The `EsnTorch` Library: Efficient Implementation of Transformer-Based Echo State Networks

Jérémie Cabessa[1,2,3], Hugo Hernault[1], Yves Lamonato[1], Mathieu Rochat[1,4], and Yariv Z. Levy[1]

[1] Playtika Ltd., CH-1003 Lausanne, Switzerland

[2] Laboratory DAVID, UVSQ – Université Paris-Saclay
78000 Versailles, France

[3] Institute of Computer Science of the Czech Academy of Sciences
8207 Prague 8, Czech Republic

[4] Mathematics Section (SMA), EPFL, 1015 Lausanne, Switzerland

`jeremie.cabessa@uvsq.fr`, `{hugoh,yvesl,yarivl}@playtika.com`, `mathieu.rochat@epfl.ch`

Abstract. Transformer-based models have revolutionized NLP. But in general, these models are highly resource consuming. Based on this consideration, several reservoir computing approaches to NLP have shown promising results. In this context, we propose `EsnTorch`, a library that implements echo state networks (ESNs) with transformer-based embeddings for text classification. `EsnTorch` is developed in `PyTorch`, optimized to work on GPU, and compatible with the `transformers` and `datasets` libraries from Hugging Face: the major data science platform for NLP. Accordingly, our library can make use of all the models and datasets available from Hugging Face. A transformer-based ESN implemented in `EsnTorch` consists of four building blocks: (1) An embedding layer, which uses a transformer-based model to embed the input texts; (2) A reservoir layer, which can implements three kinds of reservoirs: recurrent, linear or null; (3) A pooling layer, which offers three kinds of pooling strategies: mean, last, or None; (4) And a learning algorithm block, which provides six different supervised learning algorithms. Overall, this work falls within the context of sustainable models for NLP.

Keywords: reservoir computing · echo state networks · natural language processing (NLP) · text classification · transformers · BERT · python library · Hugging Face.

1 Introduction

In 2017, the *transformer* model opened the way for a new generation of language models [29]. A transformer consists of encoder-decoder blocks augmented with a self-attention mechanism. This architecture solves parallelization and long-term dependency issues encountered by classical recurrent neural networks (like LSTMs, GRUs, etc.). The transformer gave rise to a multitude of models that broke the barriers of NLP. In particular, the BERT model, which is composed of several encoder blocks, achieves impressive performance on most common NLP tasks [3]. In its pre-trained form, BERT can be used as a powerful word or sentence dynamic embedding, taking over several previous pre-trained embeddings, like word2vec, GloVe, FastText, and ELMo.

But the transformer-based models are highly resource consuming. For instance, BERT contains from 110M to 340M parameters. And while the pre-trained model is available off-the-shelf, the fine-tuning process remains computationally expensive. In an effort to address these drawbacks, lighter and faster versions of BERT have been proposed [20,26].

The issues of high model complexity and expensive fine-tuning process have been addressed from the perspective of *reservoir computing (RC)*, and more particularly, using *echo state networks (ESNs)* [12–15]. An ESN is composed of an inputs layer, a random recurrent reservoir on neurons, and a output layer. During training, the input and reservoir weights are kept fixed, and only the output weights are learned, usually via simple regression methods. The recurrent architecture of the reservoir provides the memory necessary to the handling of textual data. Their fast and light training process counterbalances the high computational cost of transformer-based models. ESNs have been applied successfully to a large variety of machine learning problems [17,18]. Recently, deep ESNs have been introduced [6].

