bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

EpyNN: Educational python for Neural Networks

Florian Malard[1], Laura Danner[1], Emilie Rouzies[2], Jesse G Meyer[1], Ewen Lescop[3], and Stéphanie Olivier-Van Stichelen[1]

[1]Department of Biochemistry, Medical College of Wisconsin, Milwaukee, USA
[2]INRAE, Riverly, 69625 Villeurbanne Cedex, France
[3]Institut de Chimie des Substances Naturelles, CNRS UPR 2301, Université Paris-Saclay, LabEx LERMIT, 1 avenue de la Terrasse, 91190 Gif-sur-Yvette, France

December 6, 2021

Abstract

Summary: Artificial Neural Networks (ANNs) have achieved unequaled performance for numerous problems in many areas of Science, Business, Public Policy, and more. While experts are familiar with performance-oriented software and underlying theory, ANNs are difficult to comprehend for non-experts because it requires skills in programming, background in mathematics and knowledge of terminology and concepts. In this work, we release EpyNN, an educational python resource meant for a public willing to understand key concepts and practical implementation of scalable ANN architectures from concise, homogeneous and idiomatic source code. EpyNN contains an educational Application Programming Interface (API), educational workflows from data preparation to ANN training and a documentation website setting side-by-side code, mathematics, graphical representation and text to facilitate learning and provide teaching material. Overall, EpyNN provides basics for python-fluent individuals who wish to learn, teach or develop from scratch.

Availability: EpyNN documentation is available at https://epynn.net and repository can be retrieved from https://github.com/synthaze/epynn.

Contact: Stéphanie Olivier-Van-Stichelen, solivier@mcw.edu.

Supplementary Information: Supplementary files and listings.

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

1 Introduction

Approaches based on Artificial Neural Networks (ANNs) are implemented to solve problems in many fields with strong implications for topics of public interest such as life sciences and medicine. Therein, highly promising applications of ANNs may include drug discovery [1], gene interaction and disease prediction [2] as well as protein structure prediction [3]. However, the day-to-day integration of ANNs in data analysis workflows remains mostly tied to the expert community, well at ease with ANN theory and Application Programming Interface (API) available from performance-oriented, widespread software [4, 5, 6, 7]. Limitations in programming skills, background in mathematics or terminology and concepts likely explain why deep learning is not standard in every lab dealing with large sets of data.

Non-experts still have opportunities to become familiar with ANNs. The proficient programmer can take advantage of performance-oriented high-level APIs to achieve state-of-the-art results. This, however, requires using ANNs as black-boxes with no understanding of the inner workings. An expert programmer may find readable source codes on sharing platforms but will face highly heterogeneous contents. On the other hand, people with limited programming skills can rely on the extremely rich web documentation including mainstream articles, programming and/or mathematics oriented posts and notebooks. However, while this documentation contains useful items, they do not provide functional codes and are time-consuming.

To cope with the before-mentioned limitations, we release an integrated Educational python resource for Neural Networks (EpyNN) providing homogeneous implementations of diverse ANNs architectures. The EpyNN education-oriented API is written concisely and exhaustively commented to facilitate learning and use as teaching material. The repository contains examples of workflows from data preparation to ANNs training and prediction. EpyNN comes along with a documentation website which provides side-by-side code, mathematics and graphical representations along with standard package documentation. Overall, EpyNN is directed toward non-experts of the field who wish to learn, teach or develop from scratch.

2 Implementation

EpyNN is written in Python (3.7.1) [8] and computational flows are written in pure NumPy, the worldwide standard for array programming [9, 10]. Educational examples for data preparation and training of ANNs using EpyNN are provided as regular Python code and Jupyter notebook [11]. The EpyNN documentation website available at https://epynn.net was built with Sphinx [12] using a theme provided by ”Read the Docs” [13] and runs with Apache2 [14] over HTTP with SSL/TLS [15] on GNU/Linux Debian 10 (Buster) [16]. EpyNN is licensed under the GNU General Public License v3.0 [17] and is compatible with GNU/Linux, MacOS and Windows.

3 Features

3.1 Educational API

ANN models are built by stacking layers with distinct architectures (Figure 1). EpyNN implements a set of major layers extensively documented at https://epynn.net. This includes the following layers: *Fully connected* (Dense) [18, 19], *Recurrent Neural Network* (RNN) [20], *Gated Recurrent Unit* (GRU) [21], *Long Short-Term Memory* (LSTM) [22], *Convolution 2D* and *Pooling* layers [23, 24]. EpyNN also provides *Dropout* [25] and

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

Flatten layers for network regularization and data reshaping, respectively, as well as a *Template* pass-through layer. Finally, the input *Embedding* layer supports data normalization, one-hot encoding and mini-batches preparation for Stochastic Gradient Descent (SGD). ANN can be created using the *EpyNN* model provided with a list of layers (See https://epynn.net/epynn). Hyperparameters can be set globally for all layers or locally for each layer. Supported activation functions include the Rectifier Linear Unit (ReLU) and Leaky ReLU, Exponential Linear Unit (ELU) and logistic functions sigmoid, hyperbolic tangent and softmax (See https://epynn.net/activation). Supported loss functions include Mean Squared Error (MSE), Mean Absolute Error (MAE), Mean Squared Logarithmic Error (MSLE), Binary Cross-Entropy (BCE) and Categorical Cross-Entropy (CCE) (See https://epynn.net/loss). Evaluation metrics include accuracy, precision, recall, specificity and F-Score.

3.2 Educational Python

One layer is defined inside one directory containing four files for model definition, forward propagation, backward propagation and parameters-related functions (See https://github.com/synthaze/epynn/tree/main/epynn). Sources strictly layer mathematical definitions and therefore they are accurate and concise with definition ranging from 66 to 269 lines of codes for the *Template* and LSTM layers, respectively. The ANN model *EpyNN* and other modules for activation, loss functions, metrics and more were written following the same guidelines. Importantly, sources are extensively commented with an average of 0.35 line of comments for each line of code (Table 1). Overall, native implementations are written in idiomatic Python/NumPy with a strong focus on homogeneity across the whole package.

3.3 Educational Workflows

We provide 14 educational workflows on data type, structure and preparation along with principles to design and train ANN, both in regular Python and Jupyter notebooks formats (Table 2). In the notebooks, we introduced educational features of EpyNN's API. Among others, this includes exhaustive initialization logs reporting on layers' dimensions and shapes for input, output and processing intermediates. In addition, layers in EpyNN share the same cache structure, which makes easy content manipulation out of built-in procedures and purposes. To facilitate monitoring, we implemented colorful logs during model training along with automatic plot facilities upon completion.

