Noedify

Introduction

Noedify is a Unity plugin that aims to make AI and neural networks accessible to game developers within the Unity environment. Noedify offers a high-level API similar to TensorFlow APIs such as Keras.

With Noedify you can:

- Build deep neural networks inside your Unity project
- Train fully-connected (dense) or 2D convolutional neural networks during runtime
- Utilize Unity's Jobs System to train in background processes, taking advantage of multiple CPU threads
- Import trained models from TensorFlow Keras to be used in your Unity project or to be further trained for model fine-tuning

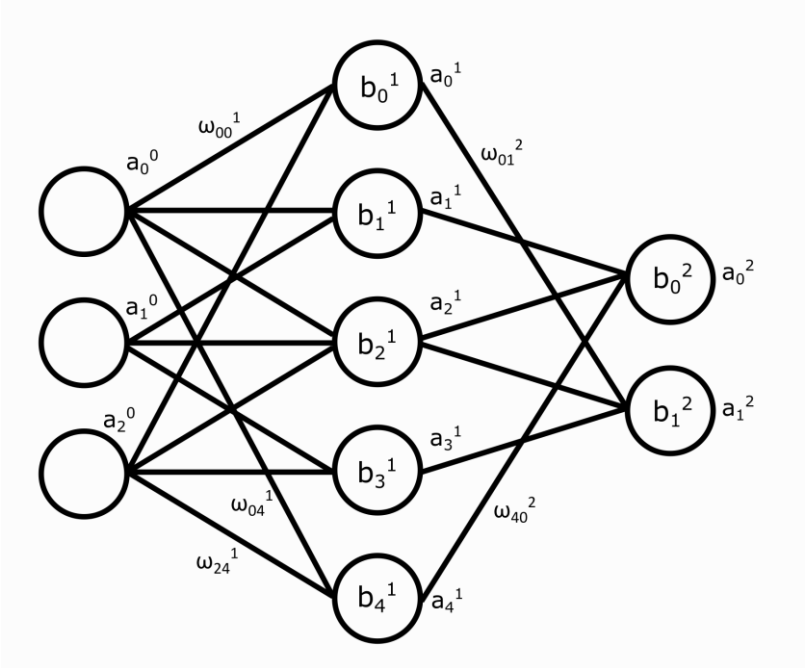
The Noedify package includes the following demos:

1. **Digit Drawing and Recognition Demo:** Simple example of training either a fully-connected (dense) or 2D convolutional network to recognize handwritten digits.
2. **Hoppy Game:** Train an AI to navigate a simple 2D environment, avoiding barriers to survive as long as possible.
3. **Fully-Connected and Convolutional Live Training Demo (handwritten digits):** Import a model trained using Tensorflow Keras to recognise handwritten digits in real time.

Background

The following is a brief summary of how the simplest neural networks function.

Neural Networks



Each node in a neural network operates with a simple equation:

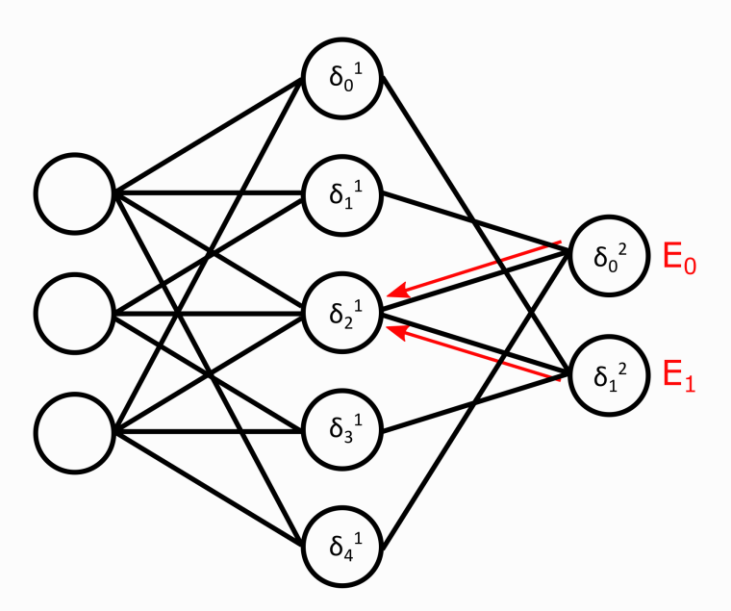
$$a_j^l = \sigma(\, a_0^{l-1}\omega_{0j}^l + a_1^{l-1}\omega_{1j}^{l-1} + \cdots + \; a_i^{l-1}\omega_{ij}^{l-1} + b_j^l)$$

where σ is an activation function (or flattening function). Each node's output is determined by the outputs from the previous layer and the node's connection weights ω_{ij}^l and bias b_j^l. The inputs to the network are set to be the outputs of the first layer, and the outputs of the network are the outputs of the last layer.