In the context of NLP, several studies based on echo state networks have already been conducted (see [1] for further details on these related works). For instance, a biologically inspired reservoir computing approach to grammatical inference, semantic representation, and language acquisition with applications in human-robot interaction has been proposed [5,8–11,27]. On the machine learning side, ESNs have been considered for automatic speech recognition, showing that decoders can be replaced by ESNs without performance drop [24]. ESNs have also been applied to named entity recognition (NER) [19] and authorship attribution [22]. Attention-based ESNs with FastText embedding as inputs have been proposed and successfully applied to question classification [4]. ESNs have also been considered in the general context of text classification, using either static GloVe or dynamic BERT embedding as inputs [1,2]. These ESNs achieve good accuracy with particularly fast training times. Besides, ESNs have been considered as one among other fast methods for computing sentence representations, using the pre-trained word embeddings FastText and GloVe as inputs [30]. This work shows that the quality of the pre-trained word embedding plays a crucial role in the performance of the subsequent encoder that builds upon it. Finally, a different reservoir computing approach to transformers has also been proposed [23]. In this work, the so-called reservoir transformers achieve better performance-efficiency trade-offs than classical transformers.

Based on these considerations as well as on recent studies from ours [1, 2], we propose `EsnTorch`, a library that implements echo state networks (ESNs) with transformer-based embeddings as inputs, in the context of text classification. `EsnTorch` is developed in `PyTorch` and optimized to work on GPU in a parallelized way. `EsnTorch` operates in conjunction with the `transformers` and `datasets` libraries from Hugging Face: the major data science platform for NLP. Accordingly, it can make use of the 60K models and 7K datasets available from this platform. A transformer-based ESN implemented in `EsnTorch` consists of four building blocks: (1) An embedding layer which uses a transformer-based model to embed the input texts; (2) A reservoir layer which implements three kinds of reservoirs: recurrent, linear or null; (3) A pooling layer which offers three kinds of pooling strategies: mean, last, or None; (4) And a learning algorithm block which provides six different supervised learning algorithms. We believe that the combined transformer-ESN approach to NLP proposed in this work offers major advantages in terms of computational efficiency. Overall, this study falls within the context of sustainable models for NLP. `EsnTorch` is available on GitHub at the following address: https://github.com/PlaytikaResearch/esntorch.

2 Related works

Several Python libraries targeted to the implementation of ESNs already exist, but to the best of our knowledge, none of them possess the combined features of being implemented in `PyTorch`, optimized to operate on GPU, specifically targeted for NLP, and compatible with Hugging Face.

In particular, `ReservoirPy` is a complete, well designed and user-friendly library for reservoir computing implemented in `numpy` [28]. The library contains several attractive features: offline and online training, parallel implementation, sparse matrix computation, advanced learning rules, and compatibility with `hyperopt` for hyperparameter tuning. `DeepESN`, `PyRCN` and `easyesn` are three libraries for deep ESNs and ESNs, respectively, also implemented in `numpy`. The two last ones are compatible with `scikit-learn` [25]. In addition, `EchoTorch` is a very complete library for ESNs implemented in `PyTorch` [21]. It has been used for an NLP application [22]. PyTorch-ESN is a well-designed `PyTorch` module implementing Echo State Networks. The readout is trainable by ridge regression or by PyTorch's optimizers. Implementation of deep ESNs is also possible.

3 ESNs for text classification

Echo state networks. An *leaky integrator echo state network (ESN)* is a recurrent neural network composed of N_u input units, N_x hidden units referred to as the *reservoir*, and N_y output units. The input units are linked to the reservoir (weights $\mathbf{W}_{\text{in}}$), the reservoir is recurrently connected (weights $\mathbf{W}_{\text{res}}$ and bias $\mathbf{b}_{\text{res}}$), and projects onto the output units (weights $\mathbf{W}_{\text{out}}$ and bias $\mathbf{b}_{\text{out}}$).

