # Batch, Channels, Height, Width

# Export to ONNX
torch.onnx.export(
    model,                      # Model to export
    dummy_input,                # Model input
    "your_model.onnx",          # Output file
    export_params=True,         # Store trained parameters
    opset_version=15,           # ONNX opset version (required for Sentis)
    do_constant_folding=True,   # Optimize constant folding
    input_names=['input'],      # Input tensor name
    output_names=['output'],    # Output tensor name
    dynamic_axes={
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

Converting TensorFlow Models to ONNX

Converting TensorFlow models requires the tf2onnx package:

```
import tensorflow as tf
import tf2onnx

# Load your TensorFlow model
model = tf.keras.models.load_model('your_model.h5')

# Convert to ONNX
spec = (tf.TensorSpec((None, 224, 224, 3), tf.float32, name="input"),)
model_proto, _ = tf2onnx.convert.from_keras(
```

```
    model,
    input_signature=spec,
    opset=15
)

# Save ONNX model
with open("your_model.onnx", "wb") as f:
    f.write(model_proto.SerializeToString())
```

Integration with Unity Features

Performance Optimization

Backend Selection

Sentis is generally faster with IL2CPP than with Mono for Standalone builds. Choose the right backend:

```
public class BackendOptimization : MonoBehaviour
{
    void Start()
    {
        BackendType optimalBackend = GetOptimalBackend();
        Debug.Log($"Using backend: {optimalBackend}");
    }

    BackendType GetOptimalBackend()
    {
        // Check platform and capabilities
        if (Application.platform == RuntimePlatform.WebGLPlayer)
        {
            return BackendType.CPU; // WebGL doesn't support compute shaders
        }

        if (SystemInfo.supportsComputeShaders)
        {
            // Check if we have a powerful GPU
            if (SystemInfo.graphicsMemorySize > 2048) // 2GB+ VRAM
            {
                return BackendType.GPUCompute;
            }
        }
```

```
        return BackendType.CPU; // Fallback to CPU
    }
}
```

Asynchronous Processing

For real-time applications, process AI inference asynchronously:

```
using System.Collections;
using Unity.Sentis;

public class AsyncInference : MonoBehaviour
{
    private IWorker worker;
    private bool isProcessing = false;

    public void ProcessAsync(TensorFloat inputTensor)
    {
        if (!isProcessing)
        {
            StartCoroutine(ProcessInferenceCoroutine(inputTensor));
        }
    }

    IEnumerator ProcessInferenceCoroutine(TensorFloat inputTensor)
    {
        isProcessing = true;

        // Execute on worker
        worker.Execute(inputTensor);

        // Wait for completion (non-blocking)
        yield return new WaitForEndOfFrame();

        // Get results
        using var output = worker.PeekOutput() as TensorFloat;
        output.MakeReadable();

        // Process results
        ProcessResults(output.ToReadOnlyArray());

        isProcessing = false;
    }
```

```
    void ProcessResults(float[] results)
    {
        // Handle your results here
        Debug.Log($"Inference complete: {results.Length} outputs");
    }
}
```

Memory Management

Always properly dispose of tensors and workers:

```
public class ProperResourceManagement : MonoBehaviour
{
    private IWorker worker;
    private List<TensorFloat> activeTensors = new List<TensorFloat>();

    void ProcessData()
    {
        // Create tensor
        var tensor = new TensorFloat(new TensorShape(1, 224, 224, 3), data);
        activeTensors.Add(tensor);

        // Use tensor...
        worker.Execute(tensor);

        // Clean up immediately if not needed
        tensor.Dispose();
        activeTensors.Remove(tensor);
    }

    void OnDestroy()
    {
        // Clean up all resources
        foreach (var tensor in activeTensors)
        {
            tensor?.Dispose();
        }
        activeTensors.Clear();

        worker?.Dispose();
    }
}
```

Troubleshooting Guide

Common Issues and Solutions

Issue 1: "Tensor data cannot be read from" Error

Problem: InvalidOperationException: Tensor data cannot be read from, use .ReadbackAndClone() to allow reading from tensor

Solution:

```
// Wrong way:
var output = worker.PeekOutput() as TensorFloat;
float[] data = output.ToReadOnlyArray(); // This will fail

// Correct way:
var output = worker.PeekOutput() as TensorFloat;
output.MakeReadable(); // Add this line!
float[] data = output.ToReadOnlyArray(); // Now it works
```

Issue 2: GPU Backend Not Working

Problem: GPU compute shaders not supported or failing

Solution:

```
void Start()
{
    BackendType backend = BackendType.GPUCompute;

    // Check if GPU compute is supported
    if (!SystemInfo.supportsComputeShaders)
    {
        Debug.LogWarning("GPU Compute not supported, falling back to CPU");
        backend = BackendType.CPU;
    }

    try
    {
        worker = WorkerFactory.CreateWorker(backend, runtimeModel);
    }
```

```
    catch (System.Exception e)
    {
        Debug.LogError($"Failed to create {backend} worker: {e.Message}");
        Debug.Log("Falling back to CPU backend");
        worker = WorkerFactory.CreateWorker(BackendType.CPU, runtimeModel);
    }
}
```

Issue 3: Model Loading Failures

Problem: ONNX model won't load or gives validation errors

Solutions:

1. **Check ONNX Opset Version**:
    - Ensure your model uses **ONNX opset version 15**
    - Re-export with correct opset if needed

Validate Model:

```
 import onnx

# Load and check model
model = onnx.load("your_model.onnx")
onnx.checker.check_model(model)
print("Model is valid!")
```

2.
3. **Supported Operators**:
    - Check if all operators in your model are supported
    - Refer to the Sentis supported ONNX operators documentation

Issue 4: Poor Performance

Performance Optimization Checklist:

1. **Use Appropriate Backend**:
    - For most models generated with the ML-Agents Toolkit, CPU will be faster than GPU. Use GPU only if you use the ResNet visual encoder or have a large number of agents with visual observations
2. **Optimize Model Size**:

- Use smaller models when possible
- Consider model quantization
- Reduce input resolution if applicable

Batch Processing:

```
// Process multiple inputs at once instead of one by one
var batchedInput = CreateBatchedTensor(multipleInputs);
worker.Execute(batchedInput);
```

3.

