Neural Network Engine Overview

An overview of Unreal Engine's neural network engine.

Learn to use this **Beta** feature, but use caution when shipping with it.

The **Neural Network Engine (NNE)** provides a common API to access different neural network runtimes (sometimes referred to as execution providers), and to evaluate neural networks without the need of runtime-specific coding. Unreal Engine includes individual runtimes that you can add to your project by enabling their corresponding plugins. You can dynamically choose an optimal runtime to create and execute a neural network from an NNE asset from a project, depending on the model and target hardware you are using.

NNE provides different interfaces that are implemented by runtimes to cover different use cases. Each of these use cases may require a different way of running a neural network. These interfaces determine whether to run on the CPU, on the GPU, or on GPU aligned with rendering a frame. This implementation provides you with the maximum control over whether and how a neural network model is executed in your project.

You can use NNE to run real-time inference to augment a game with artificial intelligence (AI) and to implement editor-based features as asset operations, queries, and artist-assisting tools.

These are the key elements that make up NNE:

- **Interfaces** are the accessible parts of NNE runtime plugins.
- **Runtimes** are the plugins used to implement one or more interfaces in your project.
- **Assets** are the neural network model assets which can be imported into Unreal Engine when a runtime is enabled.
- **Models** are the data stored within neural network model assets.

Interfaces

Interfaces are the accessible parts of a NNE runtime plugin. To keep the API clean and descriptive, instead of accessing a runtime directly, you use an Interface class that the runtime implements, and that interface exposes the API for that runtime. Each interface is intended for a single, clearly-defined use-case, and runtimes implement interfaces based on whether or not they are capable of evaluating that type of use-case.

An example use case would be invoking a synchronous inference call on CPU to align the evaluation of a model with a frame rendering on the GPU using `INNERuntimeCPU` and `INNERuntimeGPU`. The `INNERuntimeRDG` is enqueued asynchronously using Render Dependency Graph (FRDGBuilder). NNE exposes different interfaces with each designed to cover a single kind of use case in an effort to keep the API clean and descriptive.

This consistency ensures that runtime developers can focus on supporting the corresponding functionality for each interface. New interfaces can be added in the future to cover new kinds of use cases, while maintaining backward compatibility.

The following interfaces are available:

Interface Name	Description
`INNERuntimeCPU`	This interface covers all use cases where inference should happen on the CPU. Input and output tensors are provided on the CPU and there is no memory transfer to another device. It is suited for use cases when there is no budget on the GPU, or when synchronization with GPU memory does not justify the speedup in computation. Model instances created by this interface can run on the game thread (synchronous), as an async task, or on any other thread as long as the caller takes care of thread safety and memory lifetime of the model, including both input and output memory.
`INNERuntimeGPU`	This interface covers runtimes that evaluate neural networks on the GPU. Input and output tensors are provided as CPU memory, and require synchronization with the GPU, where the runtime will upload and download them. Inference happens independent of the rendering of a frame, but will compete with the rendering pipeline for GPU resources. This interface typically serves editor-only use cases where the additional GPU synchronization and resource competition does not affect performance. Similarly to `INNERuntimeCPU`, model instances created by this interface can be run from any thread as long as the caller takes care of thread safety and memory lifetime.
`INNERuntimeRDG`	This interface covers the evaluation of neural networks as part of the Render Dependency Graph (RDG), which adds the model evaluation to the provided RDG Graph Builder. This interface is used when a neural network consumes and produces resources being used while rendering a frame. Inference is carried out on the GPU, and input and output tensors have to be provided as RDG buffers.The inference call is invoked from the render thread, while the model creation and setup is typically done on the game thread. This allows for tight integration with the engine resources.

Runtimes

Runtimes are plugins that implement NNE Interfaces. Runtimes typically register themselves with the NNE at startup.

Use the function `TArray<FString>UE::NNE::GetAllRuntimeNames()` to get the names of all available runtimes independent of what interfaces they implement.

Use the templated function `TArray<FString>UE::NNE::GetAllRuntimeNames<T>()` to get a pre-filtered list of runtime names that implement the specified interface. For example, to get all runtimes that implement interfaces that happen on the CPU, use `UE::NNE::GetAllRuntimeNames<INNERuntimeCPU>()`.

To retrieve a runtime, use the function `TWeakInterfacePtr<INNERuntime>UE::NNE::GetRuntime(const FString& Name)`. Or, retrieve the corresponding templated function with `TWeakInterfacePtr<T>UE::NNE::GetRuntime<T>(const FString& Name)`.

Not all runtimes are available on all platforms, even when the corresponding plugin is enabled. If a runtime is not available, or if it is available but does not implement the interface passed in the templated function, the returned weak pointer will be null. As runtimes can unload themselves, you should run a test for validity of the weak pointer before using it.

Runtimes typically register, unregister, load, and unload themselves along with their related plugin and module. However, runtime’s lifetime and registration is up to its specific implementation.

Assets

NNE plugins enable you to import neural network model files directly into Unreal Engine when a Runtime plugin supports that file type. Unreal Engine creates **NNE Model Data** assets in the Content Browser for imported neural network models.

Not all runtimes support all file formats. File types can show as successfully imported, but creating the corresponding model for that specific runtime can still fail. Refer to the runtime you are using for the file formats it supports.