The goal in training a network is to determine the set of weights and biases that will produce the desired network output given an input.

For example: determining the set of weights and biases to produce a network where the inputs are a photo of a cat or dog, and the output is a '1' if the photo is a dog or '0' if it's a cat.

Training Neural Networks



Finding the correct weights and biases requires a lot of trial and error. When you have a training set of data with the input sets along with the known outputs, the error can be calculated by subtracting the correct output from that of the untrained network's output. This error can then be used to calculate **gradients,** which are incremental changes to the weights and biases that would slightly reduce the error.

These gradients are calculated through a process call **backpropagation**. The weight and bias gradients are calculated many, many times to modify the network parameters accordingly. Eventually, the error should decrease if the network is training correctly. This process is called **gradient descent**.

Contents

Quick Start Guide

Before running any demos, be sure to move the "…/Assets/Noedify/Resources" folder to "…/Assets/Resources"

Preparing Your Dataset

Datasets consist of inputs and outputs. The inputs must be formatted in

```
List<float[,,]> trainingInputs
```

Typically, these three dimensions are used as: [channels,x-dimension,y-dimension]. In the case where you are only using 1D inputs to your network, it can be formatted into a 3D float array using:

```
float[,,] Noedify_Utils.AddTwoSingularDims(float[] trainingInputs1D)
```

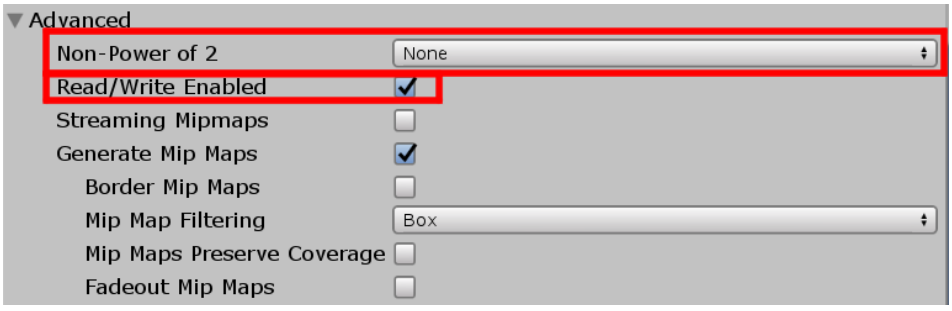
The output training set must be formatted as

```
List<float[]> trainingOutputs
```

When using images as your training data, it's recommended to first format them as Texture2D and then convert them into the appropriate format using:

```
public static void ImportImageData(
    ref List<float[,,]> trainingInputData,
    ref List<float[]> trainingOutputData,
    List<Texture2D[]> imageDataset,
    bool grayscale)
```

If you are using image files stored locally in your project, ensure that the import settings are set as the following:



Building Your Network

Deciding the optimal arrangement and size of hidden layers for an application is a complex topic. In the simplest case, we can add an input layer for 1D input data, a fully-connected hidden layer, and an output layer:

```
Noedify.Net net = new Noedify.Net(); // Instantiate our network


/* Input layer */
Noedify.Layer inputLayer = new Noedify.Layer(
    Noedify.LayerType.Input, // layer type
    10, // number of input nodes
    "input layer" // layer name
    );
net.AddLayer(inputLayer);

/* Hidden layer 1 */
Noedify.Layer hiddenLayer0 = new Noedify.Layer(
    Noedify.LayerType.FullyConnected, // layer type
    25, // layer size
    Noedify.ActivationFunction.Sigmoid, // activation function
    "fully connected 1" // layer name
    );
net.AddLayer(hiddenLayer0);

/* Output layer */
Noedify.Layer outputLayer = new Noedify.Layer(
    Noedify.LayerType.Output, // layer type
    5, // number of output nodes
    Noedify.ActivationFunction.Sigmoid, // activation function
    "output layer" // layer name
    );
net.AddLayer(outputLayer);

net.BuildNetwork(); // Compile the network
```

For a full list of possible layer types and how format them, see the definitions of **Noedify.Layer**.

