

AIMC 2023

[neuralnet]: A Pure Data External for the Creation of Neural Networks Written in Pure C

Alexandros Drymonitis¹

¹PhD, Independent scholar

URL: <https://aimc2023.pubpub.org/pub/3j3fx7y1>

License: [Creative Commons Attribution 4.0 International License \(CC-BY 4.0\)](https://creativecommons.org/licenses/by/4.0/)

ABSTRACT

[neuralnet] is a Pure Data external object that allows the creation of densely connected Neural Networks of any structure. This object is written entirely in C, without any dependencies. The code is a translation from Python code to C. The Python code is taken from the book *Neural Networks from Scratch in Python*. The aim is to create a Pd object that can be easily compiled for all OSes, so it can be easily used by Pd users of any level of expertise. This is a lightweight object that is easy to set up, train, validate, and use for predictions. It covers three modes of operation, including classification, regression, and binary logistic regression, and it has many settable parameters, so users can customise it to their own needs. This is an open-source project.

INTRODUCTION

Neural Networks (NN) have made their way into art, and more specifically music, with a multitude of software, dating back to the early 1990s [1]. Languages like MATLAB have been used in a musical context to provide an interface to NNs [2][3][4]. Software like the Wekinator [5] has made it possible for musicians to easily integrate NNs into their practice [6][7]. With the advent of greater processing power in personal computers and the inclusion of Graphical Processing Units (GPU) in AI, libraries like TensorFlow [8], or Keras [9], which builds on top of TensorFlow, providing a more user-friendly interface, have been utilised in numerous projects [10][11][12].

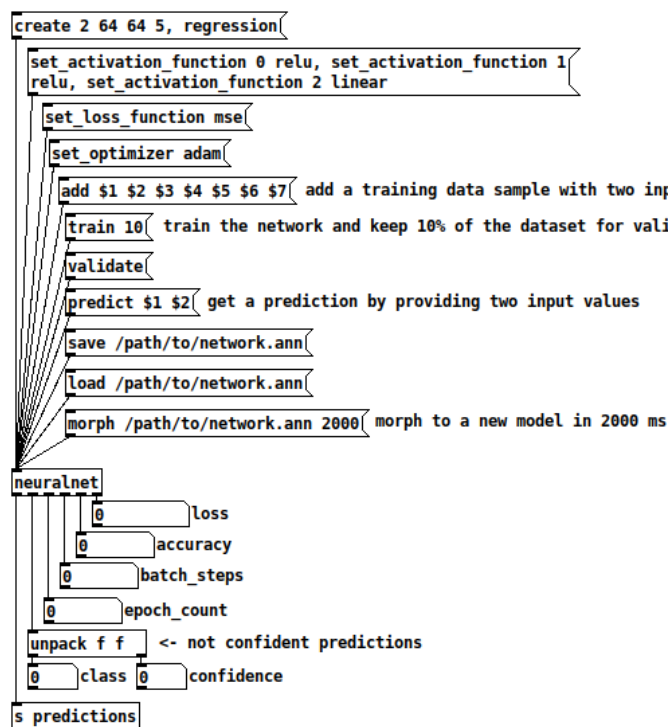
Recent research in the field of ML in music has incorporated more advanced NN structures for various musical tasks [13][14][15][16][12]. These include Recurrent Neural Networks (RNN) with Long Short-Term Memory (LSTM) [17] for generating sequences, or the GANSynth [18] and the DDSP [19] libraries, for generating audio. Even though the focus of ML in music has widened, the most fundamental type of NNs, the Multilayer Perceptron (MLP) [20] can have many creative applications in music, as demonstrated by [2][3][5], and the examples realised with [neuralnet], presented in its documentation and further on in this paper.

Most of the approaches in applying ML and NNs to musical projects have two things in common: multiple software and dependencies. [neuralnet] is a framework for MLPs of various structures, written entirely in C, without any dependencies, as a Pure Data (Pd) external object. The starting point for writing this external was the book *Neural Networks from Scratch in Python* [21]. The C code of [neuralnet] is a translation from the Python code of this book, adapted to the Pd API. The translation from Python to C was entirely done manually.