3.4 Educational Website

https://epynn.net was designed to provide integrative educational material, traditional python package documentation and Jupyter notebooks. Therein, systematic descriptions put side-by-side code, mathematical notation, graphical representation and succinct text-based explanations for most object in EpyNN (Figure 2). Because ANNs may be difficult to comprehend for non-expert, we consider this multi-levels approach is key to facilitate understanding with regards to the diversity of users and backgrounds. Accordingly, we expect more individuals to find an anchor point and therefore evolve toward a global understanding. Overall, we implemented https://epynn.net to be an easy learning material and teaching support.

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

4 Related work

EpyNN should not be confounded in the crowded space of deep-learning libraries that attempt to provide a high-level API that is easy to use. Such popular deep-learning libraries are generally developed for performance in production environment and not for educational purposes (*e.g.,* Tensorflow [5], Keras [4], Torch [6], Fastai [7], and others). The main and crucial difference is simple: the source code of high-level APIs in production-focused deep-learning libraries is not written to be used as a teaching or learning material. This means that it turns extremely difficult to locate and understand the algorithms explaining the inner working of ANNs in the source code of such popular, production-focused libraries (Listing S1, S2).

Still, EpyNN is highly suited to teach and learn the backbone of other popular, production-focused libraries. EpyNN computational schemes were validated by direct comparison with Tensorflow/Keras [4] in 264 distinct configurations (File S1). While providing identical results for identical configurations, the *tensorflow/python* directory contains 1291 python files for a total of 333 527 lines of codes (File S2). By contrast, the *EpyNN/epynn* directory contains 58 files for a total of 2317 lines of code. Therefore, EpyNN provides the opportunity to read and understand every line of code behind its educational API which may remain highly similar in use compared with other libraries, including Tensorflow/Keras [4].

5 Context of Use

The main goal of EpyNN is to provide an integrated environment allowing to understand the inner working of ANNs trained with backpropagation. EpyNN brings together theory and practice through its lightweight API which relies on sources that are written to be read and modified by users. This in conjunction with an educational website that describes the functional code line by line with diagrams, mathematics and text. Said differently, there is virtually no abstraction between the source code of EpyNN and the integrated documentation.

Typically, one user would use EpyNN by setting up the following environment: one text editor session to review and customize the source code of EpyNN educational API, another session to write the executable script for ANN design and training, a web session at https://epynn.net to take advantage of the extended documentation and finally a terminal session to proceed with ANN training and further operations (Figure 3). Users may work on a local copy of EpyNN educational API sources to design exercises, to customize and compare variants of existing architectures or even to implement new architecture via the *Template* layer (See https://epynn.net/epynnlayers#template-layer).

6 Conclusion

We developed EpyNN, an education-oriented python resource aiming to encourage practice, understanding and adoption of ANNs by non-experts researchers and beyond. EpyNN features an educational API, concise python sources, educational workflows and a rich educational website. Because EpyNN is easy to customize, we anticipate and encourage deposition of additional content from push request at https://github.com/synthaze/epynn.

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

Acknowledgments

We thank Axelle Malard (https://axellemalard.com), artistic director and graphic designer who kindly offered to provide EpyNN logos and favicons. We thank Christelle Gloor (D-INFK, ETH Zurich) for useful discussions during the development of EpyNN. This work was supported by the Medical College of Wisconsin and the National Institute of Health (R00 HD087430).

References

[1] Igor I Baskin, David Winkler, and Igor V Tetko. A renaissance of neural networks in drug discovery. *Expert opinion on drug discovery*, 11(8):785–795, 2016.

[2] Giulia Muzio, Leslie O'Bray, and Karsten Borgwardt. Biological network analysis with deep learning. *Briefings in bioinformatics*, 22(2):1515–1530, 2021.

[3] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, pages 1–11, 2021.

[4] François Chollet et al. keras, 2015.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[6] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[7] Jeremy Howard and Sylvain Gugger. Fastai: a layered api for deep learning. *Information*, 11(2):108, 2020.

[8] Python Software Foundation. Python 3.7.1. `https://www.python.org/downloads/release/python-371/`, 2018.

[9] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in science & engineering*, 13(2):22–30, 2011.

[10] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[11] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. *Jupyter Notebooks-a publishing format for reproducible computational workflows.*, volume 2016. 2016.

[12] Georg Brandl. Sphinx. python documentation generator. `https://www.sphinx-doc.org/en/master/index.html`, 2007.

[13] Inc & contributors Read the Docs. Read the docs. `https://readthedocs.org/`, 2007.

[14] The Apache Software Foundation. Apache. http server project. `https://httpd.apache.org/`, 2007.

[15] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. The cost of the "s" in https. *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 133–140, 2014.

[16] The community-supported Debian Project. Debian. the universal operating system. `https://www.debian.org/`, 2007.

[17] Free Software Foundation. Gnu general public license v3.0. `https://www.gnu.org/licenses/gpl-3.0.html`, 2007.

[18] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[19] Frank Emmert-Streib, Zhen Yang, Han Feng, Shailesh Tripathi, and Matthias Dehmer. An introductory review of deep learning for prediction models with big data. *Frontiers in Artificial Intelligence*, 3:4, 2020.

[20] John A Bullinaria. Recurrent neural networks. *Neural Computation: Lecture*, 12, 2013.

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

[21] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[23] Samer Hijazi, Rishi Kumar, Chris Rowen, et al. Using convolutional neural networks for image recognition. *Cadence Design Systems Inc.: San Jose, CA, USA*, pages 1–12, 2015.

[24] Manli Sun, Zhanjie Song, Xiaoheng Jiang, Jing Pan, and Yanwei Pang. Learning pooling for convolutional neural network. *Neurocomputing*, 224:96–104, 2017.

[25] Pierre Baldi and Peter J Sadowski. Understanding dropout. *Advances in neural information processing systems*, 26:2814–2822, 2013.

[26] Eugenia Wulff-Fuentes, Rex R Berendt, Logan Massman, Laura Danner, Florian Malard, Jeet Vora, Robel Kahsay, and Stephanie Olivier-Van Stichelen. The human o-glcnacome database and meta-analysis. *Scientific data*, 8(1):1–11, 2021.

[27] Florian Malard, Eugenia Wulff-Fuentes, Rex R Berendt, Guillaume Didier, and Stéphanie Olivier-Van Stichelen. Automatization and self-maintenance of the o-glcnacome catalog: a smart scientific database. *Database*, 2021, 2021.