On successful import, the engine creates a **UNNEModelData** asset. Open the asset from the Content Browser to enable and disable a model for specific implemented runtimes in your project. Removing unwanted runtimes increases packaging speed and decreases package size of your project. Each model is optimized for each runtime by default, but it is a good practice to only select runtimes with models you're planning to use.

Unreal Engine loads NNE model data assets are the same way as other UE assets. For example, a public class variable of type `UNNEModelData` with the UPROPERTY decorator is defined inside an actor, and a model is assigned to it in the editor is loaded automatically when the actor is spawned. Alternatively, the Unreal Engine function `LoadObject` can be used to load the asset programmatically if the content path is known.

Models

A **Model** is contained within an interface and implemented by a runtime with a loaded UNNEModelData asset. You can create the model from the asset using the following functions:

- `TSharedPtr<UE::NNE:IModelCPU> INNERuntimeCPU::CreateModelCPU(const TObjectPtr<UNNEModelData> ModelData)`
- `TSharedPtr<UE::NNE:IModelGPU> INNERuntimeGPU::CreateModelGPU(const TObjectPtr<UNNEModelData> ModelData)`
- `TSharedPtr<UE::NNE:IModelRDG> INNERuntimeRDG::CreateModelRDG(const TObjectPtr<UNNEModelData> ModelData)`

Not every runtime can create a model from any NNE model data asset. Runtimes are available inside the Unreal Editor as a requirement for cooking, but a model may be unable to run inference on the current platform. You can use the following functions to return a status indicating whether a model can be created or not:

- `ECanCreateModelCPUStatus INNERuntimeCPU::CanCreateModelCPU(const TObjectPtr<UNNEModelData> ModelData) const`
- `ECanCreateModelGPUStatus INNERuntimeGPU::CanCreateModelGPU(const TObjectPtr<UNNEModelData> ModelData) const`
- `ECanCreateModelRDGStatus INNERuntimeRDG::CanCreateModelRDG(const TObjectPtr<UNNEModelData> ModelData) const`

You should call these functions before model creation. Even if the result indicates that a model can be created, the actual creation can still fail due to internal errors. You should always check the shared pointer to the returned model for validity before using it.

The created model typically contains immutable buffers as model weights and parameters that are shared among different instances of the model. You can release UNNEModelData after model creation since the models will internally make sure they keep pointers to required data.

Model Instances

You can run inference by creating a **Model Instance** from a neural network model through one of the following functions:

- `TSharedPtr<UE::NNE::IModelInstanceCPU> UE::NNE::IModelCPU::CreateModelInstanceCPU()`
- `TSharedPtr<UE::NNE::IModelInstanceGPU> UE::NNE::IModelGPU::CreateModelInstanceGPU()`
- `TSharedPtr<UE::NNE::IModelInstanceRDG> UE::NNE::IModelRDG::CreateModelInstanceRDG()`

Model instances typically contain session specific data as internal states and intermediate buffers. You can create multiple instances from a single model that shares the model's immutable data as weights and parameters. The model can then be released after the instances have been created since the instances maintain their own reference to any required shared data.

Before the first inference call, you must call the following function to allow the model to allocate properly sized internal buffers:

```
UE::NNE::ESetInputTensorShapesStatus
SetInputTensorShapes(TConstArrayView<UE::NNE::FTensorShape> InInputShapes)

```

Copy full snippet

A runtime reports any potential errors that could not be caught at import or cook time, and you should check the returned status code. The function must be called again whenever any input shape is changed. Avoid unnecessary repeated calls to keep the computing resource overhead low.

The caller owns input and output memory of any NNE model instance. The caller must make sure that the memory of any input and output tensor remains valid and untouched through the full inference call. This requires special care in threaded use cases.

We recommend batching of input data since it's more performant to evaluate a batch than to run multiple individual calls when needing to evaluate a model instance multiple times per tick or frame. if it is not possible for synchronization of multiple calls through batching, multiple instances can run inference concurrently without violating the thread safety required by NNE.

Minimal NNE Example

You can use the following example of required points and code snippets to get started using a neural network on the CPU.

For simplicity and readability, this example does not go into detail about how to prepare and set up input and output tensors. This example also does not check any results, which you should do in a real use case.

For additional information on getting started, see Neural Network Engine Quick Start Guide.

To get started:

1. Enable a NNE Runtime plugin from the **Plugins** browser in Unreal Engine.
2. Add the module `NNE` as a dependency to the project's `.Build.cs` file.
3. Import a neural network file (such as `*.onnx` file type) to Unreal Engine through the Content Browser.

Load and execute the model with the following code:

```
// Include the NNE headers
#include "NNE.h"
#include "NNERuntimeCPU.h"
```

```cpp
#include “NNEModelData.h”

// Create the model from a neural network model data asset
TObjectPtr<UNNEModelData> ModelData = LoadObject<UNNEModelData>(GetTransientPackage(), TEXT("/path/to/asset"));
TWeakInterfacePtr<INNERuntimeCPU> Runtime = UE::NNE::GetRuntime<INNERuntimeCPU>(FString("NNERuntimeORTCpu"));
TSharedPtr<UE::NNE::IModelInstanceCPU> ModelInstance = Runtime->CreateModelCPU(ModelData)->CreateModelInstanceCPU();

// TODO: Setup input and output tensors and tensor bindings as well as corresponding input shapes

// Prepare the model given a certain input size
ModelInstance->SetInputTensorShapes(InputShapes);

// Run the model passing caller owned CPU memory
ModelInstance->RunSync(Inputs, Outputs);
```

Copy full snippet