As the name of [21] implies, the entire code is written from scratch. The same applies to the C code of [neuralnet]. This Pd object does not use any external frameworks to build the NNs, nor does it apply activation or loss functions, or optimisers from other libraries. All parts of the MLP are independent of other software apart from the Pd API, for which it is aimed. The goal of this object is to provide a lightweight MLP, and an

easy way to import or even train NNs inside Pd without needing any external resources other than the [neuralnet] object itself.

As with all Pd external objects that have no dependencies, compiling for any major Operating System (OS) is very easy. [neuralnet] has been compiled for and has been tested on Linux, macOS, and Windows, with the help of the Pd community. This is an open-source project, released under the GNU GPL license. Its source code is hosted on [GitHub](#).



The [neuralnet] object with various messages for network creation including structure and mode, activation and loss functions, optimisers, training data samples, and others.

RELATED WORK

A few Pd external objects and libraries for creating, training, and using NNs exist. The recent [fann] object [22], which is an updated version of the [ann_mlp] object of the Ann library [23], is an MLP based on the FANN library (Fast Artificial Neural Network) [24]. FANN is written in C, so, in this respect, the [fann] object bares great resemblance to [neuralnet]. [fann] though depends on FANN, and the initial Ann library was built against an older version of FANN, which was the reason for the developer of [fann] to update the object. Thus, [fann], even though it is a Pd object written in C, depends on another library and it has to keep up with its updates. Depending on the FANN library though, does have advantages, as it includes numerous activation and

loss functions, and training algorithms. Currently though, [fann] is built for classification only, whereas [neuralnet] has more modes, which are described in a later section in this paper.

Another Pd library for ML is ml.lib [25]. This library contains many objects for various ML tasks. An object of this library that relates to [neuralnet] is [ann], an MLP for classification and regression. The [ann] object has various settable parameters, including the number of neurons per layer, but its structure is constrained to one hidden layer only. This library is written in C++, which means that the process of compiling its objects is a bit more involved than compiling an object written in C, as Pd's default language is C, and it does not officially support objects written in other languages.

FluCoMa [26] is a new library for ML that targets Pd, Max, and SuperCollider. This is a project of five years of research that ended up in a rich library with a unified interface across the different programming environments it supports. As with ml.lib, FluCoMa has a much wider context than [neuralnet], which is a single object. Supporting three different programming environments adds to the difference of magnitude between FluCoMa and [neuralnet]. Still, this library includes NNs, as well as Deep Neural Networks (DNNs), for classification and regression, which relates to [neuralnet]. Its MLP objects provide flexibility by enabling various settable parameters, similar to [neuralnet]. Compared to the latter, it forces the same activation function across all layers, except the output layer, providing a structure that is somewhat less flexible. This library has a much more open design though, with separate objects for datasets and MLPs, which can provide flexibility on a different level than the MLP structure, as the user has access to data that is exchanged between objects during the process of training. This library is also written in C++.

Apart from external objects and libraries, there are two projects that build NNs as Pd patches or abstractions [27][3]. [27] is built entirely in Pd, whereas [3] loads NN models created and trained in MATLAB. Once a model is trained though, it is loaded onto a Pd patch and no additional software is needed. This project uses the iemmatrix external library for Pd to apply matrix operations. For educational purposes, having access to the internal structure of an NN can be beneficial, as users can monitor the training or prediction process of an MLP. A single, black-box-type external object though is likely to be faster in its computations, as it packs all of its processing, waiving the Pd interface.

From all the projects presented in this section, the NNs and DNNs that can be created with FluCoMa are the most relevant to [neuralnet], because DNNs are also possible, as opposed to the [ann] object, and the networks are created in compiled code, whether C or C++, and not as a Pd patch. At the time of writing, FluCoMa is the most recent library, being actively maintained, and having an active user community [28]. Since [neuralnet] was completed around the same time the FluCoMa research project was concluded, I consider the two to be contemporary. For all these reasons, I choose FluCoMa as the best point of comparison throughout this paper.

THE NEURALNET OBJECT

This section focuses on the design process of [neuralnet], the approach to how the object was developed and highlights some of its features. A reflection on the black box versus the open-design model is also made, with a comparison between [neuralnet] and the FluCoMa library, focusing on how each is used inside a Pd patch.