Training Your Network

Now that we have a network defined and compiled, we can go ahead and train it with our formatted training data:

```
Noedify_Solver solver = Noedify.CreateSolver(); // Instantiate a solver object

solver.TrainNetwork(
       net, // reference to the compiled network
       training_inputs,
       training_outputs,
       trainingEpochs, // Number of epochs (iterations) to train for
       trainingBatchSize, // Size of the training set ensemble
       trainingRate, // Speed at which the solver will apply gradients
       Noedify_Solver.CostFunction.MeanSquare, // Training cost function
       Noedify_Solver.SolverMethod.MainThread, // solver computation method
       decision_weights, // (optional) weights given to training sets. 1 weight per set
       N_threads); // Number of threads to use when using background training
```

The **trainingEpochs** determines the duration of the training session in terms of the number of iterations.

trainingBatchSize determines the number of simultaneous training sets used during each epoch.

trainingRate determines the magnitude by which the network parameters will change after each epoch. Setting this too low will result in slow convergence, while having set too high will give poor accuracy and, in some cases, result in instability. A value of 0.2-0.8 works well for most applications.

The **costFunction** specifies the error calculation method. In most cases, `Noedify_Solver.CostFunction.CrossEntropy` will provide faster convergence but can result in instability.

The **SolverMethod** determines whether training will occur in the main Unity thread (freezing gameplay until complete) or in the background (using: `Noedify_Solver.SolverMethod.Background`) where gameplay is uninterrupted. When using background training, special care should be taken to ensure that the solver has completed training before trying to evaluate the network. See the demos for examples on how to do this using Coroutines.

N_threads applies only when using Background training. This determines the number of simultaneous batches computed on different CPU threads.

When training is complete, assuming a suitable low cost is found, we can then move on to deploying and evaluating the network. If the cost is not sufficiently small, it's recommended to train for additional epochs or modify the network layer structure.

Evaluating Your Network

Now that the network is trained, we can test it with evaluation inputs:

```
solver.Evaluate(
       net, // reference to compiled (and trained) network
       testInput, // evaluation network inputs
       Noedify_Solver.SolverMethod.MainThread); // solver computation method
```

Similar to the TrainNetwork() function, Evaluate() can either be run in the main thread or in the background. When running in the main thread, the prediction output can then be accessed from:

```
float[] predictions = solver.prediction;
```

If using background training, the predicted outputs can be accessed in the same way but care should be taken to only attempt to retrieve the outputs after `solver.evaluationInProgress` is false.

Deploying Your Network

Rather than always training your network from scratch during runtime, the trained network can be saved using:

```
net.SaveModel("trainedModel");
```