The input, reservoir and output states of the network at time $t > 0$ are denoted by $\mathbf{u}(t) \in \mathbb{R}^{N_u}$, $\mathbf{x}(t) \in \mathbb{R}^{N_x}$ and $\mathbf{y}(t) \in \mathbb{R}^{N_y}$, respectively. The state $\mathbf{x}(0)$ is the *initial state*. The dynamics of the network is governed by the following equations:

$$\tilde{\mathbf{x}}(t+1) = f_{\text{res}}\big(\mathbf{W}_{\text{in}}\mathbf{u}(t+1) + \mathbf{W}_{\text{res}}\mathbf{x}(t) + \mathbf{b}_{\text{res}}\big) \tag{1}$$

$$\mathbf{x}(t+1) = (1-\alpha)\mathbf{x}(t) + \alpha\tilde{\mathbf{x}}(t+1) \tag{2}$$

$$\mathbf{y}(t+1) = f_{\text{out}}\big(\mathbf{W}_{\text{out}}\mathbf{x}(t+1) + \mathbf{b}_{\text{out}}\big) \tag{3}$$

where f_{res} and f_{out} are the *activation functions* of the reservoir and output units (applied component-wise), and α is the *leaking rate* $(0 \leq \alpha \leq 1)$.

The leaking rate α modulates the updating speed of the reservoir dynamics (cf. Equation (2)) [17]. The input weights $\mathbf{W}_{\text{in}}$ as well as the biases $\mathbf{b}_{\text{res}}$ and $\mathbf{b}_{\text{out}}$ are initialized randomly from uniform distributions $\mathcal{U}(-a, a)$ and $\mathcal{U}(-b, b)$, respectively, where a is the *input scaling* and b is the *bias scaling*. The input scaling affects the non-linearity of the reservoir dynamics [17]. The reservoir weights $\mathbf{W}_{\text{res}}$ are also initialized randomly from a uniform or a Gaussian distribution, then modified to have a given *sparsity* r, and finally rescaled such that the *spectral radius*[5] of the matrix $\mathbf{W} := (1-\alpha)\mathbf{I} + \alpha\mathbf{W}_{\text{res}}$ is equal to some given value ρ. In practice, taking $\rho < 1$ ensures that the *echo state property* – an asymptotic stability condition ensuring that a consistent learning can be achieved – is satisfied in most situations [7,14,17,18,32]. The spectral radius ρ regulates the effect of past inputs on the reservoir states: larger spectral radii being associated with longer input memories [17].

In an ESN, the input and reservoir weights $\mathbf{W}_{\text{in}}$ and $\mathbf{W}_{\text{res}}$ are kept fixed, and only the output weights $\mathbf{W}_{\text{out}}$ are trained. This feature render ESNs particularly computationally efficient. Notice that an ESN with N_x reservoir units contains only $|\mathbf{W}_{\text{out}}| + |\mathbf{b}_{\text{out}}| = N_y \times (N_x + 1)$ learning parameters (e.g., 2002 parameters for an ESN with 1000 reservoir units and 2 output units). Usually, the output weights $\mathbf{W}_{\text{out}}$ are computed by minimizing a loss function of the predictions and labels by means of a Ridge regression. However, any other supervised learning algorithm can be envisioned. In general, some initial transient of the ESN dynamics is used to *warm up* the reservoir, and the initial state of the reservoir modified accordingly [17].

Training paradigm. ESNs have been successfully applied to text classification tasks [1, 2]. In this framework, a *transformer-based ESN* consists of a 4-block model of the form:

EMBEDDING + RESERVOIR + POOLING + LEARNING ALGO.

More specifically, the model takes a tokenized text as input. The EMBEDDING layer embeds the successive tokens into corresponding input vectors. The

[5] The spectral radius of a matrix $\mathbf{W}$, denoted by $\rho(\mathbf{W})$, is the largest absolute value of the eigenvalues of $\mathbf{W}$.

RESERVOIR layer then passes the embedded inputs into a reservoir, producing a sequence of reservoir states (cf. Equations (1)-(2)). The POOLING layer merges the reservoir states into a single vector, which constitutes the *text embedding* per se. After all texts have been processed in this way, the LEARNING ALGO block takes all *text embeddings* together with their corresponding *labels* and fed them to a supervised learning algorithm, in order to learn the association between them. The whole process is illustrated in Figure 1.

The `EsnTorch` library implements the above mentioned model and training paradigm in an optimized way.