DESIGN PROCESS

The starting point for writing the code of this object was the C implementation provided by the authors of [21] on GitHub [29]. The provided code though is minimal and covers only the very first steps of the Python code that implements an entire NN, with different activation and loss functions, optimisers, and other features. From that point on, the Python code of the full version was used as a template, translated to C.

Reverse Engineering

The original Python code from [21] makes heavy use of the NumPy module [30]. This is a very efficient module for scientific computing in Python, used in many projects. By abstracting many of its functionalities, it provides an intuitive interface that can quickly get users up and running. When translating code from Python that uses NumPy to C though, a certain amount of reverse engineering is necessary. This included getting a normal distribution in random values, getting the dot product of matrices, transforming matrices to get the dot product right, creating Jacobian matrices, or applying other calculations to matrices.

With NumPy, most of the calculations applied to matrices are done by simply passing the entire matrix as an argument to the corresponding function. In C, each matrix, created as a 2D array, has to be unpacked and the desired calculation has to be applied to each element separately. Even this simple translation from one language to the other, needed careful design to get the correct results.

Memory Management

Managing memory in [neuralnet] was the hardest part of the design process. The latter included a lot of testing and refining, with many crashes happening. Debugging the code with debuggers like GDB or Valgrind was constantly employed. Before attempting to adapt the code to the Pd API, the way memory was managed had to be decided. The issues that arose were made apparent only after [neuralnet] was written as a Pd object and loaded in Pd. The debugging process and the finalisation of the memory management were both realised once a Pd object binary existed.

In Python, memory management is very different than in C. In the case of the former, creating a matrix of weights for a layer is simply done by calling a NumPy random function with normal distribution, within the `__init__()` method of a class. Once this is done, the object of this class will keep the matrix of random weights in its memory. In C, this is achieved by creating pointers that can, later on, be allocated to memory, based on the structure of the MLP. In case this structure changes, the allocated memory has to be freed and new

memory has to be allocated. In Python, this is simply done by calling `del(objectname)` and `gc.collect()` from the GC module, which deals with garbage collection.

Memory management concerning training datasets is also done very differently from the Python code. [neuralnet] can either load training data from Pd arrays, or this data can be sent straight to [neuralnet] where they will be stored internally. In any case, memory has to either be allocated when the amount of data is known or reallocated when new data comes in. Reallocation is done when training data are sent to the object without the use of Pd arrays.

Adapting to the Pd API

After the reverse engineering and the approach to memory management were dealt with, the design focused on adapting to the Pd API. This included how data either for training, validating, or predicting would be input to the object, how information during training and validating, and predictions would be output, and how the object would be structured, concerning its inlets and outlets.

The structure I ended up with was one inlet for all data and messages, and a separate outlet for each information or output data. The input messages to [neuralnet] are way too many for each to have its own inlet, therefore the approach of one inlet accepting different messages was utilised. The messages are distinguished by their selector, like “create”, “destroy”, “add”, and others.

The output data of [neuralnet] do not have so many different types as the input data. Six outlets were enough to cover all the desired functionalities. These include predictions, not so confident predictions, epoch count, batch steps, accuracy, and loss values. Instead of creating six outlets, the object could also have one outlet and use the [route] object to route the different output data, as many other Pd objects do. Still, I opted for separate outlets, as I feel this is clearer design.

MONOLITHIC DESIGN WITH A MODULAR APPROACH

The [neuralnet] object is written as a monolithic C file, with its accompanying header file. Even though its source code is monolithic, within the main file, the code is modular, as it breaks the various processes for training, testing, and predicting, from the routines that call them. The same applies to the various activation and loss functions, and the available optimisers. This approach renders both maintenance and development of the object rather easy. On one hand, there are two files only that need to be maintained. On the other hand, it is easy to add new features or modify existing ones, like activation and loss functions, since these are broken out of other parts of the code. The code below is taken from the header file of the object and provides an example of how the activation functions are separated from the rest of the code.

```
static t_float sigmoid_forward(t_neuralnet *x, int input_size, int index, int out);
static t_float relu_forward(t_neuralnet *x, int input_size, int index, int out);
```

```
static t_float linear_forward(t_neuralnet *x, int input_size, int index, int out);
static t_float softmax_forward(t_neuralnet *x, int input_size, int index, int out);
```

Since all activation functions take the same arguments, it is possible to store them in an array and call the function set by the user based on an index. The line of code below shows how this is done.

```
out_size = x->x_act_funcs[act_index](x, input_size, index, out);
```

In the line above, `x->x_act_funcs` is an array of function pointers that is part of the main data structure of the object, and `act_index` is an integer set by the user that points to the desired function. If `act_index` is 0, the sigmoid function will be called, if it is 1, the ReLU function will be called, and so on.