and later loaded using:

```
net.LoadModel("trainedModel");
```

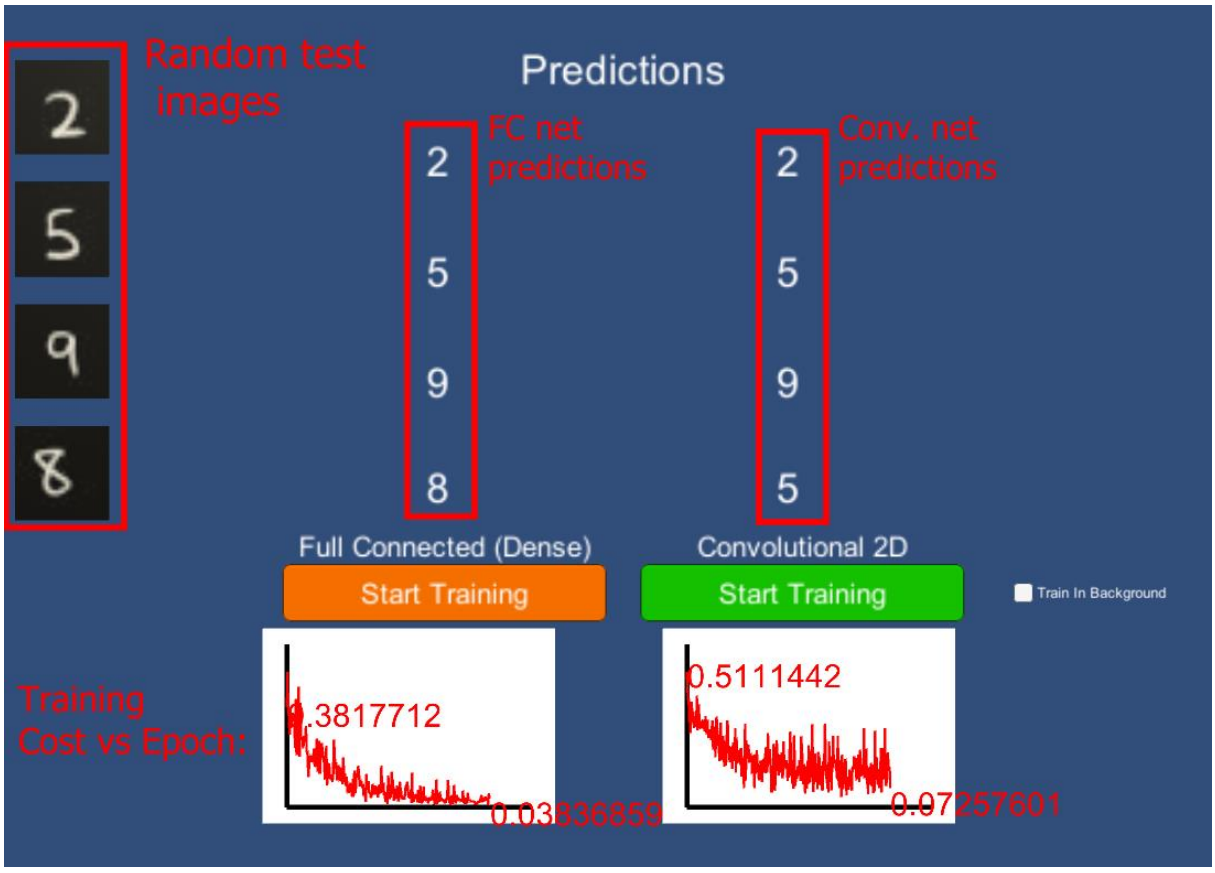
Loaded models can be evaluated as if they were trained from scratch or retrained to fine-tune the performance.

Demos

Before running any demos, be sure to move the "…/Assets/Noedify/Resources" folder to "…/Assets/Resources"