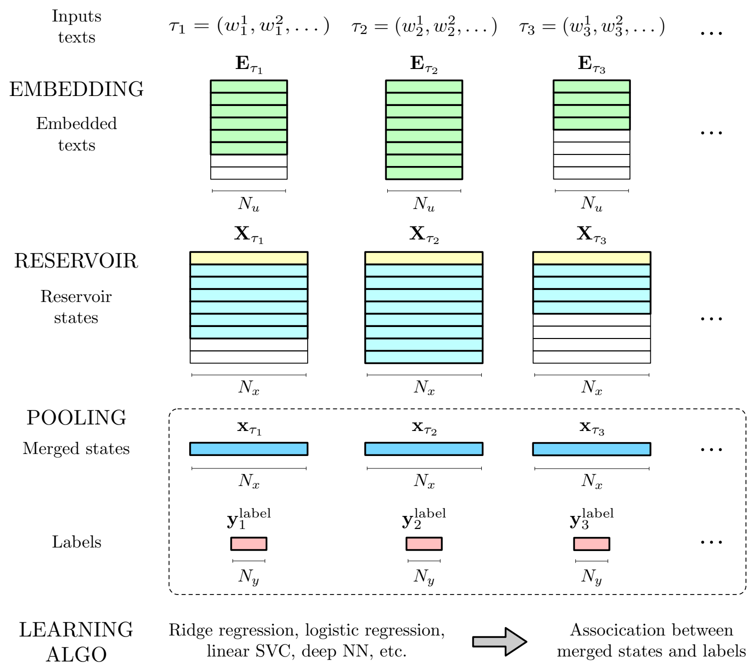

Fig. 1. Custom training paradigm of an ESN for a text classification task. Horizontal rectangles represent vectors. Empty rectangles represent null padding vectors used for batch parallelization. The ESN is composed of 4 blocks. **EMBEDDING:** each raw input text is tokenized and embedded into a sequence of input vectors (green rectangles). **RESERVOIR:** the input vectors (green rectangles) are passed through the ESN with "warm" initial state (yellow rectangle), yielding corresponding reservoir states (blue rectangles). **POOLING:** the reservoir states are merged into a single merged state (blue rectangle). The process is repeated for all input texts. **LEARNING ALGO:** a supervised learning algorithm is trained to learn the association between the merged states (blue rectangles) and their corresponding labels (red rectangles).

4 EsnTorch

The `EsnTorch` library is developed in `PyTorch`, optimized to work on GPU in a parallelized way, and operates in conjunction with the `transformers` and `datasets` libraries from Hugging Face [16, 31]. The required imports are the following:

```
import torch

from datasets import load_dataset, Dataset, concatenate_datasets
from transformers import AutoTokenizer
from transformers.data.data_collator import DataCollatorWithPadding

import esntorch.esn as esn
import esntorch.learning_algo as la
```

As described in Section 3, a transformer-based ESN for text classification consists of a 4-block model. `EsnTorch` implements each of these blocks (see Sections 4.1-4.4 below). In particular:

- `EsnTorch` has access to the 7K datasets provided by Hugging Face.
- The EMBEDDING layer can access the 60K transformer-based models provided by Hugging Face to embed the input texts.
- The RESERVOIR layer can implements three kinds of reservoirs: recurrent, linear or null.
- The POOLING layer offers three types of pooling strategies: mean, last, or None.
- The LEARNING ALGO block provides six different learning algorithms.

The next sections describe the dataset creation, as well as the instantiation, training and evaluation of a model in more details.