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

Figures and Tables

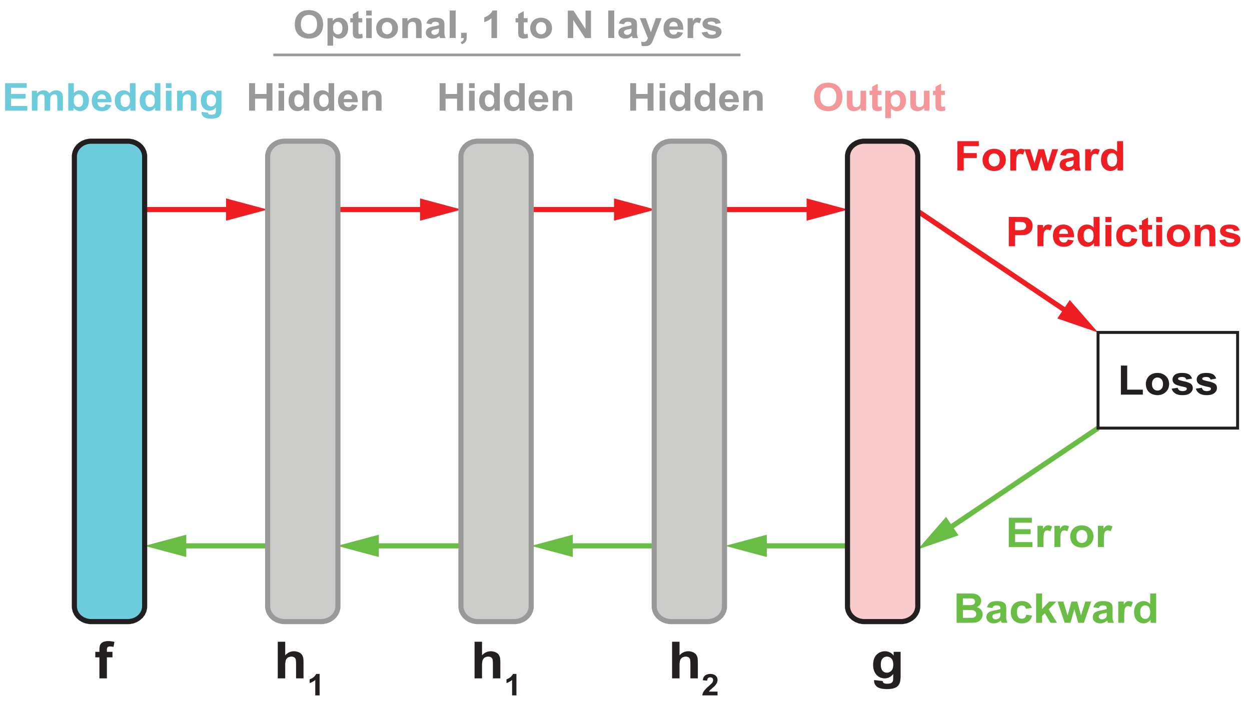
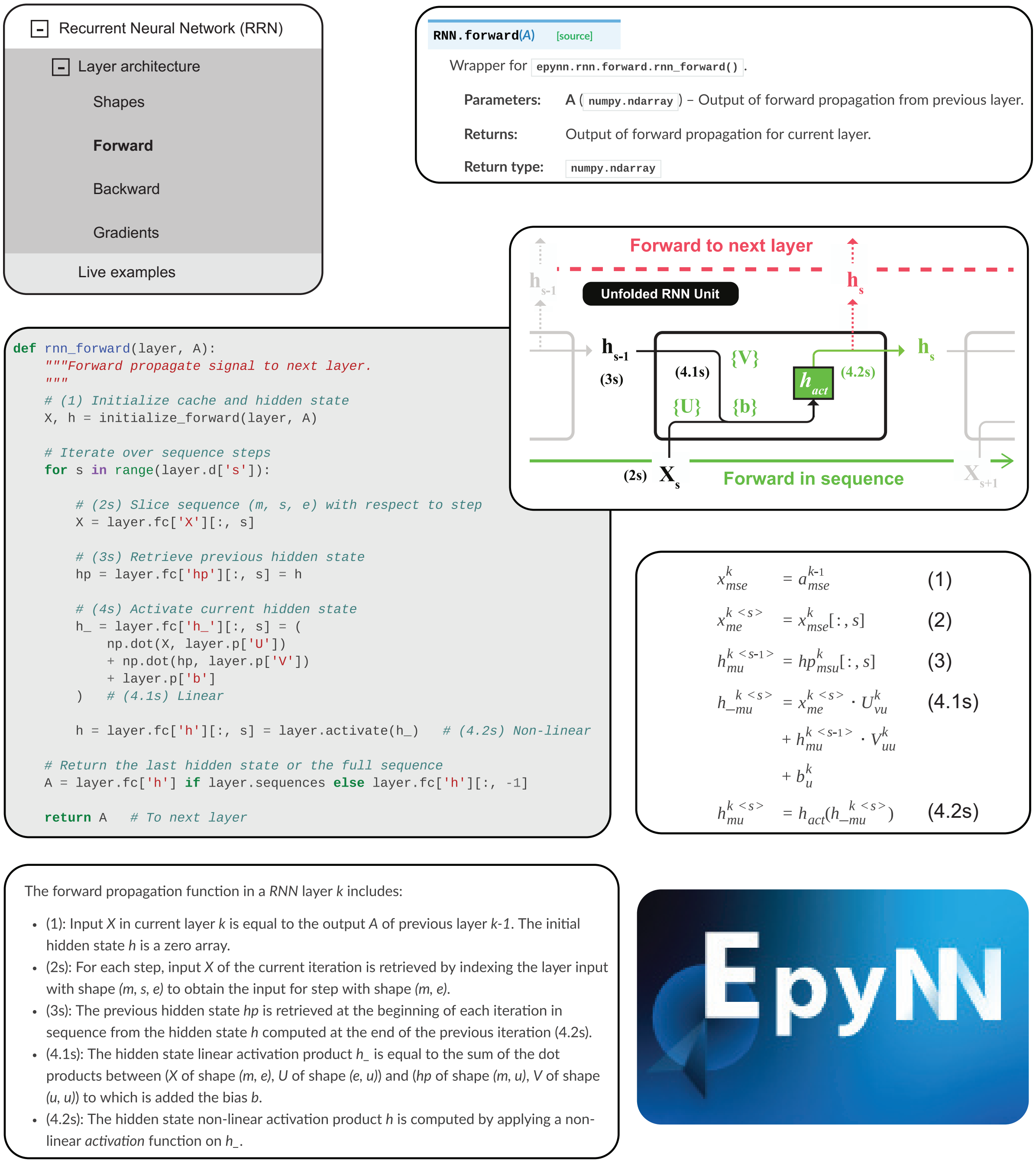


Figure 1: **ANN model and layers.** Here is depicted a set of five layers, equivalent to a set of five functions $\{f, h_1, h_1, h_2, g\}$. This set contains a subset of four distinct layer architectures $\{f, h_1, h_2, g\}$ because two layers are defined by the same function h_1. The ANN model featuring the set of five stacked layers is the composite function $(g \circ h_2 \circ h_1 \circ h_1 \circ f)$ which takes input and returns predictions (forward, red) further compared to target values through the derivative of a loss function (e.g., Mean Squared Error). The error gradient is propagated backward through each layer and, if applicable, trainable parameters are updated accordingly (backward, green). See https://epynn.net/epynn for more details.

