Musically Embodied Machine Learning

The project is an investigation into the musically expressive potential of machine learning when embodied within physical musical instruments. It proposes 'tuneable ML', a novel approach to exploring the musicality of ML models, when they can be adjusted, personalised and remade, using the instrument as the interface.

The project asks how instruments can be designed to make effective and musical use of embedded ML processes, and questions the implications for instrument designers and musicians when tunable processes are a fundamental driver of an instrument's musical feel and musical behaviour.

MEML will run from April 2024 to Sept 2025

UKRI Arts and Humanities Research Council

EmuteLab

c.kiefer@sussex.ac.uk

Interactive Demo

Play a kick drum pattern on the drum machine. A machine learning model, running on the **uSEQ** module in the modular synthesiser, will accompany your pattern with other drums on the drum machine. Try some varied patterns to hear different accompaniments.

The model has been trained on a small dataset of drum patterns. Based on your kick drum pattern in the previous two bars, it predicts whether or not the other drums should be playing in the next time step.

Technology

The drum prediction model needs to run in **realtime** within the **constrained hardware** environment of the uSEQ synthesiser module: a low-power RP2040 microcontroller, overclocked at 250MHz, with 220Kb RAM, and no floating point unit (a £1 computer!)

The model uses **Differentiable Logic** to create a system that runs extremely quickly, using only bitwise logic operations.

The model is trained in PyTorch, then exported to C code to run on the microcontroller.

The model has 9 layers or 240 nodes. It consumes < 2Kb RAM, and interference takes around 1.4ms.

Petersen, F., Borgelt, C., Kuehne, H. and Deussen, O., 2022. Deep differentiable logic gate networks. Advances in Neural Information Processing Systems, 35, pp.2006-2018.

The model looks like this:

```
void logic_gate_net(char const *inp, char *out) {
      const char v0 = (char) (inp[20]);
      const char v1 = (char) (inp[17] ^ inp[6]);
      const char v2 = (char) (inp[15]);
      const char v3 = (char) (inp[17]);
      const char v4 = (char) (inp[21]);
      const char v5 = (char) (inp[2] & ~inp[2]);
      const char v6 = (char) (~inp[3]);
      const char v7 = (char) (inp[9]);
      const char v8 = (char) (inp[31]);
      const char v9 = (char) (inp[22] & ~inp[19]);
      const char v10 = (char) (inp[31]);
      const char v11 = (char) (~(char) 0);
      const char v12 = (char) (inp[31] & inp[31]);
      const char v13 = (char) (~(inp[5] ^ inp[26]));
      const char v14 = (char) (inp[15]);
      const char v15 = (char) (inp[13]);
      const char v16 = (char) (inp[6] ^ inp[30]);
      const char v17 = (char) (inp[24] ^ inp[14]);
      const char v18 = (char) (~inp[31]);
      const char v19 = (char) (~inp[16] | inp[29]);
      const char v20 = (char) (inp[7] & inp[16]);
```