4.1 Dataset

The creation or download, tokenization, and preparation of a dataset and its dataloaders are achieved by means of the `datasets` library [16]. The code below illustrates the preparation of the TREC dataset (question classification). Here, the `bert-base-uncased` tokenizer is used. For compatibility purposes with our library, the `label` and `length` columns should be renamed by `labels` and `lengths`, respectively. Sorting the data by length significantly increases their processing speed. The batch size should be determined by the available memory.

```
# Load, tokenize and prepare dataset and dataloaders
tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')

def tokenize(sample):
    sample = tokenizer(sample['text'], truncation=True,
                                       padding=False,
                                       return_length=True)
    return sample

dataset = load_dataset('trec', split=None)
dataset = dataset.map(tokenize, batched=True)
dataset = dataset.rename_column('label-coarse', 'labels')
dataset = dataset.rename_column('length', 'lengths')
dataset = dataset.sort('lengths')
dataset.set_format(type='torch', columns=['input_ids', 'attention_mask',
                                          'labels', 'lengths'])

dataloader_d = {}

for k, v in dataset.items():
    dataloader_d[k] = torch.utils.data.DataLoader(v, batch_size=256,
        collate_fn=DataCollatorWithPadding(tokenizer))
```

4.2 Model

The instantiation of a model is achieved in four steps:

1. Set the parameters of the model;
2. Define a pooling strategy;
3. Define a learning algorithm;
4. Warm up the model, if needed.

The code below provides an example of instantiation of an ESN.

```
# ESN parameters
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

esn_params = {
            'embedding': 'bert-base-uncased', # model name
            'input_dim': 768,                 # embedding dim
            'distribution': 'uniform'         # 'uniform', 'gaussian'
            'dim': 1000,
            'sparsity': 0.9,
            'spectral_radius': 0.9,
            'leaking_rate': 0.5,
            'activation_function': 'tanh',    # 'tanh', 'relu'
            'input_scaling': 0.1,
            'bias_scaling': 0.1,
            'mean': 0.0,
            'std': 1.0,
            'pooling_strategy': 'mean',     # 'mean', 'last', None
            'learning_algo': None,          # initialized below
            'criterion': None,              # initialized below
            'optimizer': None,              # initialized below
            'bidirectional': False,         # True, False
            'mode' : 'recurrent_layer',     # 'no_layer',
                                            # 'linear_layer',
                                            # 'recurrent_layer'
            'deep' : False,                 # for deep esn
            'nb_layers' : None,             # if deep, nb of reservoirs
            'device': device,
            'seed': 42
            }

# Step 1: Instantiate the ESN
ESN = esn.EchoStateNetwork(**esn_params)

# Step 2: Instantiate the learning algo
ESN.learning_algo = la.RidgeRegression(alpha=10.0)

ESN = ESN.to(esn_params['device']) # put model on device

# Step 3: Warm up the ESN on 20 sentences
ESN.warm_up(dataset['train'].select(range(20)))
```

Regarding step 1, most parameters are self-explanatory and more details can be found in the documentation of the library. In particular, The `embedding` parameter is the name of the Hugging Face model used to embed the input texts (the list of all models can be found here). The `input_dim` should correspond to the dimension of this embedding. The next 10 parameters specify the reservoir characteristics: size, distribution, weights, etc. The `mean` and `std` parameters are only considered if `distribution = "gaussian"`. The `bidirection` parameter specifies whether the input texts are processed in a bidirectional way or not.

The `mode` parameter can take three different values

`"recurrent_layer"`, `"no_layer"` or `"linear_layer"`

which specifies the kind of reservoir to be considered. In the `"recurrent_layer"` mode, a recurrent reservoir is defined according to the previous parameters. In this case, a "classical" ESN is implemented, as described in Section 3. In the `"no_layer"` mode, no reservoir is considered, meaning that the input layer is directly fed to the learning algorithm. This feature allows to assess the proper contribution of the reservoir to the results, by shutting down the whole reservoir. In the `"linear_layer"` mode, a linear (i.e., non-recurrent) reservoir is implemented. This feature enables to evaluate the contribution of the recurrence of the reservoir, by removing this characteristics.