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.



- Recurrent Neural Network (RRN)
  - Layer architecture
    - Shapes
    - **Forward**
    - Backward
    - Gradients
  - Live examples

RNN.forward(A) [source]

Wrapper for `epynn.rnn.forward.rnn_forward()`.

Parameters: A (`numpy.ndarray`) – Output of forward propagation from previous layer.

Returns: Output of forward propagation for current layer.

Return type: `numpy.ndarray`

```
def rnn_forward(layer, A):
    """Forward propagate signal to next layer.
    """
    # (1) Initialize cache and hidden state
    X, h = initialize_forward(layer, A)

    # Iterate over sequence steps
    for s in range(layer.d['s']):

        # (2s) Slice sequence (m, s, e) with respect to step
        X = layer.fc['X'][:, s]

        # (3s) Retrieve previous hidden state
        hp = layer.fc['hp'][:, s] = h

        # (4s) Activate current hidden state
        h_ = layer.fc['h_'][:, s] = (
            np.dot(X, layer.p['U'])
            + np.dot(hp, layer.p['V'])
            + layer.p['b']
        )   # (4.1s) Linear

        h = layer.fc['h'][:, s] = layer.activate(h_)   # (4.2s) Non-linear

    # Return the last hidden state or the full sequence
    A = layer.fc['h'] if layer.sequences else layer.fc['h'][:, -1]

    return A   # To next layer
```

$$x^{k}_{mse} = a^{k-1}_{mse} \qquad (1)$$

$$x^{k\,<s>}_{me} = x^{k}_{mse}[:,s] \qquad (2)$$

$$h^{k\,<s-1>}_{mu} = hp^{k}_{msu}[:,s] \qquad (3)$$

$$h\_^{k\,<s>}_{mu} = x^{k\,<s>}_{me} \cdot U^{k}_{vu} + h^{k\,<s-1>}_{mu} \cdot V^{k}_{uu} + b^{k}_{u} \qquad (4.1s)$$

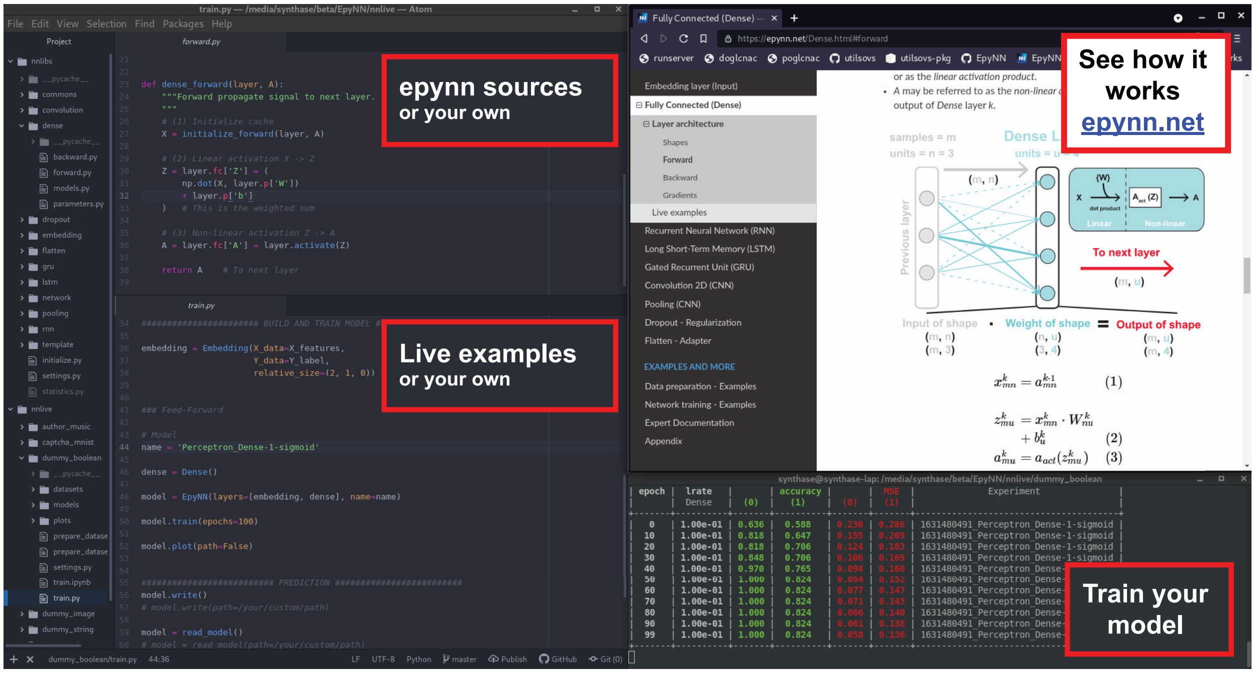
$$h^{k\,<s>}_{mu} = h_{act}(h\_^{k\,<s>}_{mu}) \qquad (4.2s)$$

The forward propagation function in a *RNN* layer *k* includes:

- (1): Input *X* in current layer *k* is equal to the output *A* of previous layer *k-1*. The initial hidden state *h* is a zero array.
- (2s): For each step, input *X* of the current iteration is retrieved by indexing the layer input with shape *(m, s, e)* to obtain the input for step with shape *(m, e)*.
- (3s): The previous hidden state *hp* is retrieved at the beginning of each iteration in sequence from the hidden state *h* computed at the end of the previous iteration (4.2s).
- (4.1s): The hidden state linear activation product *h_* is equal to the sum of the dot products between (*X* of shape *(m, e)*, *U* of shape *(e, u)*) and (*hp* of shape *(m, u)*, *V* of shape *(u, u)*) to which is added the bias *b*.
- (4.2s): The hidden state non-linear activation product *h* is computed by applying a non-linear *activation* function on *h_*.