BLACK BOX VS OPEN DESIGN

Due to the design approach of FluCoMa, which is quite different from [neuralnet]’s approach, and for the reasons stated in the *Related Work* section of this paper, I will reflect on decisions made during the design process of [neuralnet], and compare them to the respective objects in FluCoMa. This does not mean that the design process of [neuralnet] was influenced by the FluCoMa library. This subsection serves only to reflect on the design approach applied to [neuralnet], and FluCoMa has been chosen as a reference.

The FluCoMa library applies a more open design than [neuralnet], with separate objects for the datasets and for the MLP [31]. All data are stored in audio buffer objects, as accessible open-ended memory. [neuralnet] can either draw data from Pd arrays (which are used by FluCoMa too, referred to as an audio buffer object), or data can be saved inside the object’s memory. In the latter case, the user has no access to the training data, once it is stored. Besides the dataset object, FluCoMa also includes a labelset object that holds labels, when classification is employed. [neuralnet] does not distinguish between classification and regression datasets, and the data of these sets are provided the same way.

The MLP object of the FluCoMa library provides input and output access to any layer. [neuralnet] is a black box that does not provide such access. Once a network is created, only the network’s input is provided by the user, and only its output is given back to him/her, without the user being able to access the data in between layers. Overall, FluCoMa hides as less as possible from the user, providing access to many points in the process of training, validating, and predicting.

Being an entire library with many more objects than the MLP, the reasons behind the design approach are different than [neuralnet]. For example, the dataset object is common for any object of the library that needs access to a dataset. Perhaps, it is this approach of providing training data to their various models, that led the FluCoMa team to make even their MLP more open. [neuralnet], being a black box, does not provide any access to its internal workings. From an educational point of view, the open-design approach of FluCoMa is probably better than the black box, as the former can give more insight into how various ML processes work. From a creative coding perspective, I consider the black box to provide a faster way for creating MLPs, and for

providing it with training and validating data. On the other hand, an open design like FluCoMa, can provide creative potential through techniques like network bending [\[32\]](#), or the visualisation of the training or prediction processes.

The [neuralnet] object is open-source, so its internal workings are not actually hidden. Not all Pd users though are comfortable with the C language. Having this in mind, when designing an object that aims at being transparent, transparency should be raised to the level of the Pd patch, rather than lowered to the level of the source code. [neuralnet] though does not aim to be an educational tool, but more of a functional tool that can provide an easy-to-build MLP framework.

NN MODES AND EXAMPLE PATCHES

[neuralnet] supports three modes of operation: classification, regression, and binary logistic regression. Classification is used to classify input among different classes provided in the training dataset. As a proof of concept, the fashion MNIST dataset [\[33\]](#) is used in the examples that come with the object, copied from [\[21\]](#).

In regression mode, the network can provide arbitrary values based on its training. The examples that come with the object installation provide two regression patches that control the parameters of a synthesizer. One example takes input from the mouse pointer coordinates and the other takes OSC messages as input, from the accelerometer of a smartphone. Both patches control the parameters of an FM synthesis patch, where the connections between input and output are non-linear, and the number of inputs and outputs is different within each network model.

In binary logistic regression mode, the network outputs 1s or 0s for each of its outputs. In this mode, a network can be used to output different states, like person/not person, and indoors/outdoors [\[21\]](#). The binary logistic regression mode example of [neuralnet] is the XOR example, inspired by the examples that come with the help patch of the [fann] object.

ACTIVATION AND LOSS FUNCTIONS AND OPTIMISERS

Currently, [neuralnet] provides four activation functions and four loss functions. The activation functions are Linear, Sigmoid, ReLU (Rectified Linear Units), and Softmax. The Sigmoid function is often used in regression and binary logistic regression modes, whereas the Softmax function is typically used in classification mode. The ReLU function is related to Sigmoid, though simpler but CPU cheaper. The Linear function is mostly used in the output layer when in regression mode.