For step 2, the `pooling_strategy` parameter can take the three values

`"mean"`, `"last"`, or `None`

which correspond to three kinds of pooling layers. The `"mean"` and `"last"` pooling define the merged state as the mean and last of the reservoir states, respectively. The `None` pooling leaves the reservoir states unmerged (cf. [1,2,22] for further details). In general, the `"mean"` pooling performs significantly better than the others. Finally, the `deep` parameter implements a deep ESN and is discussed in further details in Section 4.5.

Regarding step 3, six learning algorithms divided two families can be considered:

(i) The ones for which there exists a closed-form solution (e.g. Ridge regression), or which are adapted from the `scikit-learn` library (e.g. Linear SVC):

`RidgeRegression(...)`, `RidgeRegression_skl(...)`, `LinearSVC(...)` and `LogisticRegression_skl(...)`.

In this case, only the algorithm is to be specified.

(ii) The ones that are trained via a gradient descent method implemented inside the library (e.g. logistic regression):

`LogisticRegression(...)` and `DeepNN(...)`.

In this case, in addition to the learning algorithm itself, a `pytorch` criterion and optimizer need to be given.

Example of a learning algorithms of type (i) and (ii) are given in the code snippet below. Custom learning algorithms can be added to the file `learning_alog.py`.

```
# Type (i) learning algo
ESN.learning_algo = la.LinearSVC(C=1.0)

# Type (ii) learning algo
ESN.learning_algo = la.LogisticRegression(input_dim=768, output_dim=6)
ESN.criterion = torch.nn.CrossEntropyLoss()
ESN.optimizer = torch.optim.Adam(ESN.learning_algo.parameters(),lr=0.01)
```

Once instantiated, the model is put on the required device (CPU or GPU) by means of the `.to(device)` method.

In step 4, the model is warmed up by passing a certain number of texts from the train set, and its initial state is modified accordingly. This is done by means of the `warm_up()` method.

4.3 Training

The training of the model is achieved via the `fit()` method. For training algorithms of type (i), no additional parameter is required. For training algorithms of type (ii), the number of epochs and number of steps after which the loss is printed can be specified (otherwise default values are used). The training process is illustrated in the code snippet below.

```
# For learning algo of type (i):
ESN.fit(dataloader_d["train"])

# For learning algo of type (ii):
ESN.fit(dataloader_d["train"], epochs=10, iter_steps=50)
```

4.4 Evaluation

After training, the predictions and accuracy on the train and test sets can be obtained by means of the `predict()` method. The predictions can further be used to compute the classification table of the model. These features are illustrated in the following code snippet.

```
# Train predictions and accuracy
train_pred, train_acc = ESN.predict(dataloader_d["train"])
train_pred, train_acc

(array([0, 3, 0, ..., 1, 1, 0]), 92.49816581071167)

# Test predictions and accuracy
test_pred, test_acc = ESN.predict(dataloader_d["test"])
test_pred, test_acc

(array([0, 0, 0, ..., 3, 1, 1]), 93.2)

# Classification report
from sklearn.metrics import classification_report

test_truth = dataset_d['test']['labels']
print(classification_report(test_pred.tolist(), test_truth.tolist()))

              precision    recall  f1-score   support
```

```
           0       0.99      0.91      0.95       151
           1       0.79      0.93      0.85        80
           2       0.78      1.00      0.88         7
           3       0.95      0.94      0.95        66
           4       0.98      0.97      0.97       115
           5       0.93      0.93      0.93        81

    accuracy                           0.93       500
   macro avg       0.90      0.94      0.92       500
weighted avg       0.94      0.93      0.93       500
```

4.5 Deep ESNs

Deep ESNs can also be implemented. The instatiation, training and evaluation of a deep ESN is similar to what has been described for a regular ESN (see Sections 4.2–4.4). The only difference relies in the instantiation part, where a group of reservoirs instead of a single one is instantiated. Towards this purpose, the `"deep"` parameter is set to `True` and the `"nb_layers"` to some integer $P \in \mathbb{N}$ that represents the desired number of reservoirs. Each other parameter related to the reservoir characteristics (like `"dim"`, `"sparsity"`, `"spectral_radius"`, etc.) is given either as a list of values `[v1,...,vP]` or as a single value `v`. In the former case, the successive reservoirs $R_1, \ldots, R_P$ of the deep ESN are built on the basis of the successive parameter values of `[v1,...,vP]`, respectively. In the latter case, all reservoirs $R_1, \ldots, R_P$ are constructed with respect to the same parameter value `v`.