Figure 2: **EpyNN https://epynn.net educational website - Example of integrated documentation.** The EpyNN educational API is documented with systematic descriptions involving classic API documentation and corresponding source codes explained line-by-line via direct translation into diagrams, mathematics and text. This scheme is shown for the RNN layer and is extracted from https://epynn.net/RNN. Objects and methods/functions related to network and layer models or activation and loss functions, among others, were documented following this scheme. See https://epynn.net/glossary#notations for conventions about notations.

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.



epynn sources
or your own

```python
def dense_forward(layer, A):
    """Forward propagate signal to next layer.
    """
    # (1) Initialize cache
    X = initialize_forward(layer, A)

    # (2) Linear activation X -> Z
    Z = layer.fc['Z'] = (
        np.dot(X, layer.p['W'])
        + layer.p['b']
    )   # This is the weighted sum

    # (3) Non-linear activation Z -> A
    A = layer.fc['A'] = layer.activate(Z)

    return A    # To next layer
```

Live examples
or your own

```python
###################### BUILD AND TRAIN MODEL #

embedding = Embedding(X_data=X_features,
                      Y_data=Y_label,
                      relative_size=(2, 1, 0))


### Feed-Forward

# Model
name = 'Perceptron_Dense-1-sigmoid'

dense = Dense()

model = EpyNN(layers=[embedding, dense], name=name)

model.train(epochs=100)

model.plot(path=False)


######################## PREDICTION ########################
model.write()
# model.write(path=/your/custom/path)

model = read_model()
# model = read_model(path=/your/custom/path)
```

See how it works
epynn.net

or as the *linear activation product*.

- *A* may be referred to as the *non-linear a* output of *Dense* layer *k*.

$$x^k_{mn} = a^{k\text{-}1}_{mn} \qquad (1)$$

$$z^k_{mu} = x^k_{mn} \cdot W^k_{nu} + b^k_u \qquad (2)$$

$$a^k_{mu} = a_{act}(z^k_{mu}) \qquad (3)$$

synthase@synthase-lap: /media/synthase/beta/EpyNN/nnlive/dummy_boolean

```
| epoch |  lrate   |       | accuracy |       |  MSE  |              Experiment               |
|       |  Dense   |  (0)  |   (1)    |  (0)  |  (1)  |                                       |
+-------+----------+-------+----------+-------+-------+---------------------------------------+
|   0   | 1.00e-01 | 0.636 |  0.588   | 0.230 | 0.286 | 1631480491_Perceptron_Dense-1-sigmoid |
|  10   | 1.00e-01 | 0.818 |  0.647   | 0.155 | 0.209 | 1631480491_Perceptron_Dense-1-sigmoid |
|  20   | 1.00e-01 | 0.818 |  0.706   | 0.124 | 0.183 | 1631480491_Perceptron_Dense-1-sigmoid |
|  30   | 1.00e-01 | 0.848 |  0.706   | 0.106 | 0.169 | 1631480491_Perceptron_Dense-1-sigmoid |
|  40   | 1.00e-01 | 0.970 |  0.765   | 0.094 | 0.160 | 1631480491_Perceptron_Dense-
|  50   | 1.00e-01 | 1.000 |  0.824   | 0.084 | 0.152 | 1631480491_Perceptron_Dense-
|  60   | 1.00e-01 | 1.000 |  0.824   | 0.077 | 0.147 | 1631480491_Perceptron_Dense-
|  70   | 1.00e-01 | 1.000 |  0.824   | 0.071 | 0.143 | 1631480491_Perceptron_Dense-
|  80   | 1.00e-01 | 1.000 |  0.824   | 0.066 | 0.140 | 1631480491_Perceptron_Dense-
|  90   | 1.00e-01 | 1.000 |  0.824   | 0.061 | 0.138 | 1631480491_Perceptron_Dense-
|  99   | 1.00e-01 | 1.000 |  0.824   | 0.058 | 0.136 | 1631480491_Perceptron_Dense-
+-------+----------+-------+----------+-------+-------+--------
```

Train your model

Figure 3: **How to use EpyNN - Suggestion of interface.** The interface is made of text editor sessions for EpyNN *epynn* library (left, top) and live examples (left, bottom), EpyNN https://epynn.net educational website (right, top) and a terminal session to run EpyNN (right, bottom). EpyNN sources can refer to the root module files or a working copy in the current directory. Note that new layer architectures can be implemented on-the-fly from existing and template layers. See https://epynn.net/epynnlayers and https://epynn.net/quickstart#how-to-use-epynn for more documentation.

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

Subdirectory	Files	Lines	Docstring	Code	Inline	Block	Comment	Description
network	8	601	151	325	4	125	0.4	ANN model, methods e.g., training, prediction
embedding	5	318	108	176	6	34	0.23	Input layer Pre-processing
convolution	4	274	58	158	19	58	0.49	4D - Images Through space
dense	4	181	54	108	16	19	0.32	2D - Series Historical concept
dropout	4	153	54	80	6	19	0.31	Regularization layer Prevent overfitting
flatten	4	135	54	66	6	15	0.32	Adapter layer Reshape 3D+ to 2D
gru	4	316	58	219	46	39	0.39	3D - Sequences Extended memory
lstm	4	377	62	269	64	46	0.41	3D - Sequences Extended memory
pooling	4	223	56	113	9	54	0.56	4D - Images Compression layer Features extraction
rnn	4	241	58	155	30	28	0.37	3D - Sequences e.g., Time series, text
template	4	134	56	66	6	12	0.27	Pass-through layer, template to customize
commons	9	1107	429	571	44	107	0.26	Common modules e.g., metrics, loss...
Total	58	4060	1198	2306	256	556	0.35	-

Table 1: **EpyNN *epynn* library tree - Description and source code statistics**. The educational library module in EpyNN repository corresponds to the *epynn* directory. Each subdirectory except *network* and *commons* implement a specific layer architecture. RNN, LSTM and GRU are so-called recurrent layers. Files: Number of files in each subdirectory. Lines: Total number of lines in files, excluding blank lines. Docstring: Lines accounting for documentation strings or ”global” code comments. Code: Lines accounting for executable python code. Inline, Block: Lines accounting for ”local” code comments. Comment: Is equal to $\frac{Inline+Block}{Code}$ or lines of ”local” code comments for one line of executable code. Review the *epynnlive* directory at https://github.com/synthaze/epynn/tree/main/epynn.