Demo 1: Classifying Handwritten Digits

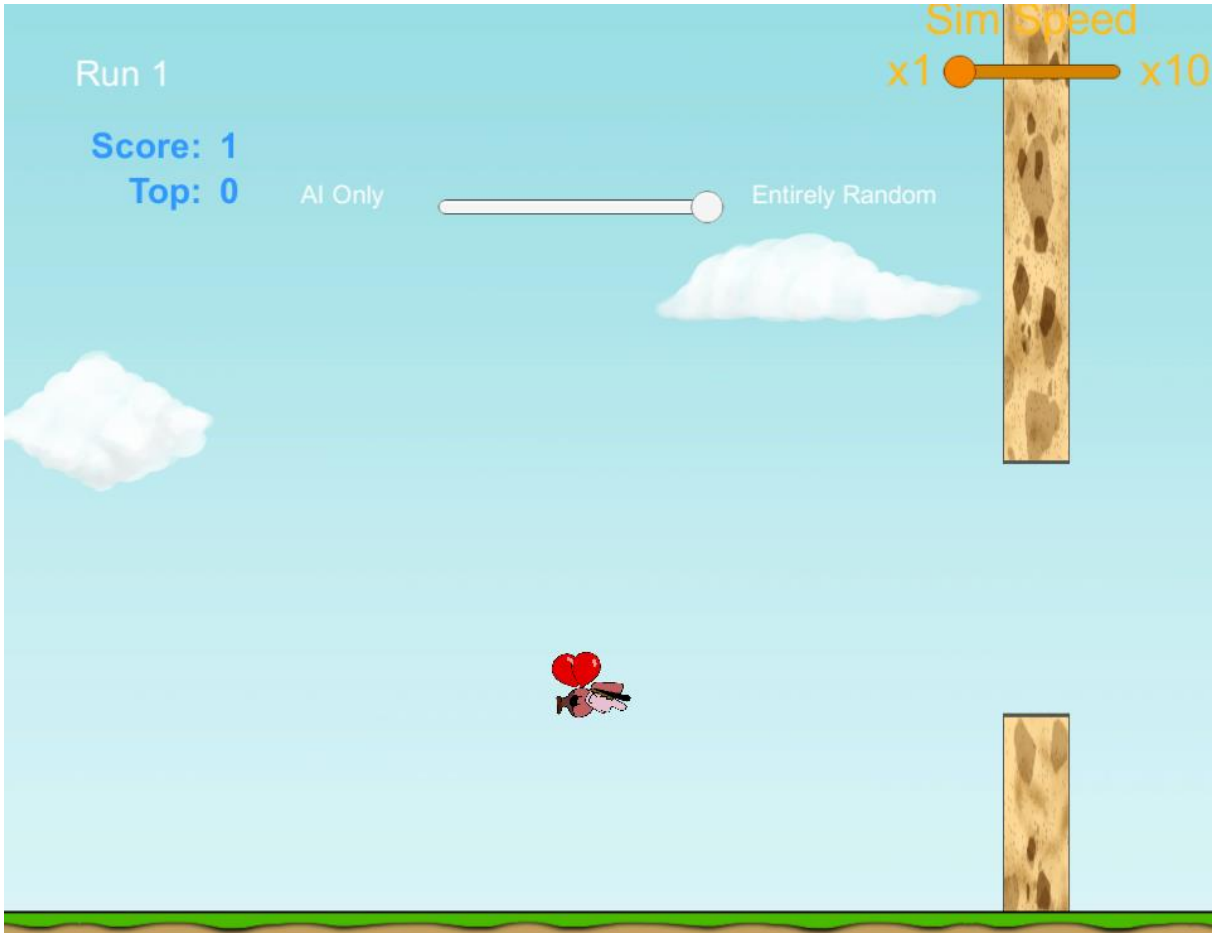
This demo demonstrates the training of a network for recognizing handwritten digits with a variety of techniques: fully-connected vs convolutional 2D and MainThread training vs. Background training.



See the scripts attached to the *"NN_Object"* component to see the list of training images used, as well as the individual scripts for the fully-connected and convolutional networks. The "*Train in background"* toggle button can be used to switch the solver mode between MainThread and Background. See how modifying the network structure can improve convergence or speed up the training process.

Demo 2: Hoppy AI

This demo will demonstrate how to train a simple 2D AI to avoid obstacles using a fully-connected network.



There are many ways to train an AI to perform tasks. In this case, we will use a simple approach to generate useful training data:

1. Collect a set of observations based on each Hoppy's current location (*y-pos, next obstacle distance, current fall speed, ect.)*
2. Generate a semi-random behaviour (*hop upwards, or continue falling*)
3. After some delay, check if Hoppy is still alive. If so, add this Hoppy's observation/decision to a list of successful training sets.
4. After some minimum number of training sets is collected, train the network.
5. As more training data is collected, gradually switch from random decisions to AI-based decisions from the trained model.

Rather than using purely-random decisions, we can speed up the generation of a useful decision dataset by applyng some simple rules to Hoppy's behavious. For example:

- If Hoppy is too close to the ground, make it more likely that he will hop upwards.
- If Hoppy's y-position is above the next obstacles y-position, make it less likely for him to hop.

To successfully train Hoppy to navigate the obstacles, it's recommended to go through these steps:

1. Use the arrow buttons to select the number of simultaneous Hoppy's. The larger the number, the faster the training set will be generated. It's recommended that 50-100 Hoppy's are used.
2. Press the "*Start"* button
3. You can watch the runs play out in real time as the training sets are generated, or you can use the time slider to speed up the process. *Note: using a very high time factor can cause physics glitches which may impact training*
4. By default, every 10 runs the network will be trained. The decision slider will initially be "Entirely Random", meaning the evaluated outputs of the network are not used in decision making. After 10-20 runs, start moving the slider towards the "AI Only" side. This will improve the training set generation rate.
5. To test out the model, move the decision slider all the way to "AI Only". This will make the trained network outputs entirely responsible for decision making, and only a single Hoppy will spawn.