5 Conclusion

We introduced `EsnTorch`, a user-friendly library that implements echo state networks with transformer-based embeddings for NLP applications. `EsnTorch` is developed in `PyTorch`, optimized to work on GPU, and operates in conjunction with the `transformers` and `datasets` libraries from Hugging Face: the major data science platform for NLP. The transformer-ESN model described in Section 3 has already been implemented with a previous version of this library, and shown promising results [1,2].

We believe that the combined transformer-ESN approach to NLP proposed in this work offers major advantages in terms of computational efficiency. Overall, this study falls within the context of sustainable models for NLP.

Acknowledgments

The authors are grateful to Playtika Ltd. for contributing to an inspiring R&D environment. The research was partially done with institutional support RVO: 67985807 and partially supported by the grant of the Czech Science Foundation AppNeCo No. GA22-02067S.

References

1. Cabessa, J., Hernault, H., Kim, H., Lamonato, Y., Levy, Y.Z.: Efficient text classification with echo state networks. In: International Joint Conference on Neural Networks, IJCNN 2021. pp. 1–8. IEEE (2021)
2. Cabessa, J., Lamonato, H.H.Y., Levy, Y.Z.: Combining bert and echo state networks for efficient text classification. Applied Intelligence (Submitted, 2022)
3. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: Burstein, J., Doran, C., Solorio, T. (eds.) Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, Volume 1, 2019. pp. 4171–4186. ACL (2019)
4. Di Sarli, D., Gallicchio, C., Micheli, A.: Question classification with untrained recurrent embeddings. In: et al., M.A. (ed.) AI*IA 2019, Proceedings. LNCS, vol. 11946, pp. 362–375. Springer (2019)
5. Dominey, P.F., Hoen, M., Inui, T.: A neurolinguistic model of grammatical construction processing. Journal of Cognitive Neuroscience **18**(12), 2088–2107 (2006)
6. Gallicchio, C., Micheli, A., Pedrelli, L.: Design of deep echo state networks. Neural Networks **108**, 33–47 (2018)
7. Gandhi, M., Jaeger, H.: Echo state property linked to an input: Exploring a fundamental characteristic of recurrent neural networks. Neural Computation **25**(3), 671–696 (2013)
8. Hinaut, X., Dominey, P.F.: Real-time parallel processing of grammatical structure in the fronto-striatal system: A recurrent network simulation study using reservoir computing. PLOS ONE **8**(2), 1–18 (02 2013)
9. Hinaut, X., Lance, F., Droin, C., Petit, M., Pointeau, G., Dominey, P.F.: Corticostriatal response selection in sentence production: Insights from neural network simulation with reservoir computing. Brain Lang **150**, 54–68 (2015)
10. Hinaut, X., Petit, M., Pointeau, G., Dominey, P.F.: Exploring the acquisition and production of grammatical constructions through human-robot interaction with echo state networks. Frontiers in Neurorobotics **8**, 16 (2014)
11. Hinaut, X., Twiefel, J.: Teach your robot your language! trainable neural parser for modeling human sentence processing: Examples for 15 languages. IEEE Trans. Cogn. Dev. Syst. **12**(2), 179–188 (2020)
12. Jaeger, H.: Short term memory in echo state networks. GMD-Report 152, GMD - German National Research Institute for Computer Science (2002)
13. Jaeger, H.: Echo state network. Scholarpedia **2**(9), 2330 (2007)
14. Jaeger, H.: The "echo state" approach to analysing and training recurrent neural networks. GMD Report 148, GMD - German National Research Institute for Computer Science (2001)
15. Jaeger, H., Haas, H.: Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. Science **304**(5667), 78–80 (2004)
16. Lhoest, Q., Villanova del Moral, A., Jernite, Y., Thakur, A., von Platen, P., Patil, S., Chaumond, J., Drame, M., Plu, J., Tunstall, L., Davison, J., Šaško, M., Chhablani, G., Malik, B., Brandeis, S., Le Scao, T., Sanh, V., Xu, C., Patry, N., McMillan-Major, A., Schmid, P., Gugger, S., Delangue, C., Matussière, T., Debut, L., Bekman, S., Cistac, P., Goehringer, T., Mustar, V., Lagunas, F., Rush, A., Wolf, T.: Datasets: A community library for natural language processing. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. pp. 175–184. ACL (2021)