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

Subdirectory	Files	Lines	Docstring	Code	Block	Comment	Models	Description
dummy_boolean	3	122	26	64	32	0.52	1	Python basics, Boolean, Perceptron
dummy_image	3	174	32	100	42	0.42	2	Image definition and generation, CNN
dummy_string	3	169	26	106	37	0.35	4	String data type, one-hot encoding, recurrent layers
dummy_time	3	196	34	119	43	0.36	4	Time series, signal detection, recurrent layers
author_music	3	209	23	138	48	0.35	3	WAV audio files manipulation, recurrent layers
captcha_mnist	3	148	13	104	31	0.3	2	IDX format, encoded image, CNN
ptm_protein	3	178	13	127	38	0.3	4	*O*-GlcNAcylated peptides [26, 27], recurrent layers
Total	21	1196	167	758	271	0.36	20	-

Table 2: **EpyNN *epynnlive* examples tree - Description and source code statistics**. Educational workflows for data preparation and model training are located inside the *epynnlive* directory in the EpyNN repository. Each subdirectory contains 3 python files *prepare_dataset.py*, *train.py* and *settings.py* and Jupyter notebooks versions *prepare_dataset.ipynb* and *train.ipynb* not accounted here. In descrption, CNN refers to Convolutional Neural Network. Lines: Total number of lines in files, excluding blank lines. Docstring: Lines accounting for documentation strings or "global" code comments. Code: Lines accounting for executable python code. Block: Lines accounting for "local" code comments. Comment: Is equal to $\frac{Block}{Code}$ or lines of "local" code comments for one line of executable code. Models: Number of distinct ANN configuration reviewed in *train.py*. Review the *epynnlive* directory at https://github.com/synthaze/epynn/tree/main/epynnlive and Jupyter notebooks for data preparation at https://epynn.net/nbdata and model training at https://epynn.net/nbtraining.

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

Supplementary Material for:
EpyNN: Educational python for Neural Networks

Supplementary files

Supplementary file S1

FileS1.zip: Contains executable python codes used for validation of EpyNN against Tensorflow/Keras using tensorflow==2.3.0 pip package. See README for details. Note that a summary of results is available at https://epynn.net/index #is-epynn-reliable.

Supplementary file S2

FileS2.zip: XLSX file summary and python script used to count the number of files and lines of code within tensorflow/python directory from tensorflow==2.3.0 pip package. By lines of code we mean all but not blank lines, docstrings and block comments.

Supplementary listings

Supplementary listing S1

```python
def lstm_forward(layer, A):
    """Forward propagate signal to next layer.
    """
    # (1) Initialize cache, hidden and memory states
    X, h, C_ = initialize_forward(layer, A)

    # Iterate over sequence steps
    for s in range(layer.d['s']):

        # (2s) Slice sequence (m, s, e) w.r.t to step
        X = layer.fc['X'][:, s]

        # (3s) Retrieve previous states
        hp = layer.fc['hp'][:, s] = h       # (3.1s) Hidden
        Cp_ = layer.fc['Cp_'][:, s] = C_    # (3.2s) Memory

        # (4s) Activate forget gate
        f_ = layer.fc['f_'][:, s] = (
            np.dot(X, layer.p['Uf'])
            + np.dot(hp, layer.p['Vf'])
            + layer.p['bf']
        )   # (4.1s)

        f = layer.fc['f'][:, s] = layer.activate_forget(f_)      # (4.2s)

        # (5s) Activate input gate
        i_ = layer.fc['i_'][:, s] = (
            np.dot(X, layer.p['Ui'])
```

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

```
            + np.dot(hp, layer.p['Vi'])
            + layer.p['bi']
        )   # (5.1s)

        i = layer.fc['i'][:, s] = layer.activate_input(i_)       # (5.2s)

        # (6s) Activate candidate
        g_ = layer.fc['g_'][:, s] = (
            np.dot(X, layer.p['Ug'])
            + np.dot(hp, layer.p['Vg'])
            + layer.p['bg']
        )   # (6.1s)

        g = layer.fc['g'][:, s] = layer.activate_candidate(g_)   # (6.2s)

        # (7s) Activate output gate
        o_ = layer.fc['o_'][:, s] = (
            np.dot(X, layer.p['Uo'])
            + np.dot(hp, layer.p['Vo'])
            + layer.p['bo']
        )   # (7.1s)

        o = layer.fc['o'][:, s] = layer.activate_output(o_)      # (7.2s)

        # (8s) Compute current memory state
        C_ = layer.fc['C_'][:, s] = (
            Cp_ * f
            + i * g
        )   # (8.1s)

        C = layer.fc['C'][:, s] = layer.activate(C_)             # (8.2s)

        # (9s) Compute current hidden state
        h = layer.fc['h'][:, s] = o * C

    # Return the last hidden state or the full sequence
    A = layer.fc['h'] if layer.sequences else layer.fc['h'][:, -1]

    return A    # To next layer
```

Listing S1: **EpyNN sources for *epynn.lstm.forward.lstm_forward()***. The function describes the forward propagation algorithm for the LSTM layer in EpyNN. This function is easy to find in the source of EpyNN because it is simply located within the *epynn.lstm.forward* module which contains a total of two functions for less than hundred lines of code. Moreover, it is easy to understand because written in idiomatic Python/NumPy and exhaustively commented. Finally, extended documentation on this code is available at https://epynn.net/LSTM#forward. See https://epynn.net/glossary#notations for conventions about notations.