Demo 3: Live Digit Prediction

The demo will demonstrate importing a Tensorflow-trained model into Noedify Unity and evaluating it during runtime to predict handwritten digits.

In order to run this demo properly, the "…/Assets/Noedify/Resources" folder must be moved to "…/Assets/Resources"

Upon running the demo for the first time, the network parameters will be loaded from the `Noedify-Model_Digit_Drawing_Test.txt` file, and then saved to a binary file. Loading from parameter files takes longer, so future runs of this demo will load from the binary file. Make sure to delete this binary file if you want to load an updated parameters file.

Once the model is loaded, you can use the mouse to draw shapes within the red box. The network will be evaluated and provide a live prediction of the digit you are drawing. Press 'C' to erase the drawing.

Code is also provided to train the MNIST model in Tensorflow. Both Python and Jupyter Notebook files are provided for this. Before running this, ensure you have installed Python (Python 3.7 is recommended) as well as the following python modules:

- Tensorflow (CPU or GPU)
- Keras
- Numpy

See https://www.tensorflow.org/install for instructions on installing Tensorflow.

To train the network with Python, open a command line window in the following folder:

"…/Assets/Noedify/Demos\TensorflowImport_MNIST_Drawing\Tensorflow\Python\"

Then run the command: "python TrainMNIST.py" and you should observe the following output:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 6s 102us/sample - loss: 0.6271 - accuracy: 0.8209 - val_loss: 0.2653 - val_accuracy: 0.9229
Epoch 2/10
60000/60000 [==============================] - 6s 101us/sample - loss: 0.1932 - accuracy: 0.9424 - val_loss: 0.1569 - val_accuracy: 0.9548
Epoch 3/10
60000/60000 [==============================] - 4s 71us/sample - loss: 0.1386 - accuracy: 0.9592 - val_loss: 0.1262 - val_accuracy: 0.9609
Epoch 4/10
60000/60000 [==============================] - 7s 110us/sample - loss: 0.1035 - accuracy: 0.9690 - val_loss: 0.1049 - val_accuracy: 0.9685
Epoch 5/10
60000/60000 [==============================] - 9s 148us/sample - loss: 0.0829 - accuracy: 0.9750 - val_loss: 0.0991 - val_accuracy: 0.9691
Epoch 6/10
60000/60000 [==============================] - 6s 104us/sample - loss: 0.0659 - accuracy: 0.9804 - val_loss: 0.0827 - val_accuracy: 0.9744
Epoch 7/10
60000/60000 [==============================] - 8s 130us/sample - loss: 0.0543 - accuracy: 0.9831 - val_loss: 0.0872 - val_accuracy: 0.9725
Epoch 8/10
60000/60000 [==============================] - 8s 138us/sample - loss: 0.0434 - accuracy: 0.9870 - val_loss: 0.0759 - val_accuracy: 0.9776
Epoch 9/10
60000/60000 [==============================] - 8s 129us/sample - loss: 0.0344 - accuracy: 0.9897 - val_loss: 0.0730 - val_accuracy: 0.9774
Epoch 10/10
60000/60000 [==============================] - 8s 129us/sample - loss: 0.0279 - accuracy: 0.9921 - val_loss: 0.0690 - val_accuracy: 0.9801
```

An output file named "FC_mnist_600x300x140_parameters.txt" should be created after running this script which contains the new network parameters. This file should be reference within Noedify Unity when importing network parameters, so it's recommended that it is moved to the "Resources" folder.

To run the training script with Jupyter Notebook and assuming you have Jupyter Notebook installed, open the notebook in:

"…/Assets/Noedify/Demos\TensorflowImport_MNIST_Drawing\Tensorflow\Jupyter\"

like any other Jupyter Notbook, and run the "TrainMNIST" script.

Importing Models

Importing from Unity Binary File

The fastest way to save/load trained models is saving/loading them as binary files. For example, to save the trained network 'net' in the default `Application.persistentDataPath`:

```
net.SaveModel("savedNet");
```

and later load using:

```
net.LoadModel("savedNet");
```

Other paths for saving/loading models can also be specified:

```
net.SaveModel("savedNet", <other path>);
```

This approach saves both the model's parameters and structure, meaning that a model doesn't need to be compiled before being loaded.

Importing from Tensorflow Keras

Currently, only loading Keras models consisting only of fully-connected (dense) layers is support.

Tensorflow can be used to quickly train large models. With the Keras high-level API, these models can be imported into Noedify Unity and evaluated or further trained.

From the python side, after training a Keras model 'model':

```
import NoedifyPython

NoedifyPython.ExportModel(model, "trained_model_parameters.txt")
```

This will generate a parameters files which can be imported into Noedify Unity using:

```
bool status = net.LoadModel("trained_model_parameters.txt");
```

Where 'status' will return true if the model import was successful.

An important note: unlike when importing a binary model file, in this case, our network 'net' must be defined and compiled before importing the parameters, since the model structure is not stored in the parameters file. If the Tensorflow model structure does not match this compiled Noedify model, LoadModel() will return an error.

Loading the model parameters is significantly slower than importing a binary model file, and so it is recommended that after import the parameters the first time, a binary file should be saved for future imports.

Classes and Functions

Noedify.Net

```
public class Noedify.Net
```

Neural network object containing all network weights and biases and connection mappings.

Key Public Properties:

```
public List<Noedify.Layer> Noedify.Net.layers;
```

Public Methods:

```
public Net()
```

Constructor for creating new network object.

```
public Noedify.Layer Noedify.Net.AddLayer(Noedify.Layer new_layer)
```
Add layer to existing network object. Layers should be added front to back, starting with the input layer.

```
public void Noedify.Net.BuildNetwork()
```

Construct network and initialize network parameters. Run this after adding layers.

```
public void Nodeify.Net.SaveModel(string name, string dir = "")
```

Save current model file as a binary file. If directory name "dir" is left empty, `Application.persistentDataPath` *will be used as the default.*

```
public bool Noedify.Net.LoadModel(string name, string dir = "")
```

Load Model binary file into Noedify.Net object. If directory name "dir" is left empty, `Application.persistentDataPath` *will be used as the default. Return "false" if the load operation was unsuccessful.*

Noedify.Layer

```
public class Noedify.Layer
```

Neural network layer definition.

Key Public Properties:

```
public Noedify.LayerType layer_type;
public string name;
public int layerSize;
public int[] layerSize2D;
public int channels;
public int layer_no;
public bool trainingActive;
public Noedify.NN_Weights weights;
public Noedify.NN_Biases biases;
public Noedify.ActivationFunction activationFunction;
```

Public Methods:

```
public Layer(
       LayerType newType,
       int newLayerSize,
       string newLayerName = "") // (a) Input 1D

public Layer(
       LayerType newType,
       int[] inputSize2D,
       int noChannels,
       string newLayerName = "") // (b) Input 2D

public Layer(
       LayerType newType,
       int newLayerSize,
       ActivationFunction actFunction = ActivationFunction.Sigmoid,
       string newLayerName = "") // (c) Fully-Connected(Dense)/Output
```

```
public Layer(
       LayerType newType,
       Layer previousLayer,
       int[] filtsize,
       int[] strd,
       int nfilters,
       int[] pdding,
       ActivationFunction actFunction = ActivationFunction.Sigmoid,
       string newLayerName = "") // (d) Convolutional 2D

public Layer(
       LayerType newType,
       Layer previousLayer,
       int[] shape,
       int[] strd,
       int[] pdding,
       PoolingType pooling_type,
       string newLayerName = "") // (e) Pooling 2D
```

Constructor for creating a new layer object. There are different input configurations for:

(a) *1D Input Layer*
(b) *2D Input Layer*
(c) *Fully-Connected (Dense) hidden layer or output layer*
(d) *Convolutional 2D hidden layer*
(e) *2D Pooling layer*

Available activation functions:

- *Sigmoid*
- *ReLU (rectified linear unit)*
- *Linear*
- *SoftMax*
- *ELU (exponential linear unit with α = 1)*
- *Hard_sigmoid*
- *Tanh*
- *LeakyReLU (leaky rectified linear unit with α = 0.05)*

Noedify_Solver

```
public class Noedify_Solver
```

Noedify neural network solver monobehavior object used for training and evaluating networks. Create this object using:

```
static public Noedify_Solver Noedify.CreateSolver()
```

and destroy using:

```
static public void Noedify.DestroySolver(Noedify_Solver solver)
```

Key Public Properties:

```
public bool trainingInProgress;
public bool evaluationInProgress;
public bool suppressMessages = false;
```
Enabling "suppressMessages" will prevent warning and error messages from appearing in the console as well as suppress the training cost value messages.