17. Lukoševičius, M.: A Practical Guide to Applying Echo State Networks, pp. 659–686. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
18. Lukoševičius, M., Jaeger, H.: Reservoir computing approaches to recurrent neural network training. Computer Science Review **3**(3), 127–149 (2009)
19. Ramamurthy, R., Stenzel, R., Sifa, R., Ladi, A., Bauckhage, C.: Echo state networks for named entity recognition. In: Tetko, I.V., Kůrkovà, V., Karpov, P., Theis, F. (eds.) ICANN 2019, Proceedings. pp. 110–120. No. 11731 in LNCS, Springer (2019)
20. Sanh, V., Debut, L., Chaumond, J., Wolf, T.: Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. CoRR **abs/1910.01108** (2019)
21. Schaetti, N.: Echotorch: Reservoir computing with pytorch. https://github.com/nschaetti/EchoTorch (2018)
22. Schaetti, N.: Behaviors of reservoir computing models for textual documents classification. In: International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019. pp. 1–7. IEEE (2019)
23. Shen, S., Baevski, A., Morcos, A., Keutzer, K., Auli, M., Kiela, D.: Reservoir transformers. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). pp. 4294–4309. ACL, Online (2021)
24. Shrivastava, H., Garg, A., Cao, Y., Zhang, Y., Sainath, T.N.: Echo state speech recognition. In: IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2021. pp. 5669–5673. IEEE (2021)
25. Steiner, P., Jalalvand, A., Stone, S., Birkholz, P.: Pyrcn: A toolbox for exploration and application of reservoir computing networks (2021)
26. Sun, Z., Yu, H., Song, X., Liu, R., Yang, Y., Zhou, D.: Mobilebert: a compact task-agnostic BERT for resource-limited devices. In: Jurafsky, D., Chai, J., Schluter, N., Tetreault, J.R. (eds.) Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020. pp. 2158–2170. ACL (2020)
27. Tong, M.H., Bickett, A.D., Christiansen, E.M., Cottrell, G.W.: Learning grammatical structure with echo state networks. Neural Networks **20**(3), 424–432 (2007)
28. Trouvain, N., Pedrelli, L., Dinh, T.T., Hinaut, X.: Reservoirpy: An efficient and user-friendly library to design echo state networks. In: Farkas, I., Masulli, P., Wermter, S. (eds.) Artificial Neural Networks and Machine Learning - ICANN 2020 - 29th International Conference on Artificial Neural Networks, Proceedings, Part II. LNCS, vol. 12397, pp. 494–505. Springer (2020)
29. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017. pp. 5998–6008 (2017)
30. Wieting, J., Kiela, D.: No training required: Exploring random encoders for sentence classification. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net (2019)
31. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T.L., Gugger, S., Drame, M., Lhoest, Q., Rush, A.M.: Transformers: State-of-the-art natural language processing. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. pp. 38–45. ACL, Online (Oct 2020)
32. Yildiz, I.B., Jaeger, H., Kiebel, S.J.: Re-visiting the echo state property. Neural Networks **35**, 1–9 (2012)