Supplementary listing S2

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

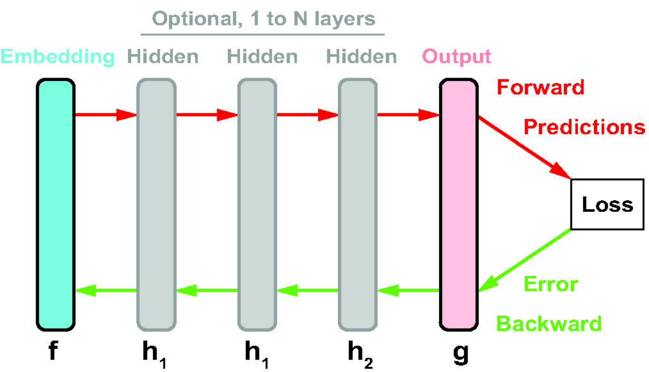
```
def call(self, inputs, states, training=None):
  h_tm1 = states[0]  # previous memory state
  c_tm1 = states[1]  # previous carry state

  dp_mask = self.get_dropout_mask_for_cell(inputs, training, count=4)
  rec_dp_mask = self.get_recurrent_dropout_mask_for_cell(
      h_tm1, training, count=4)

  if self.implementation == 1:
    if 0 < self.dropout < 1.:
      inputs_i = inputs * dp_mask[0]
      inputs_f = inputs * dp_mask[1]
      inputs_c = inputs * dp_mask[2]
      inputs_o = inputs * dp_mask[3]
    else:
      inputs_i = inputs
      inputs_f = inputs
      inputs_c = inputs
      inputs_o = inputs
    k_i, k_f, k_c, k_o = array_ops.split(
        self.kernel, num_or_size_splits=4, axis=1)
    x_i = K.dot(inputs_i, k_i)
    x_f = K.dot(inputs_f, k_f)
    x_c = K.dot(inputs_c, k_c)
    x_o = K.dot(inputs_o, k_o)
    if self.use_bias:
      b_i, b_f, b_c, b_o = array_ops.split(
          self.bias, num_or_size_splits=4, axis=0)
      x_i = K.bias_add(x_i, b_i)
      x_f = K.bias_add(x_f, b_f)
      x_c = K.bias_add(x_c, b_c)
      x_o = K.bias_add(x_o, b_o)

    if 0 < self.recurrent_dropout < 1.:
      h_tm1_i = h_tm1 * rec_dp_mask[0]
      h_tm1_f = h_tm1 * rec_dp_mask[1]
      h_tm1_c = h_tm1 * rec_dp_mask[2]
      h_tm1_o = h_tm1 * rec_dp_mask[3]
    else:
      h_tm1_i = h_tm1
      h_tm1_f = h_tm1
      h_tm1_c = h_tm1
      h_tm1_o = h_tm1
    x = (x_i, x_f, x_c, x_o)
    h_tm1 = (h_tm1_i, h_tm1_f, h_tm1_c, h_tm1_o)
    c, o = self._compute_carry_and_output(x, h_tm1, c_tm1)
  else:
```

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.

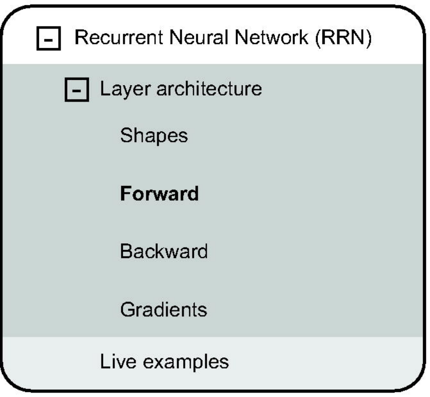
```
      if 0. < self.dropout < 1.:
        inputs = inputs * dp_mask[0]
      z = K.dot(inputs, self.kernel)
      z += K.dot(h_tm1, self.recurrent_kernel)
      if self.use_bias:
        z = K.bias_add(z, self.bias)

      z = array_ops.split(z, num_or_size_splits=4, axis=1)
      c, o = self._compute_carry_and_output_fused(z, c_tm1)

    h = o * self.activation(c)
    return h, [h, c]
```

Listing S2: **Tensorflow/Keras sources for *tensorflow.python.keras.layers.recurrent.LSTMCell.call()*.** The method describes the forward propagation algorithm for the LSTM cell in Tesorflow/Keras. This method is not easy to find in the source of Tensorflow/Keras because of the number of recursion levels down to *tensorflow.python.keras.layers.recurrent* module which contains 34 objects, 200 methods and 1 766 lines of code. In addition, there is no block or inline comment in this code.

bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.



bioRxiv preprint doi: https://doi.org/10.1101/2021.12.06.470764; this version posted December 7, 2021. The copyright holder for this preprint (which was not certified by peer review) is the author/funder. All rights reserved. No reuse allowed without permission.



⊟ Recurrent Neural Network (RRN)

⊟ Layer architecture

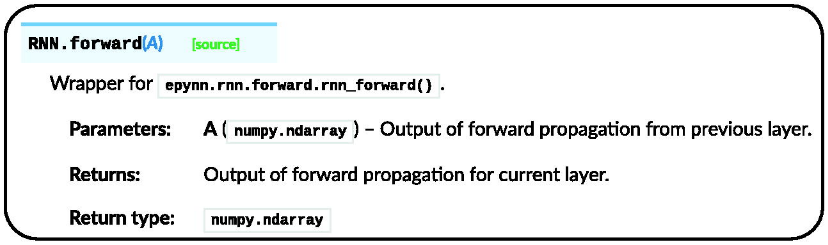
Shapes

Forward

Backward

Gradients

Live examples



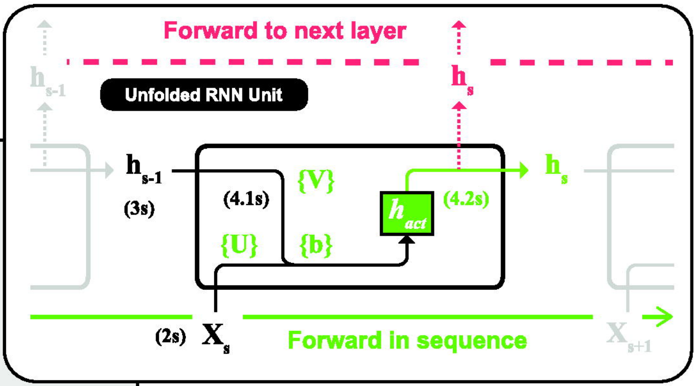
RNN.forward(*A*) [source]

Wrapper for `epynn.rnn.forward.rnn_forward()`.

Parameters: **A** (`numpy.ndarray`) – Output of forward propagation from previous layer.

Returns: Output of forward propagation for current layer.

Return type: `numpy.ndarray`



```
def rnn_forward(layer, A):
    """Forward propagate signal to next layer.
    """
    # (1) Initialize cache and hidden state
    X, h = initialize_forward(layer, A)

    # Iterate over sequence steps
    for s in range(layer.d['s']):

        # (2s) Slice sequence (m, s, e) with respect to step
        X = layer.fc['X'][:, s]

        # (3s) Retrieve previous hidden state
        hp = layer.fc['hp'][:, s] = h

        # (4s) Activate current hidden state
        h_ = layer.fc['h_'][:, s] = (
            np.dot(X, layer.p['U'])
            + np.dot(hp, layer.p['V'])
            + layer.p['b']
        )   # (4.1s) Linear

        h = layer.fc['h'][:, s] = layer.activate(h_)   # (4.2s) Non-linear

    # Return the last hidden state or the full sequence
    A = layer.fc['h'] if layer.sequences else layer.fc['h'][:, -1]

    return A   # To next layer
```