```
public float[] cost_report;
```
Following training, the cost at the end of each epoch is stored in the "cost_report" array.

```
public float costThreshold;
```
Cost threshold which, when reached, will cause training to cease. Value is set to -100 be default.

```
public float[] prediction;
```
After running an evaluation operation, the network output is stored in the "prediction" array.

```
public JobHandle activeJob;
public int fineTuningLayerLimit;
```

"fineTuningLayerLimit" specifies the highest layer number to train. For example, setting this to 0 will train all layers in the network. Setting this to 2 for a network with only 1 hidden layer will train only the output layer and not the hidden layer. This is useful for fine-tuning pretrained models.

Public Methods:

```
public void Evaluate(
	Noedify.Net net,
	float[,,] evaluationInputs,
	SolverMethod solverMethod = SolverMethod.MainThread)
```

Run a single forward evaluation of the network "net" with network inputs "evaluationInputs" and store the network outputs in "Noedify_Solver.prediction". Use "solverMethod" to select "SolverMethod.MainThread" for evaluating in the main thread or "SolverMethod.Background" to run in a background thread using Unity's Jobs System.

```
public void TrainNetwork(
	Noedify.Net net,
	List<float[,,]> trainingInputs,
	List<float[]> trainingOutputs,
	int no_epochs,
	int batch_size,
	float trainingRate,
	CostFunction costFunction,
	SolverMethod solverMethod,
	List<float> trainingSetWeighting = null,
	int N_threads = 8)
```

Train network net with training set "trainingInputs" and labels "trainingOutputs". When using "SolverMethod.Background", specifiy the number of CPU threads to use with "N_threads".

Noedify_Utils

```
public class Noedify_Utils
```

Collection of utilities for preparing and manipulating training data.

Public Methods

```
public static int ConvertOneHotToInt(float[] oneHot)
```
Converts from one-hot (categorical) representation to Int representation. Selects maximum value of input float array.

```
public static float[] FlattenDataset(float[,,] matrix)
```

```
public static List<float[]> FlattenDataset(List<float[,,]> matrix)
```

Flattens a 3D float array with size (h, w, d) (or 3D float array list) into 1D with size (hwd).

```
public static int[] Shuffle(int[] inputArray)
```
Returns a randomly shuffled array of the input array.

```
public static float[,] RotateImage90(float[,] imageData)
```
Rotates image array "imageData" 90 degrees clockwise.

```
public static float[,] FlipImage(float[,] imageData, bool horizontal = true)
```
Flips image array "imageData" either horizontally or vertically

```
public static float[,,] AddSingularDim(float [,] array)
public static float[,] AddSingularDim(float[] array)
public static float[,,] AddTwoSingularDims(float[] array)
```
Add a singular dimension (size 1) to a 1D or 2D array, making it 2D or 3D, respectfully, or add two singular dimensions to a 1D array making it 3D.

```
public static float[,] SqueezeDim(
       float[,,] array,
       int squeezeIndex,
       int dimensionIndex = 0)
public static float[] SqueezeDim(
       float[,] array,
       int squeezeIndex,
       int dimensionIndex = 0)
```
Removes dimension "dimensionIndex" from a 2D or 3D array. For example, a 3D array G with size (h, w, d): SqueezeDim(G, squeezeIndex, dimensionIndex = 1) would result in the 2D array G[:,squeezeIndex,:]

```
public static void ImportImageData(
       ref List<float[,,]> trainingInputData,
       ref List<float[]> trainingOutputData,
       List<Texture2D[]> imageDataset,
       bool grayscale)
public static void ImportImageData(
       ref float[,,] predictionInputData,
       Texture2D imageDataset,
       bool grayscale)
```

Load a single Texture2D image or a List of Texture2D[] images into a 3D input float or list of 3D floats (r,g,b). When loading lists of images, training labels are also exported based on the List index of the List<Texture2D[]> input.

Support

If you are having issues with Noedify or have a suggestion, please go to https://www.TinyAngleLabs.com/contact-us and get in touch.