The available loss functions are Mean Squared Error, Mean Absolute Error, Categorical Cross-Entropy, and Binary Cross-Entropy. The first two are mostly used in regression mode. Categorical Cross-Entropy is used in classification mode, and Binary Cross-Entropy is typically used in binary logistic regression, even though in the XOR example patch, the loss function is set to Mean Squared Error.

The available optimisers are also four, including Stochastic Gradient Descent (SGD), Adaptive Gradient (AdaGrad), Root Mean Square Propagation (RMSProp), and Adaptive Momentum (Adam). Even though all the examples of the object use the Adam optimiser, the other three were also included, as they were part of [\[21\]](#), which is the starting point of [neuralnet].

OTHER FEATURES

[neuralnet] is not merely a translation of the Python code found in [\[21\]](#), as it includes some features that are not part of the original code. Saving and loading models is a feature inspired by the original code, but an additional feature, that of morphing between models, has been implemented.

Saving and Loading Trained Models

Being able to save and load network models is important so that users do not have to train a network every time they want to use it. In the Python code [neuralnet] is based on, this is done by saving the model as a binary file, with Python's native pickle module. In [neuralnet], a model is saved as a text file, with the .ann suffix. This is a simple text file where the network structure, its mode, its activation functions, and its weights and biases are stored, separated by lines starting with the hash symbol, followed by text describing the type of the following values.

Loading a saved model is done by loading the corresponding text file. Once loaded, the file is parsed and all the necessary information is used to replicate a network, by creating the structure, setting the mode and activation and loss functions, and the values for the weights and biases. Once a model is loaded and parsed, the network can be used for predictions right away.

Morphing Between Network Settings

A feature that is not included in the original Python code is morphing between network settings. This is possible with networks with the same number of layers and the same number of neurons in each layer, as well as the same activation and loss functions. This means that there is no reallocation of memory when morphing, but only the weights and biases are changed. This morphing is done linearly, where the user can set the time it will take for the object to go from one setting to the other. This is like using the native [line] object to make a ramp from one value to another, and indeed, the implementation of the morphing feature is copied from the source code of [line]. The video below demonstrates this feature.



The morphing feature of the [neuralnet] object.

USE CASES

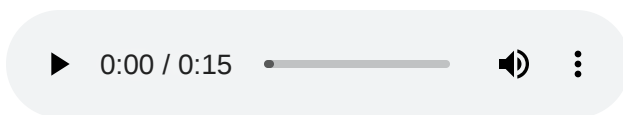
One use case of [neuralnet] is to control many parameters of a synth with only a few input values. The video below shows an MLP realised with [neuralnet], that controls twenty parameters of an FM synthesizer with the two mouse pointer coordinate values. The mouse pointer coordinates can be seen at the top part of the screen, and the twenty FM synth parameters that are controlled by the MLP can be seen at the bottom part of the screen. The FM synth is a Pd adaptation of one of the examples that come with the Pyo module for DSP in Python [\[34\]](#). This patch has been compared to a version realised in Python using Keras, where the Pd version with [neuralnet] consumes between 1.3% and 2.0% CPU, while the Python version consumes between 4.0% and 7.0% CPU. These tests have been run on the same machine, with the DSP turned off, so that the CPU usage is limited to the NN and the mouse pointer polling, and in Pd's case, to the GUI too.



An MLP regressor created with [neuralnet], controlling twenty parameters of an FM synth, with two input values, from the mouse pointer coordinates.

PLAYING NICELY WITH FLUCOMA

Being a Pd object, [neuralnet] can be used with any other Pd control-rate object. This includes objects from the FluCoMa ecosystem. One such example is the combination of [neuralnet] with [fluid.mfcc~], which extracts the Mel Frequency Cepstrum Coefficients (MFCC) [35]. The output of [fluid.mfcc~] can be fed to [neuralnet] to classify audio based on its timbre. The audio file below, demonstrates how vowels can be classified with [neuralnet], by feeding it with MFCC values from [fluid.mfcc~]. The audio output is frozen instances of text-to-speech synthesis of the pronounced vowels. Freezing is achieved with FFT, and text-to-speech synthesis is done with the [flite] Pd external object, based on the Flite library [36]. This example is inspired by the examples of FluCoMa.