Frame Rate Management:

```
public class FrameRateOptimization : MonoBehaviour
{
    public int targetFrameRate = 60;
    public int inferenceFrequency = 10; // Run AI every 10 frames

    private int frameCount = 0;

    void Update()
    {
        frameCount++;

        if (frameCount % inferenceFrequency == 0)
        {
            RunInference();
        }
    }
}
```

4.

Platform-Specific Issues

WebGL Platform

In the Editor, It is not possible to use Sentis with GPU device selected when Editor Graphics Emulation is set to OpenGL(ES) 3.0 or 2.0 emulation

- Use **BackendType.CPU** only
- Reduce model complexity for better performance
- Consider using WebAssembly builds

Mobile Platforms

1. **Memory Constraints**:
    - Use smaller models
    - Dispose of tensors immediately after use
    - Monitor memory usage
2. **Thermal Throttling**:
    - Reduce inference frequency when device gets hot
    - Use CPU backend for less heat generation

Build Errors

There might be non-fatal build time errors when target platform includes Graphics API that does not support Unity Compute Shaders

- These are usually warnings, not critical errors
- Ensure fallback to CPU backend is implemented

Advanced Topics

Model Chaining and Complex Workflows

You can chain multiple models for complex AI workflows:

```
public class ModelChaining : MonoBehaviour
{
    [Header("Models")]
    public ModelAsset preprocessorModel;
    public ModelAsset mainModel;
    public ModelAsset postprocessorModel;

    private IWorker[] workers = new IWorker[3];

    void Start()
    {
        // Load all models
        workers[0] = WorkerFactory.CreateWorker(BackendType.CPU,
ModelLoader.Load(preprocessorModel));
        workers[1] = WorkerFactory.CreateWorker(BackendType.GPUCompute,
ModelLoader.Load(mainModel));
```

```
        workers[2] = WorkerFactory.CreateWorker(BackendType.CPU,
ModelLoader.Load(postprocessorModel));
    }

    public float[] ProcessWithChain(float[] rawInput)
    {
        // Step 1: Preprocessing
        using var input1 = new TensorFloat(new TensorShape(1, rawInput.Length), rawInput);
        workers[0].Execute(input1);
        using var preprocessed = workers[0].PeekOutput() as TensorFloat;

        // Step 2: Main processing
        workers[1].Execute(preprocessed);
        using var mainOutput = workers[1].PeekOutput() as TensorFloat;

        // Step 3: Postprocessing
        workers[2].Execute(mainOutput);
        using var finalOutput = workers[2].PeekOutput() as TensorFloat;
        finalOutput.MakeReadable();

        return finalOutput.ToReadOnlyArray();
    }
}
```

Custom Tensor Operations

For advanced use cases, you can perform custom tensor operations:

```
using Unity.Sentis;

public class CustomTensorOps : MonoBehaviour
{
    void ExampleTensorOperations()
    {
        // Create custom tensors
        var tensor1 = new TensorFloat(new TensorShape(2, 3), new float[] {1, 2, 3, 4, 5, 6});
        var tensor2 = new TensorFloat(new TensorShape(2, 3), new float[] {2, 2, 2, 2, 2, 2});

        // Perform operations using Ops
        var ops = WorkerFactory.CreateOps(BackendType.CPU, null);

        // Add tensors
        using var result = ops.Add(tensor1, tensor2);
```

```
        result.MakeReadable();

        Debug.Log($"Addition result: [{string.Join(", ", result.ToReadOnlyArray())}]");

        // Clean up
        tensor1.Dispose();
        tensor2.Dispose();
        ops.Dispose();
    }
}
```

Integration with Unity ML-Agents

If you're using ML-Agents trained models:

```
// For ML-Agents models, use specific conventions
public class MLAgentsIntegration : MonoBehaviour
{
    public ModelAsset mlAgentsModel;

    void Start()
    {
        var model = ModelLoader.Load(mlAgentsModel);

        // ML-Agents models expect specific tensor names
        // Check TensorNames.cs in ML-Agents for details
        var worker = WorkerFactory.CreateWorker(BackendType.CPU, model);
    }
}
```

Conclusion

Unity Sentis opens up incredible possibilities for integrating AI into your Unity projects. Whether you're creating intelligent NPCs, implementing computer vision features, or building educational tools, Sentis provides the performance and flexibility you need.

Key Takeaways

1. **Start Simple**: Begin with the digit recognition sample to understand the basics
2. **Choose the Right Backend**: CPU is often faster for smaller models, GPU for complex vision tasks

3. **Manage Resources**: Always dispose of tensors and workers properly
4. **Optimise for Platform**: Consider your target platform's capabilities and limitations
5. **Use Community Resources**: Leverage Hugging Face models validated for Unity Sentis

Next Steps

- Experiment with different sample projects
- Try integrating pre-trained models from Hugging Face
- Create your own custom AI features
- Join the Unity AI community for support and inspiration

Additional Resources

- **Official Documentation**: Unity Sentis Package Documentation
- **Hugging Face Unity Models**: Pre-validated Sentis Models
- **Sample Repository**: Unity Sentis Samples on GitHub
- **Community Forum**: Unity AI Forum

Happy AI Development! 🤖✨