$$x^{k}_{mse} = a^{k-1}_{mse} \quad (1)$$

$$x^{k\,<s>}_{me} = x^{k}_{mse}[:, s] \quad (2)$$

$$h^{k\,<s-1>}_{mu} = hp^{k}_{msu}[:, s] \quad (3)$$

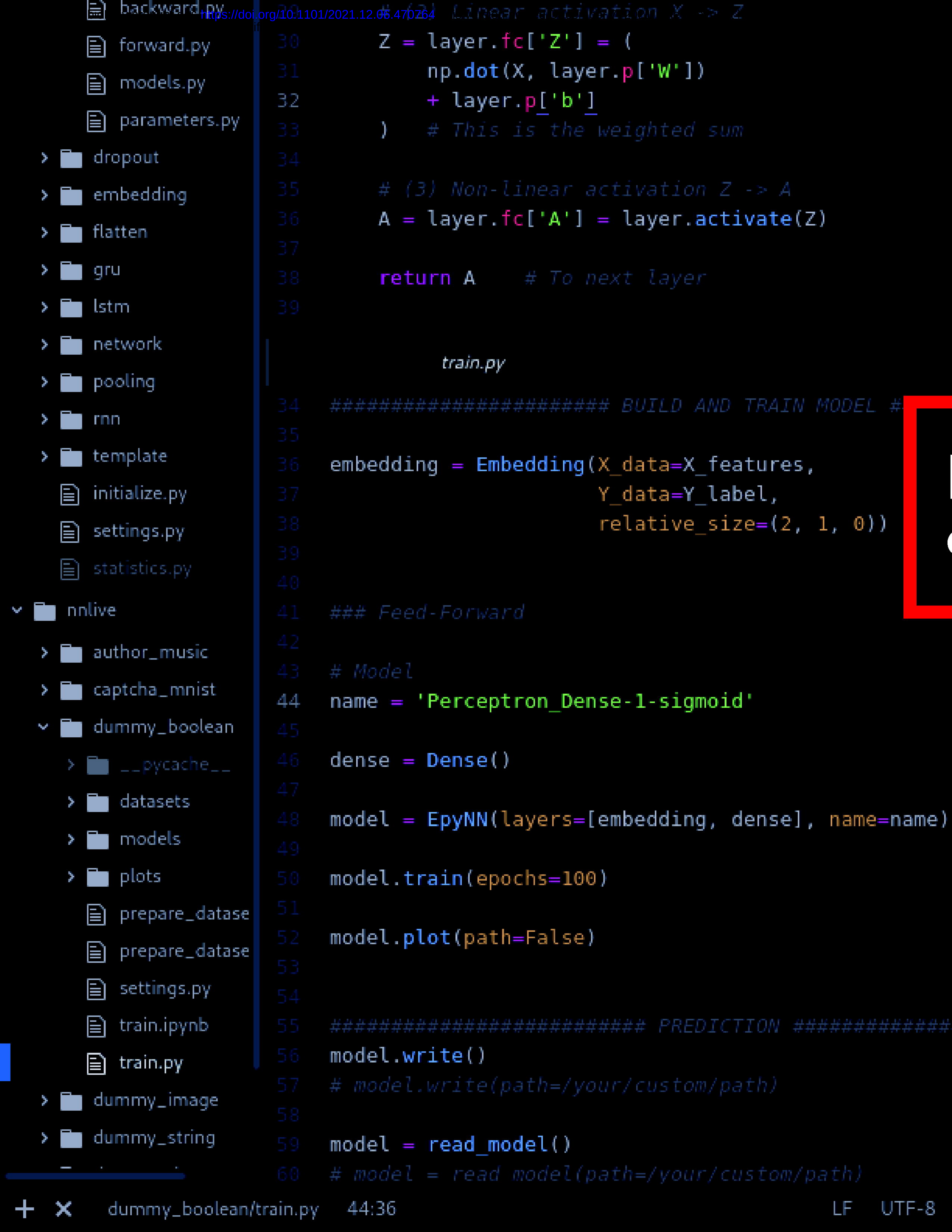
$$h^{k\,<s>}_{\_mu} = x^{k\,<s>}_{me} \cdot U^{k}_{vu} + h^{k\,<s-1>}_{mu} \cdot V^{k}_{uu} + b^{k}_{u} \quad (4.1s)$$

$$h^{k\,<s>}_{mu} = h_{act}(h^{k\,<s>}_{\_mu}) \quad (4.2s)$$

The forward propagation function in a *RNN* layer *k* includes:

- (1): Input *X* in current layer *k* is equal to the output *A* of previous layer *k-1*. The initial hidden state *h* is a zero array.
- (2s): For each step, input *X* of the current iteration is retrieved by indexing the layer input with shape *(m, s, e)* to obtain the input for step with shape *(m, e)*.
- (3s): The previous hidden state *hp* is retrieved at the beginning of each iteration in sequence from the hidden state *h* computed at the end of the previous iteration (4.2s).
- (4.1s): The hidden state linear activation product *h_* is equal to the sum of the dot products between (*X* of shape *(m, e)*, *U* of shape *(e, u)*) and (*hp* of shape *(m, u)*, *V* of shape *(u, u)*) to which is added the bias *b*.
- (4.2s): The hidden state non-linear activation product *h* is computed by applying a non-linear *activation* function on *h_*.

https://doi.org/10.1101/2021.12.06.470264



backward.py
forward.py
models.py
parameters.py
dropout
embedding
flatten
gru
lstm
network
pooling
rnn
template
initialize.py
settings.py
statistics.py
nnlive
author_music
captcha_mnist
dummy_boolean
__pycache__
datasets
models
plots
prepare_datase
prepare_datase
settings.py
train.ipynb
train.py
dummy_image
dummy_string

```
    # (2) Linear activation X -> Z
    Z = layer.fc['Z'] = (
        np.dot(X, layer.p['W'])
        + layer.p['b']
    )   # This is the weighted sum

    # (3) Non-linear activation Z -> A
    A = layer.fc['A'] = layer.activate(Z)

    return A    # To next layer
```

train.py

```
####################### BUILD AND TRAIN MODEL #

embedding = Embedding(X_data=X_features,
                      Y_data=Y_label,
                      relative_size=(2, 1, 0))


### Feed-Forward

# Model
name = 'Perceptron_Dense-1-sigmoid'

dense = Dense()

model = EpyNN(layers=[embedding, dense], name=name)

model.train(epochs=100)

model.plot(path=False)


########################## PREDICTION ############
model.write()
# model.write(path=/your/custom/path)

model = read_model()
# model = read_model(path=/your/custom/path)
```