Classifying pronounced vowels with MFCC values extracted from [fluid.mfcc~] and input to [neuralnet].

FUTURE WORK

Even though [neuralnet] is a complete MLP framework, there is room for further development. The file type of the saved NN models can be changed to the JSON format instead of the current bespoke format, or both the current and JSON could be an option. Including the JSON format could have the advantage of being able to

use models that have been created with [neuralnet] in other environments. This is possible with the RTNeural library [37] that is aimed at loading models created with TensorFlow or PyTorch [38].

Being able to break a saved model is another feature that could be added. This can be done by creating an NN with part of the structure of the saved model to be broken. An example is an autoencoder that has 1024 inputs, 512 neurons in the first hidden layer, 128 neurons in the second hidden layer, 13 in the latent space (these could be MFCC values), and the reverse order of the hidden layers and input, to complete the model. This model could be broken into two, with the first part consisting of the layers from the input up to the latent space, and the second from the latent space up to the output. If an object is created with `[neuralnet 1024 512 128 13]`, and another with `[neuralnet 13 128 512 1024]`, then the user could have access to the latent space. Even though including this feature is trivial, it has not been implemented yet.

The morphing feature should also be improved. Currently, with an MLP in regression mode that controls the parameters of a synth, when morphing from one model to another, the sound tends to shift both up and down in pitch during this process. This pitch shifting happens regardless of the weights and biases of the two models. A possible solution is to find the nearest value for each weight in a layer in the new model, compared to its current value. The way this feature is currently implemented, the weights are morphed the way they have been stored in the model file. This means that there is a possibility of many weights covering a wide range between their current and new values, whereas this could be significantly shorter.

A last addition that completes this section is a signal-rate version of this object. This version could be used with trained models only, much like the RTNeural library, or it could also include the training and validating parts of the network. Having a signal-rate object would eliminate the one-sample-block latency occurring when both input and output of the NN are signals. In the case of broken models, as described in the paragraph above, if any processing in the latent space is done with signals, then the overall latency with a control-rate object doubles. With a signal-rate object, there should be no additional latency, even with broken models.

CONCLUSIONS

This paper presents the [neuralnet] external object for Pd. This is an MLP framework written entirely in C, without any dependencies. This object is a translation of the Python code from the book *Neural Networks from Scratch in Python*, with some added functionalities. The advantage of not having dependencies is that it is easy to maintain, and it does not need to be up to date with any external development. Being written in C, it is easy to compile for Pd for all major OSes, as C is the default and supported language of the Pd API.

Compared to the FluCoMa library for ML, [neuralnet] is a black box that does not provide access to its internal workings. From an educational point of view, this can be seen as a disadvantage. Being a black box though, it can be considered to be simpler in the way an NN is created and fed with training data, and its overall use, as there is a single object that needs to be dealt with. Being a Pd object, [neuralnet] can be used in combination with the FluCoMa ecosystem, as it has been demonstrated in the *Use Cases* section of this paper. Being an

open-source project with no dependencies, [neuralnet] can be thought of as a contribution to education, different than that of FluCoMa, as the educational aspect is focused on the C code rather than the Pd patch. Even though not all Pd users are comfortable with C, there is an active community of Pd object developers, or other C coders, that could benefit from reading the source code.

Even though [neuralnet] is a complete and functional MLP framework, there is still room for improvement or further development. With certain additional functionalities, its use can be expanded to processing audio signals without converting from signal-rate to control-rate and back, or access to its internal structure could be possible. Being able to save models in the JSON format is another addition that could make it possible to use models created and trained with [neuralnet] in other frameworks, like the RTNeural library.

Overall, [neuralnet] is a lightweight MLP with many settable parameters and modes. Its complete documentation and its variety in the examples that come with the object aim at making the use of NNs an easy task. Bearing in mind that Pd has a large user and developer community, [neuralnet] adds to the vast array of tools that come with this widespread and powerful system.

References

- Arya, P., Kukreti, P., & Jha, N. (2022). *Music Generation Using LSTM and Its Comparison with Traditional Method*. <https://doi.org/10.3233/ATDE220793> ↵
- Bélanger, O. (2016). Pyo, the Python DSP Toolbox. *Proceedings of the 24th ACM International Conference on Multimedia*, 1214–1217. <https://doi.org/10.1145/2964284.2973804> ↵
- Black, A., & Lenzo, K. (2001). Flite: A Small Fast Run-Time Synthesis Engine. *Proceedings of the 4th ISCA Tutorial and Research Workshop on Speech Synthesis*. ↵
- Brent, W. (2018). *pd_fann*. https://github.com/wbrent/pd_fann ↵
- Bullock, J., & Momeni, A. (2015). ml.lib: Robust, Cross-platform, Open-source Machine Learning for Max and Pure Data. In E. Berdahl & J. Allison (Eds.), *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 265–270). Louisiana State University. <https://doi.org/10.5281/zenodo.1179038> ↵
- Chen, S., Zhong, Y., & Du, R. (2022). Automatic composition of Guzheng (Chinese Zither) music using long short-term memory network (LSTM) and reinforcement learning (RL). *Scientific Reports*, 12. <https://doi.org/10.1038/s41598-022-19786-1> ↵
- Chollet, F., & others. (2015). *Keras: deep learning library for theano and tensorflow*. 2015. ↵
- Chowdhury, J. (2021). RTNeural: Fast Neural Inferencing for Real-Time Systems. *arXiv Preprint arXiv:2106.03037*. ↵
- Cont, A., Coduys, T., & Henry, C. (2004). Real-time Gesture Mapping in Pd Environment using Neural Networks. *Proceedings of the International Conference on New Interfaces for Musical Expression*, 39–42. <https://doi.org/10.5281/zenodo.1176589> ↵
- Developers, T. (2021). *TensorFlow* (v2.5.0). Zenodo. <https://doi.org/10.5281/zenodo.4758419> ↵

- Engel, J., Agrawal, K. K., Chen, S., Gulrajani, I., Donahue, C., & Roberts, A. (2019). GANSynth: Adversarial Neural Audio Synthesis. *International Conference on Learning Representations*. <https://openreview.net/forum?id=H1xQVn09FX> ↵
- Engel, J., Hantrakul, L. (Hanoi), Gu, C., & Roberts, A. (2020). DDSP: Differentiable Digital Signal Processing. *International Conference on Learning Representations*. <https://openreview.net/forum?id=B1x1ma4tDr> ↵
- Fiebrink, R., Trueman, D., & Cook, P. R. (2009). A Meta-Instrument for Interactive, On-the-Fly Machine Learning. *Proceedings of the International Conference on New Interfaces for Musical Expression*, 280–285. <https://doi.org/10.5281/zenodo.1177513> ↵
- Gardner, M. W., & Dorling, S. R. (1998). Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric Environment*, 32(14), 2627–2636. [https://doi.org/https://doi.org/10.1016/S1352-2310\(97\)00447-0](https://doi.org/https://doi.org/10.1016/S1352-2310(97)00447-0) ↵
- Hantrakul, L. (2018). GestureRNN: A neural gesture system for the Roli Lightpad Block. In T. M. Luke Dahl Douglas Bowman (Ed.), *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 132–137). Virginia Tech. <https://doi.org/10.5281/zenodo.1302703> ↵
- Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., ... Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825), 357–362. <https://doi.org/10.1038/s41586-020-2649-2> ↵
- Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Comput.*, 9(8), 1735–1780. <https://doi.org/10.1162/neco.1997.9.8.1735> ↵
- Kerlleñevich, H., Eguía, M. C., & Riera, P. E. (2011). An Open Source Interface based on Biological Neural Networks for Interactive Music Performance. *Proceedings of the International Conference on New Interfaces for Musical Expression*, 331–336. <https://doi.org/10.5281/zenodo.1178063> ↵
- Kinsley, H., & Kukiela, D. (2020). *Neural Networks from Scratch in Python*. <https://nnfs.io> ↵
- Kinsley, H., & Kukiela, D. (2021). *NNfSiX*. <https://github.com/Sentdex/NNfSiX/tree/master/C> ↵
- Lee, M. (2018). Deep Neural Network based Music Source Conducting System. *Proceedings of the International Computer Music Conference (ICMC)*, 17–20. ↵
- Lee, M., Freed, A., & Wessel, D. (1991). Real-Time Neural Network Processing of Gestural and Acoustic Signals. *Proceedings of the International Computer Music Conference (ICMC)*. ↵
- Martin, C. P., & Torresen, J. (2019). An Interactive Musical Prediction System with Mixture Density Recurrent Neural Networks. In M. Queiroz & A. X. Sedó (Eds.), *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 260–265). UFRGS. <https://doi.org/10.5281/zenodo.3672952> ↵
- Murray-Browne, T., & Tigas, P. (2021, June). Latent Mappings: Generating Open-Ended Expressive Mappings Using Variational Autoencoders. *Proceedings of the International Conference on New Interfaces for Musical Expression*. <https://doi.org/10.21428/92fbeb44.9d4bcd4b> ↵

- Næss, T. R., & Martin, C. P. (2019). A Physical Intelligent Instrument using Recurrent Neural Networks. In M. Queiroz & A. X. Sedó (Eds.), *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 79–82). UFRGS. <https://doi.org/10.5281/zenodo.3672874> ↵
- Nissen, S. (2003). *Implementation of a Fast Artificial Neural Network library (FANN)*. http://iweb.dl.sourceforge.net/sourceforge/fann/fann_doc_complete_1.0.pdf ↵
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., & Chintala, S. (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. <https://arxiv.org/abs/1912.01703> ↵
- Snyder, J., & Ryan, D. (2014). The Birl: An Electronic Wind Instrument Based on an Artificial Neural Network Parameter Mapping Structure. *Proceedings of the International Conference on New Interfaces for Musical Expression*, 585–588. <https://doi.org/10.5281/zenodo.1178939> ↵
- Tahiroğlu, K., Kastemaa, M., & Koli, O. (2021). GANSpaceSynth. *Proceedings of the 2nd Joint Conference on AI Music Creativity*, 10. <https://doi.org/10.5281/zenodo.5137902> ↵
- Tremblay, P. A., Green, O., Roma, G., Bradbury, J., Moore, T., Hart, J., & Harker, A. (2022). *Flucoma Discourse*. <https://discourse.flucoma.org/> ↵
- Tremblay, P. A., Green, O., Roma, G., Bradbury, J., Moore, T., Hart, J., & Harker, A. (2022). *Fluid Corpus Manipulation Toolbox* (v.1). Zenodo. <https://doi.org/10.5281/zenodo.6834643> ↵
- Tremblay, P. A., Roma, G., & Green, O. (2021). Enabling Programmatic Data Mining as Musicking: The Fluid Corpus Manipulation Toolkit. *Computer Music Journal*, 45(2), 9–23. ↵
- Xiao, H., Rasul, K., & Vollgraf, R. (2017, August 28). *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. ↵
- Yaremchuk, V., Medeiros, C. B., & Wanderley, M. (2019). Small Dynamic Neural Networks for Gesture Classification with The Rulers (a Digital Musical Instrument). In M. Queiroz & A. X. Sedó (Eds.), *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 150–155). UFRGS. <https://doi.org/10.5281/zenodo.3672904> ↵
- Yee-King, M., & McCallum, L. (2021). Studio report: sound synthesis with DDSP and network bending techniques. *Proceedings of the 2nd Joint Conference on AI Music Creativity*, 9. <https://doi.org/10.5281/zenodo.5137906> ↵
- Yu, Y., Srivastava, A., & Canales, S. (2021). Conditional LSTM-GAN for Melody Generation from Lyrics. *ACM Trans. Multimedia Comput. Commun. Appl.*, 17(1). <https://doi.org/10.1145/3424116> ↵
- Zhang, W.-Q., Yang, D., Liu, J., & Bao, X. (2010). Perturbation analysis of mel-frequency cepstrum coefficients. *2010 International Conference on Audio, Language and Image Processing*, 715–718. <https://doi.org/10.1109/ICALIP.2010.5685063> ↵
- zmölnig, Io., Morelli, D., & Holzmänn, G. (2013). *ann*. <https://github.com/sebshader/ann> ↵